**Subject**

**Software Engineering**

**Ch-2**

**Software Requirement**

**Engg.**

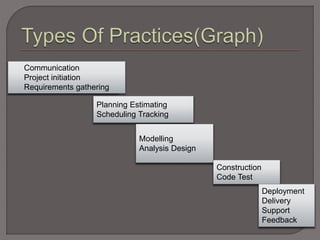
**# Core Principles of Software Engineering Practices:**

The following are the principles and tactics we must employ to stay grounded and to make reasonable technical choices based upon requirements, budgets, timelines, and expectations. Living by these principles will help your project progress seamlessly.

**The First Principle: The Reason It All Exists**  
A software system exists for one reason: to value to its users. All decisions should be made with this in mind. Before specifying a system requirement, before noting a piece of system functionality, before determining the hardware platforms or development processes, ask yourself questions such as: Does this add real value to the system? If the answer is no, don’t do it. All other principles support this one.  
  
**The Second Principle: KISS (Keep It Simple, Stupid)**  
 All design should be as simple as possible, but no simpler. These facilities having a more easily understood and easily maintained system. This is not to say that features, even internal features, should be discarded in the name of simplicity. Indeed, the more elegant designs are usually the simple ones. Simple also does not mean “quick and dirty.” In fact, it often, it often takes a lot of thought and works over multiple iterations to simplify. The pay-off is software that is more maintainable and less error-prone.  
  
**The Third Principle: Maintain the Vision**  
A clear vision is essential to the success of a software project. Without one, a project almost unfailingly ends up being “of two [or more] minds” about itself. Without conceptual integrity, a system threatens to become a patchwork of incompatible designs, held together by the wrong kind of screws…  
  
Compromising the architectural vision of a software system weakens and will eventually break even a well-designed the system. Having an empowered architect who can hold the vision and enforce compliance helps ensure a very successful software project.  
  
**The fourth principle: what you produce, others will consume**  
In some way or other, someone else will use, maintain, document, or otherwise depend on being able to understand your system. So always specify, design and implement knowing someone else will have to understand what you are doing. The audience for any product of software development is potentially large. Specify with an eye to the users. Design, keeping the implementers in mind.Code with concern for those who must maintain and extend the system. Someone may have to debug the code you writ, and that makes them a user of your code. Making their job easier adds value to the system.  
  
**The fifth principles: Be Open to the future**  
A system with a long lifetime has more value. In today’s computing environments, where specifications change on a moment’s notice and hardware platforms are obsolete after just a few months, software lifetimes are typically measured in months instead of years, however , true “ industrial- strength” software systems must endure far longer. To do this successfully, these systems must be ready to adapt to these and other changes. Systems that do this successfully are those that have been designed this way from the start. Never design yourself into a corner. Always ask “what if”, and prepare for all possible answers by creating systems that solve the general problem, not just the specific on. Author’s note: This advice can be dangerous if it is taken to extremes. Designing for the “general problem” sometimes requires performance compromises and can require more project effort. This could very possibly lead to the reuse of an entire system.  
  
**The Sixth Principle: Think!**  
This last principle is probably the most overlooked. Placing clear, complete thought before action almost always produces better results. When you think about something, you are more likely to do it right. You also gain knowledge about how to do it right again. If you do think about something and still do it wrong, it becomes valuable experience. A side effect of thinking is learning to recognize when you don’t know something, at which point you can research the answer. When clear thought has gone into a system, value comes out. Applying the first Six Principles requires intense thought, for which the potential rewards are enormous.  
  
**# Types of SE Practice:**

There are 6 types of SE practice,

1. Communication Practice
2. Planning Practice
3. Modeling Practice
4. Construction Practice
5. Testing Practice
6. Deployment Practice



# 1] COMMUNICATION PRACTICES:

Before customer requirements can be analyzed, modeled, or specified they must be gathered through a communication (also called requirements elicitation) activity. A customer has a problem that may be amenable to a computer- based solution. A developer responds to the customer’s request for help. Communication has begun. But the road from communication to understanding is often full of potholes.  
  
Effective communication (among technical peers, with the customer and other stakeholders, and with project managers) is among the most challenging activities that confront software engineer. In this context, we discuss communication principles that apply equally to all forms of communication that occur within a software project.  
  
**1: Listen.**  
Try to focus on the speaker’s words, rather than formulating your response to those words. Ask for clarification if something is unclear, but avoid constant interruptions. Never become contentions in your words or actions 9 e.g., rolling your eyes or shaking your head) as a person is talking.  
  
**2: Prepare before you communicate.**  
Spend the time to understand the problem before you meet with others. If necessary, do some research to understand business domain jargon. If you have responsibility for conducting a meeting, prepare an agenda in advance of the meeting.  
  
**3: Face–to-face communication is best.**  
But it usually works better when some other representation of the relevant information is present. For example, a participant may create a drawing or a “strawman” document that serves as a focus for discussion.  
  
**4: Take notes and documentation decisions:**  
Things have a way of falling into the cracks. Someone participating in the communication should serves to build trust among team members and creates a common goal for the team.  
  
**5: Stay focused, modularize your discussion.**  
The more the people involved in any communication, the more likely that discussion will bounce from one topic to the next. The facilitator should keep the conversation modular, leaving one topic to the next. The facilitator should keep the conversation modular, leaving one topic only after it has been resolved .  
  
**6: If something is unclear, draw a picture.**  
Verbal communication goes only so far. A sketch or drawing can often provide clarity when words fail to do the job.  
  
**: Negotiation is not a contest or a game. It works best when both parties win.**  
There are many instances in which the software engineer and the customer must negotiate functions and features, priorities, and delivery dates. If the team has collaborated well, all parties have a common goal. Therefore, negotiation will demand compromise from all parties.

# 2] PLANNING PRACTICES

The communication activity helps a software team to define its overall goals and objectives (subject, of course, to change as time passes). However, understanding these goals and objectives is not the same as defining a plan for getting there. The planning activity encompasses a set of management and technical practices that enable the software tam to define a road map as it travels towards its strategic goal and technical objectives.  
  
**1: Understand the scope of the project.** It’s impossible to use a road map if you don’t know where you’re going. Scope provides the software.  
  
**2: Involve the customer in planning activity.** The customer defines priorities and establishes the project constraints.  
  
**3: Recognize that planning is iterative.** As work begins, it is very likely that things will change. As a consequence, the plan must be adjusted to accommodate these changes. In addition, iterative and incremental process models dictate re-planning based on feedback received from users.  
  
**4: Consider risk as you define the plan.** If the team has defined risks that have high impact and high probability, contingency planning is necessary.  
  
**5: Track the plan frequently and make adjustments are required.** Software project falls behind schedule one day at a time. Therefore, it makes sense to track progress on a daily basis, looking for a problem areas and situation in which scheduled work does not confirm to actual work conducted. When slippage is encountered, the plan is adjusted accordingly.  
  
**6:How much of each resource is needed?** The answer to this question is derived by developing estimates based on answers to earlier questions.

# 3] MODELING PRACTICE:

The models are created to gain better understanding of actual entity to be built. When the entity is a physical thing, we can build a model that is identical in form of shape but smaller in scale. However, when the entity is software, our model must take a different form. It must be capable of representing the information that software transforms, the architecture and functions that enable the transformation to occur, the features that user’s desire, and the behavior of the system as the transformation is taking place.

**1: The information domain of a problem must be represented and understood.** The information domain compasses the data that flow into the system and the data stores that collect and organize persistent data objects.

**2: The functions that the software performs must be defined.** Software functions provide direct benefit to visible end-user. Some functions transform data that flow into the system; in other cases, functions effect some level of control over internal software processing or external system elements.  
**3: The behavior of the software must be represented.** The behavior of computer software is driven by its interaction with the external environment. Input provided by end-users, control data provided by an external system, or monitoring data collected over a network all cause the software to behave in a specific way.  
  
**4: The models that depict information, function, and behavior.** Analysis modeling is the first step in software engineering problem solving. It allows the practitioner to understand the problem better and establishes a basis for the solution (design). Complex problems are difficult to solve in their entirety. For this reason, we use a divide and conquer strategy. A large, complex problem is divided into sub-problems until each sub- problem is relatively easy to understand. This concept is called partitioning, and it is a key strategy in analysis modeling.  
  
**5: The analysis task should move from essential information toward implementation detail.** Analysis modeling begins by describing the problem from the end-user’s perspective. The “essence” of a problem is described without any consideration of how a solution will be implemented.

# 4] CONSTRUCTION PRACTICE:

The construction activity encompasses a set of coding and testing task that lead operational software that is ready for delivery to the customer or end-user. In modern software engineering work.

 As you begin writing code, be sure you:  
1. Constraint your algorithm by following structured programming practice.  
2. Select data structure that will meet the needs of the design.  
3. Understand the software architecture and create interfaces that are consistent.  
4. Keep conditional statement as simple as possible.  
5. Create nested loops in a way that makes them easily testable.  
6. Select meaningful variable names and follow other local coding standards.  
7. Write code that is self-documenting.  
8. Create a visual layout that aids understanding.

**5] Testing Practice:**

Testing in a process of executing with the intent of finding an error.  
• A good test case is one that has a high probability of finding as as-yet undiscovered error.  
• A successful test is one that uncovers an as- yet –undiscovered error.

**1: All tests should be traceable to customer requirements.** The objective of software testing is to uncover errors. It follows that thee most server defects (from the customer’s point of view) are those that cause the program to fail to meet its requirements/goals.  
**2: Tests should be planned long before testing begins.** Test planning can began as soon as the analysis model is complete. Detailed definition of test cases can begin as soon as the design model has been solidified. Therefore, all tests can be planned and designed before any code has been generated.  
**4: Testing should begin “in the small” and progress toward testing “in the large”.** The first tests planned and executed generally focus on individual components. As testing progresses, focus shifts in an attempt to find error in integrated clusters of components and ultimately in the entire system.  
**5: Exhaustive testing is not possible.** The number of path permutations for even a moderately sized program is exceptionally large. For this reason, it is impossible to execute every combination of paths during testing. It is possible, however, to adequately cover program logic and to ensure that all conditions in the component- level design have been exercised.

## 6] Deployment Practice:

The deployment activity encompasses three actions delivery, support, and feedback. Because modern software process models are evolutionary in nature, deployment happens not once, but a number of times as software moves towards completion. Each delivery cycle provides the customer and end-users with an operational software increment that provides usable functions and features. The delivery of a software increment represents an important milestone for any software project. A number of key principles should be followed as the team prepares to deliver an increment:  
**1: Customer expectations for the software must be managed.** The customer expects more than the team has promised to deliver and disappointment occurs immediately. This results in feedback that is not productive and which ruins team morale.  
**2: A complete delivery package should be assembled and tested.** A CD\_ ROM or other media containing all executable software, support data files, support document, and other relevant information must be assembled and thoroughly beta- tested with actual users.  
  
**3: A support regime must be established before the software is delivered.** An end-user expects responsiveness and accurate information when a question or problem arises. Support should be planned, support material should be prepared, and appropriate record keeping mechanism should be established so that the software team can conduct a categorical assessment of the kinds of support requested required.  
**4. Appropriate instructional materials must be provided to end-users.** The software team delivers more than the software itself. Appropriate training aids should be developed, trouble-shooting guidelines should be provided and a “what’s- different about- this-software-increment” description should be published.  
**5: Buggy software should be fixed first, delivered later.** Under time pressure, some software organizations deliver low-quality increments with a warning to the customer that bugs “will be fixed in the next release”. This is a mistake. There’s a saying in the software business: “Customer will forget you delivered a high- quality product a few days late, but they will never the problems that a low-quality product caused them. The software reminds them every day.”

## #Introduction to requirement engineering

* The process of collecting the software requirement from the client then understand, evaluate and document it is called as requirement engineering.
* Requirement engineering constructs a bridge for design and construction.

**Requirement engineering consists of seven different tasks as follow:**  
  
**1. Inception**

* Inception is a task where the requirement engineering asks a set of questions to establish a software process.
* In this task, it understands the problem and evaluates with the proper solution.
* It collaborates with the relationship between the customer and the developer.
* The developer and customer decide the overall scope and the nature of the question.

**2. Elicitation**

Elicitation means to find the requirements from anybody.  
The requirements are difficult because the **following problems occur in elicitation**.  
  
**Problem of scope:** The customer give the unnecessary technical detail rather than clarity of the overall system objective.  
  
**Problem of understanding:** Poor understanding between the customer and the developer regarding various aspect of the project like capability, limitation of the computing environment.  
  
**Problem of volatility:** In this problem, the requirements change from time to time and it is difficult while developing the project.  
  
**3. Elaboration**

* In this task, the information taken from user during inception and elaboration and are expanded and refined in elaboration.
* Its main task is developing pure model of software using functions, feature and constraints of a software.

**4. Negotiation**

* In negotiation task, a software engineer decides the how will the  project be achieved with limited business resources.
* To create rough guesses of development and access the impact of the requirement on the project cost and delivery time.

**5. Specification**

* In this task, the requirement engineer constructs a final work product.
* The work product is in the form of software requirement specification.
* In this task, formalize the requirement of the proposed software such as informative, functional and behavioral.
* The requirement are formalize in both graphical and textual formats.

**6. Validation**

* The work product is built as an output of the requirement engineering and that is accessed for the quality through a validation step.
* The formal technical reviews from the software engineer, customer and other stakeholders helps for the primary requirements validation mechanism

**7. Requirement management**

* It is a set of activities that help the project team to identify, control and track the requirements and changes can be made to the requirements at any time of the ongoing project.
* These tasks start with the identification and assign a unique identifier to each of the requirement.
* After finalizing the requirement traceability table is developed.
* The examples of traceability table are the features, sources, dependencies, subsystems and interface

**Types of requirements:**

**A software requirement can be of 2 types:**

* Functional requirements
* Non-functional requirements

**Product Requirement**

**Organizational Requirement**

**External Requirements**

**Non-functional requirements**

**Functional requirements**

**Types of requirements**

**- Performance - Process Policy -Inter Operability**

**- Reliability - Implementation Policy -Ethics/Standers**

**- Portability- delivery Policy -Privacy**

**- Usability - Procedure**

**1] Functional Requirements:**

These are the requirements that the end user specifically demands as basic facilities that the system should offer.

It can be a calculation, data manipulation, business process, user interaction, or any other specific functionality which defines what function a system is likely to perform.

All these functionalities need to be necessarily incorporated into the system as a part of the contract.

These are represented or stated in the form of input to be given to the system, the operation performed and the output expected.

**2] Non-functional requirements:**

 These are basically the quality constraints that the system must satisfy according to the project contract.Nonfunctional requirements, not related to the system functionality, rather define how the system should perform .

The priority or extent to which these factors are implemented varies from one project to other. They are also called non-behavioral requirements.

They basically deal with issues like:

* Portability
* Security
* Maintainability
* Reliability
* Scalability
* Performance
* Reusability
* Flexibility

**# Requirements Engineering Tasks:**

[Requirements engineering](https://www.geeksforgeeks.org/software-engineering-requirements-engineering-process/) is the most important part every business must follow, in order to build and release a project successfully, as it is the foundation to key planning and implementation.

The software requirements engineering process includes the following steps of activities:

1. **Inception**
2. **Elicitation**
3. **Elaboration**
4. **Negotiation**
5. **Specification**
6. **Inception:**

This is the first phase of the requirements analysis process. This phase gives an outline of how to get started on a project. In the inception phase, all the basic questions are asked on how to go about a task or the steps required to accomplish a task. A basic understanding of the problem is gained and the nature of the solution is addressed. Effective communication is very important in this stage, as this phase is the foundation as to what has to be done further. Overall in the inception phase, the following criteria have to be addressed by the software engineers:

1. Understanding of the problem.
2. The people who want a solution.
3. Nature of the solution.
4. Communication and collaboration between the customer and developer.
5. **Elicitation:**

 This is the second phase of the requirements analysis process. This phase focuses on gathering the requirements from the stakeholders. One should be careful in this phase, as the requirements are what establishes the key purpose of a project. Understanding the kind of requirements needed from the customer is very crucial for a developer. In this process, mistakes can happen in regard to, not implementing the right requirements or forgetting a part. The right people must be involved in this phase.

The following problems can occur in the elicitation phase:

* **Problem of Scope:**The requirements given are of unnecessary detail, ill-defined, or not possible to implement.
* **Problem of Understanding:** Not having a clear-cut understanding between the developer and customer when putting out the requirements needed. Sometimes the customer might not know what they want or the developer might misunderstand one requirement for another.
* **Problem of Volatility:**Requirements changing over time can cause difficulty in leading a project. It can lead to loss and wastage of resources and time.

1. **Elaboration:**

This is the third phase of the requirements analysis process. This phase is the result of the inception and elicitation phase. In the elaboration process, it takes the requirements that have been stated and gathered in the first two phases and refines them. Expansion and looking into it further are done as well.  The main task in this phase is to indulge in modeling activities and develop a prototype that elaborates on the features and constraints using the necessary tools and functions.

1. **Negotiation:**

This is the fourth phase of the requirements analysis process. This phase emphasizes discussion and exchanging conversation on what is needed and what is to be eliminated. In the negotiation phase, negotiation is between the developer and the customer and they dwell on how to go about the project with limited business resources. Customers are asked to prioritize the requirements and make guesstimates on the conflicts that may arise along with it. Risks of all the requirements are taken into consideration and negotiated in a way where the customer and developer are both satisfied with reference to the further implementation.

The following are discussed in the negotiation phase:

* + 1. Availability of Resources.
    2. Delivery Time.
    3. Scope of requirements.
    4. Project Cost.
    5. Estimations on development.

**5. Specification:**

This is the fifth phase of the requirements analysis process. This phase specifies the following:

* Written document.
* A set of models.
* A collection of use cases.
* A prototype.

In the specification phase, the requirements engineer gathers all the requirements and develops a working model. This final working product will be the basis of any functions, features or constraints to be observed. The models used in this phase include [ER (Entity Relationship) diagrams](https://www.geeksforgeeks.org/introduction-of-er-model/), [DFD (Data Flow Diagram)](https://www.geeksforgeeks.org/what-is-dfddata-flow-diagram/), FDD (Function Decomposition Diagrams), and [Data Dictionaries](https://www.geeksforgeeks.org/data-dictionaries-in-software-engineering/).  
A software specification document is submitted to the customer in a language that he/she will understand, to give a glimpse of the working model.

## # Software Requirement Specification - [SRS]:

## A software requirements specification (SRS) is a document that captures complete description about how the system is expected to perform. It is usually signed off at the end of requirements engineering phase.

## Qualities of SRS:

* Correct
* Unambiguous
* Complete
* Consistent
* Ranked for importance and/or stability
* Verifiable
* Modifiable
* Traceable

A software requirements specification (SRS) is a comprehensive description of the intended purpose and environment for [software](https://www.techtarget.com/searchapparchitecture/definition/software) under development. The SRS fully describes what the software will do and how it will be expectedto perform.

# Characteristics of SRS:

1. **Correctness:**  
   User review is used to ensure the correctness of requirements stated in the SRS. SRS is said to be correct if it covers all the requirements that are actually expected from the system.
2. **Completeness:**  
   Completeness of SRS indicates every sense of completion including the numbering of all the pages, resolving the to be determined parts to as much extent as possible as well as covering all the functional and non-functional requirements properly.
3. **Consistency:**   
   Requirements in SRS are said to be consistent if there are no conflicts between any set of requirements. Examples of conflict include differences in terminologies used at separate places, logical conflicts like time period of report generation, etc.
4. **Unambiguousness:**   
   A SRS is said to be unambiguous if all the requirements stated have only 1 interpretation. Some of the ways to prevent unambiguousness include the use of modelling techniques like ER diagrams, proper reviews and buddy checks, etc.
5. **Modifiability:**   
   SRS should be made as modifiable as possible and should be capable of easily accepting changes to the system to some extent. Modifications should be properly indexed and cross-referenced.
6. **Verifiability:**   
   A SRS is verifiable if there exists a specific technique to quantifiably measure the extent to which every requirement is met by the system. For example, a requirement starting that the system must be user-friendly is not verifiable and listing such requirements should be avoided.
7. **Traceability:**   
   One should be able to trace a requirement to design component and then to code segment in the program. Similarly, one should be able to trace a requirement to the corresponding test cases.

## ****Testability:****  A SRS should be written in such a way that it is easy to generate test cases and test plans from the document.

### # Benefits [Needs]of SRS:

* The users and the client get a brief idea about the software while in the initial stages.
* The purposes and the intentions as well as the expected results are properly defined. It hence lays the outline for software design.
* The desired goals are defined thereby easing off the efforts of the developers in terms of time and cost.
* It forms a basis for the agreement between the client and the developer.
* It becomes easier while transferring and using the solution elsewhere or with new customers as the basis of functioning of the software is mentioned.
* It acts as a material for reference at a later stage.
* It acts as the basis for reviews.

### #Components of SRS:

**Table of Contents**

**1. Introduction**

1.1 Purpose of this document

1.2 Scope of this document

1.3 Overview

1.4 Business Context

**2. General Description**

2.1 Product Functions

2.2 Similar System Information

2.3 User Characteristics

2.4 User Problem Statement

2.5 User Objectives

2.6 General Constraints

**3. Functional Requirements**

**4. Interface Requirements**

4.1 User Interfaces

4.2 Hardware Interfaces

4.3 Communications Interfaces

4.4 Software Interfaces

**5. Performance Requirements**

**6. Other non-functional attributes**

6.1 Security

6.3 Reliability

6.4 Maintainability

6.5 Portability

6.6 Extensibility

6.7 Reusability

6.8 Application Affinity/Compatibility

**7. Operational Scenarios**

**8. Preliminary Use Case Models and Sequence Diagrams**

8.1 Use Case Model

8.2 Sequence Diagrams

**9. Updated Schedule**

**10. Appendices**

10.1 Definitions, Acronyms, Abbreviations

10.2 References