Chap 02: Software Engineering Practices and Software Requirements Engineering

1. Software Engineering Practices:

Generic framework activities—**communication, planning, modeling, construction,** and **deployment**—and umbrella activities establish a basic architecture for software engineering work. Here, you will understand the generic concepts and principles that apply to framework activities.

1.1 The Essence of Practice

1. Understand the problem (communication and analysis).
2. Plan a solution (modeling and software design).
3. Carry out the plan (code generation).
4. Examine the result for accuracy (testing and quality assurance).

Understand the problem.

Spend a little time answering a few simple questions:

- Who has a stake in the solution to the problem? That is, who are the stakeholders?
- What are the unknowns? What data, functions, and features are required to properly solve the problem?
- Can the problem be compartmentalized? Is it possible to represent smaller problems that may be easier to understand?
- Can the problem be represented graphically? Can an analysis model be created?

Plan the solution.

Before you begin coding, do a little design:

- Have you seen similar problems before? Is there existing software that implements the data, functions, and features that are required?
- Has a similar problem been solved? If so, are elements of the solution reusable?
- Can sub problems be defined? If so, are solutions ready for the sub problems?
- Can you represent a solution in a manner that leads to effective implementation? Can a design model be created?

Carry out the plan.

- Does the solution conform to the plan? Is source code traceable to the design model?
- Is each component part of the solution provably correct? Have the design and code been reviewed, or better, have correctness proofs been applied to the algorithm?

Examine the result.

- Is it possible to test each component part of the solution? Has a reasonable testing strategy been implemented?
- Does the solution produce results that conform to the data, functions, and features that are required? Has the software been validated against all stakeholder requirements?

2. Core Principles of Software Engineering

The First Principle: The Reason It All Exists

- A software system exists for one reason: to provide value to its users. All decisions should be made with this in mind.
- Before specifying a system requirement, system functionality, before determining the hardware platforms, first determine, whether it adds value to the system.

The Second Principle: KISS (Keep It Simple, Stupid!)

- All design should be as simple as possible, but no simpler. This facilitates having a more easily understood and easily maintained system.
- It doesn't mean that features should be discarded in the name of simplicity.
- Simple also does not mean "quick and dirty." In fact, it often takes a lot of thought and work over multiple iterations to simplify.

The Third Principle: Maintain the Vision

- A clear vision is essential to the success of a software project.
- If you make compromise in the architectural vision of a software system, it will weaken and will eventually break even the well-designed systems.
- Having a powerful architect who can hold the vision helps to ensure a very successful software project.

The Fourth Principle: What You Produce, Others Will Consume

- Always specify, design, and implement by keeping in mind that someone else will have to understand what you are doing.
- The audience for any product of software development is potentially large.
- Design (make design), keeping the implementers (programmers) in mind. Code (program) with concern for those who will maintain and extend the system.
- Someone may have to debug the code you write, and that makes them a user of your code.

The Fifth Principle: Be Open to the Future

- A system with a long lifetime has more value.
- True "industrial-strength" software systems must last for longer.
- To do this successfully, these systems must be ready to adapt changes.
- Always ask "what if," and prepare for all possible answers by creating systems that solve the general problem.

The Sixth Principle: Plan Ahead for Reuse

- Reuse saves time and effort.
- The reuse of code and designs has a major benefit of using object-oriented technologies.
- Planning ahead for reuse reduces the cost and increases the value of both the reusable components and the systems into which they are incorporated.

The Seventh principle: Think!

- 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 a valuable experience.
- Applying the first six principles requires intense thought, for which the potential rewards are enormous.

3. Communication Practices/Principles.

Software requirement always begins with communication between two or more parties.
In communication principles, the initial stage is to collect or gather requirement from customer.

Principle 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 contentious in your words or actions (e.g., rolling your eyes or shaking your head) as a person is talking.

Principle 2 Prepare before you communicate:

- Spend the time to understand the problem before you meet with others. If necessary, perform some research to understand business domain.
- If you have responsibility for conducting a meeting, prepare an agenda in advance of the meeting.

Principle 3 someone should facilitate the activity:

- Every communication meeting should have a leader (a facilitator)

(1) To keep the conversation moving in a productive direction,
(2) To mediate any conflict that does occur, and
(3) To ensure that other principles are followed.

Principle 4 Face-to-face communication is best:

- It usually works better when some other representation of the relevant information is present.
- For example, a participant may create a drawing /document that serve as a focus for discussion.

Principle 5 Take notes and document decisions:

- Someone participating in the communication should serve as a recorder and write down all important points and decisions.

Principle 6 Strive for collaboration:

- Collaboration occurs when the collective knowledge of members of the team is used to describe product or system functions or features.
- Each small collaboration builds trust among team members and creates a common goal for the team.

Principle 7 Stay focused; modularize your discussion:

- The more 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 only after it has been resolved.

Principle 8 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.

Principle 9

(a) Once you agree to something, move on.

(b) If you can't agree to something, move on.

(c) If a feature or function is unclear and cannot be clarified at the moment, move on.

- The people who participate in communication should recognize that many topics require discussion and that moving on is sometimes the best way to achieve communication agility.

Principle 10 Negotiation is not a contest or a game: It works best when both parties win.

- There are many instances in which you and other stakeholders must negotiate functions and features, priorities, and delivery dates.
- If the team has collaborated well, all parties have a common goal. Still, negotiation will demand compromise from all parties.

4. Planning Practices

Planning activity establishes a plan for software engineering work that follows. Planning describes the technical tasks to be conducted, the resources that will be required, and the risks that are likely the work products to be produced.

Principle 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 team with a destination.

Principle 2: Involve stakeholders in the planning activity.

- Stakeholders define priorities and establish project constraints.
- To accommodate these realities, software engineers must often negotiate order of delivery, time lines, and other project-related issues.

Principle 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, incremental process models dictate replanning after the delivery of each software increment based on feedback received from users.

Principle 4: Estimate based on what you know.

- The intent of estimation is to provide an indication of effort, cost, and task duration, based on the team's current understanding of the work to be done.
- If information is unreliable, estimates will be equally unreliable.

Principle 5: Consider risk as you define the plan.

- If you have identified risks that have high impact and high probability, contingency planning is necessary.
- In addition, the project plan (including the schedule) should be adjusted to accommodate the thought that one or more of these risks will occur.

Principle 6: Be realistic.

- People don't work 100 percent of every day.
- Noise always enters into any human communication. Omissions and ambiguity are facts of life. Change will occur. Even the best software engineers make mistakes.
- These and other realities should be considered as a project plan is established.

Principle 7: Adjust granularity as you define the plan.

- Granularity refers to the level of detail that is introduced as a project plan is developed.
- A "high-granularity" plan provides significant work task detail that is planned over short time increments.
- A "low-granularity" plan provides broader work tasks that are planned over longer time periods.

Principle 8: Define how you intend to ensure quality.

- The plan should identify how the software team intends to ensure quality.
- If technical reviews are to be conducted, they should be scheduled.
- If pair programming is to be used during construction, it should be explicitly defined within the plan.

Principle 9: Describe how you intend to accommodate change.

- You should identify how changes are to be accommodated as software engineering work proceeds.
- For example, can the customer request a change at any time? If a change is requested, is the team ready enough to implement it immediately? How is the impact and cost of the change assessed?

Principle 10: Track the plan frequently and make adjustments as required.

- Software projects fall behind schedule one day at a time.
- Therefore, it makes sense to track progress on a daily basis, looking for problem areas and situations in which scheduled work does not conform to actual work conducted.

5. Modeling Principles

- We create models to gain a better understanding of the actual entity to be built.
- When entity to be built 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 desire, and the behavior of the system as the transformation is taking place.
- Models must accomplish these objectives at different levels of abstraction: first depicting the software from the customer's viewpoint and later representing the software at a more technical level.
- In software engineering work, two classes of models can be created: **requirements (analysis) models and design models.**

5.1 Analysis Modeling Principles

Requirements models (also called analysis models) represent customer requirements by depicting the software in three different domains:

- The information domain,
- The functional domain, and
- The behavioral domain.

Principle 1: The information domain of a problem must be represented and understood.

- The information domain encompasses the data that flow into the system, the data that flow out of the system, and the data stores that collect and organize persistent data objects.

Principle 2: The functions that the software performs must be defined.

- Software functions provide benefit to end users and also provide internal support for those features that are user visible. Some functions transform data that flow into the system.
- Functions can be described at many different levels of abstraction, ranging from a general statement of purpose to a detailed description of the processing elements

Principle 3: The behavior of the software (as a consequence of external events) must be represented.

- The computer software shows its behavior 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.

Principle 4: The models that depict information function and behavior must be partitioned in a manner that uncovers detail in a layered (or hierarchical) fashion.

- Requirement modeling allows you to better understand the problem and establishes a basis for the solution (design).
- Complex problems are difficult to solve as a whole.
- For this reason, you should 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 or separation of concerns, and it is a key strategy in requirements modeling.

Principle 5: The analysis task should move from essential information toward implementation detail.

- Requirement modeling begins by describing the problem from the end-user's perspective.
- The "essence" of the problem is described without any consideration of how a solution will be implemented.

5.2 Design Modeling Principles

Design models represent characteristics of the software that help practitioners to construct it effectively:

- The architecture,
- The user interface, and
- The component-level detail.

Principle 1: Design should be traceable to the requirements model.

- The analysis mode describes the information domain of the problem, user visible functions and various things about system.
- The design model translates this information into architecture: a set of sub systems that implement major functions, and a set of component level designs that are the realization of analysis classes.

Principle 2: Always consider the architecture of the system to be built.

- Software architecture is a skeleton of the system to be built. It affects interfaces, data structures, program control flow and behavior, the manner in which testing can be conducted, the maintainability of the resultant system and much more.
- For all these reasons, design should start with architectural considerations.

Principle 3: Design of data is as important as design of processing functions.

- Data design is an essential element of architectural design.
- The manner in which data objects are realized within the design cannot be left to chance.
- A well-structured data design helps to simplify program flow, makes the design and implementation of software components easier, and makes overall processing more efficient.

Principle 4: Interfaces (both internal and external) must be designed with care.

- The manner in which data flows between the components of a system has much to do with processing efficiency, error propagation, and design simplicity.
- A well designed interface makes integration easier and assists the tester in validating component functions.

Principle 5: User interface design should be tuned to the needs of the end user.

- The user interface is the visible manifestation of the software.
- No matter how sophisticated its internal functions, no matter how comprehensive its data structures, no matter how well designed its architecture, a poor interface design often leads to the perception that the software is bad.

Principle 6: Component-level design should be functionally independent.

- The functionality that is delivered by a component should be cohesive – that is it should focus on one and only one function or sub –function

Principle 7: Components should be loosely coupled to one another and to the external environment.

- Coupling is achieved in many ways – via a component interface, by messaging, through global data.
- As the level of coupling increases, the likelihood or error propagation also increases and the overall maintainability of the software decreases.

Principle 8: Design representations (models) should be easily understandable.

- The purpose of design is to communicate information to practitioners who will generate code, to those who will test the software, and to others who may maintain the software in the future.
- If the design is difficult to understand, it will not serve as an effective communication medium.

Principle 9: The design should be developed iteratively. With each iteration, the designer should strive for greater simplicity.

- Like almost all creative activities, design occurs iteratively.
- The first iterations work to refine the design and correct errors, but later iterations should strive to make the design as simple as is possible.

6. Construction Principles

The construction activity encompasses a set of coding and testing tasks that lead to operational software that is ready for delivery to the customer or end user.

6.1 Coding Principles

In modern software engineering work, coding may be

(1) The direct creation of programming language source code (e.g., Java),

(2) The automatic generation of source code using an intermediate design-like representation of the component to be built (e.g. Microsoft front end where code is automatically generated), or

(3) The automatic generation of executable code using a "fourth-generation programming language" (e.g., Visual C++).

Preparation principles: Before you write one line of code, be sure you

- Understand of the problem you're trying to solve.
- Understand basic design principles and concepts.
- Pick a programming language that meets the needs of the software to be built and the environment in which it will operate.
- Select a programming environment that provides tools that will make your work easier. (E.g. TC, JRE etc.).
- Create a set of unit tests that will be applied once the component you code is completed.

Programming principles: As you begin writing code, be sure you

- Constrain your algorithms by following structured programming practice.
- Consider the use of pair programming.
- Select data structures that will meet the needs of the design.
- Understand the software architecture and create interfaces that are consistent with it.
- Keep conditional logic as simple as possible.
- Create nested loops in a way that makes them easily testable.
- Select meaningful variable names and follow other local coding standards
- Write code that is self-documenting. (e.g. Comments)
- Create a visual layout (e.g., indentation and blank lines) that aids understanding.

Validation Principles: After you've completed your first coding pass, be sure you

- Conduct a code walkthrough when appropriate.
- Perform unit tests and correct errors you've uncovered.
- Refactor the code.

6.2 Testing Principles

Testing objectives:

- Testing is a process of executing a program with the intention of finding an error.
- A good test case is one that has a high probability of finding an undiscovered error.
- A successful test is one that uncovers an undiscovered error.

Levels of Testing:

The initial focus of testing is at the component level, often called *unit testing*

The other levels of testing include:

1. Integration Testing (conducted as the system is constructed).
2. Validation Testing (assesses whether requirements have been met for the complete system or software increment).
3. Acceptance Testing (conducted by the customer in an effort to check all required features and functions)

Principle 1: All tests should be traceable to customer requirements.

- The objective of software testing is to uncover errors.
- It follows that the most severe defects (from the customer's point of view) are those that cause the program to fail to meet its requirements.

Principle 2: Tests should be planned long before testing begins.

- Test planning can begin as soon as the requirements model is complete.
- Detailed definition of test cases can begin as soon as the design model has been made.
- Therefore, all tests can be planned and designed before any code has been generated.

Principle 3: The Pareto principle applies to software testing.

- In this context the Pareto principle implies that 80 percent of all errors uncovered during testing will likely to be coming from 20 percent of all program components.
- The problem, of course, is to isolate these suspect components and to thoroughly test them.

Principle 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 errors in integrated clusters of components and ultimately in the entire system.

Principle 5: Exhaustive testing is not possible.

- The number of path permutations (possible combinations) for even a moderately sized program is exceptionally large.
- For this reason, it is impossible to execute every combination of paths during testing.
- However, it is possible, to adequately cover program logic and to ensure that all conditions in the component-level design have been tested.

Deployment Principles

Concept of Delivery Cycle, Support Cycle & feedback Cycle:

- The deployment activity encompasses three actions: delivery, support, and feedback.
- Because modern software process models are evolutionary or incremental in nature, deployment happens not once, but a number of times as software move toward completion.
- Each delivery cycle provides the customer and end users with an operational software increment that provides usable functions and features.
- Each support cycle provides documentation and human assistance for all functions and features introduced during all deployment cycles to date.
- Each feedback cycle provides the software team with important guidance that result in modifications to the functions, features, and approach taken for the next increment.

Principles:

Principle 1: Customer expectations for the software must be managed.

- Too often, the customer expects more than the team has promised to deliver, and disappointment occurs immediately. This results in feedback that is not productive.
- Software engineer must be careful about sending the customer conflicting messages (e.g., promising more than you can reasonably deliver in the time frame provided or delivering more than you promise for one software increment and then less than promised for the next).

Principle 2: A complete delivery package should be assembled and tested.

- A CD-ROM or other media (including Web-based downloads) containing all executable software, support data files, support documents, and other relevant information should be assembled and thoroughly beta-tested with actual users.
- All installation scripts and other operational features should be thoroughly exercised.

Principle 3: A support regime must be established before the software is delivered.

- When a problem or question arises at end user's side, he/she expects responsiveness and accurate information.
- If support is ad hoc or nonexistent, the customer will become dissatisfied immediately. Support should be planned, support materials should be prepared, and appropriate recordkeeping mechanisms should be established so that the software team can conduct an assessment of the kinds of support requested.

Principle 4: Appropriate instructional materials must be provided to end users.

- The software team delivers more than the software itself.
- Appropriate training aids (if required) should be developed; troubleshooting guidelines should be provided

Principle 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." Customer gets disappointed by this.
- Hence it is necessary to fix the bug before the product is delivered to the customer.

7. Requirements Engineering

From software process perspective, requirements engineering is a major software engineering action that begins during the communication activity and continues into the modeling activity. It must be adapted to the needs of the process, the project, the product, and the people doing the work.

Need of Requirement Engineering:-

Requirements engineering tools assist in requirements gathering, requirements modeling, requirements management, and requirements validation.

Subtasks included in Requirement Engineering:-

- Inception
- Elicitation
- Elaboration
- Negotiation
- Specification
- Validation
- Requirements management

7.1 Requirement Engineering Tasks

Requirement Engineering encompasses seven distinct tasks: inception, elicitation, elaboration, negotiation, specification, validation, and management.

1. Inception

- Most projects begin when a business need is identified or a potential new market or service is discovered.
- Stakeholders from the business community (e.g., business managers, marketing people, and product managers) define a business case for the idea, try to identify the breadth and depth of the market, do a rough feasibility analysis, and identify a working description of the project's scope.
- At project inception, you establish a basic understanding of the problem, the people who want a solution, the nature of the solution that is desired, and the effectiveness of preliminary communication and collaboration between the customer and the software team.

2. Elicitation

- Elicitation means to define what is required.
- Requirement engineer asks the customer, user and others:

1) what the objectives for the system or product are
2) what is to be accomplished
3) how the system or product fits into the needs of the business,
4) how the system or product is to be used on a day-to-day basis

Requirement elicitation is difficult because numbers of problems are encountered:

- **Problems of scope:**

The boundary of the system is ill-defined or the customers/users specify unnecessary technical detail that may confuse, rather than clarify, overall system objectives.

- **Problems of understanding:**

The customers/users:

- are not completely sure of what is needed
- have a poor understanding of the capabilities and limitations of their computing environment,

- don’t have a full understanding of the problem domain,
- have trouble communicating needs to the system engineer,
- Omit information that is believed to be “obvious,”
- Specify requirements that conflict with the needs of other customers/users, or specify requirements that are ambiguous (unclear) or untestable.

- **Problems of volatility:**

The requirements change over time.

3. Elaboration.

- The information obtained from the customer during inception and elicitation is expanded and refined during elaboration.
- This task focuses on developing a refined requirements model that identifies various aspects of software function, behavior, and information.
- Elaboration is driven by the creation and refinement of user scenarios that describe how the end user (and other actors) will interact with the system.
- Each user scenario is parsed to extract analysis classes—business domain entities that are visible to the end user.
- The attributes of each analysis class are defined, and the services that are required by each class are identified.
- The relationships and collaboration between classes are identified, and a variety of supplementary diagrams are produced.

4. Negotiation

- It isn’t unusual for customers and users to ask for more than that can be achieved, with limited business resources.
- Requirement engineer must reconcile these conflicts through a process of negotiation.
- Customers, users, and other stakeholders are asked to rank requirements and then discuss conflicts in priority.
- Risks associated with each requirement are identified and analyzed.
- Rough estimates of development effort are made and are used to assess the impact of each requirement on project cost and delivery time.
- Using an iterative approach, requirements are eliminated, combined, and/or modified so that each party achieves some measure of satisfaction.

5. Specification.

- A specification can be a written document, a set of graphical models, a formal mathematical model, and a collection of usage scenarios, a prototype, or any combination of these.
- For large systems, a written document, combining natural language descriptions and graphical models may be the best approach.
- Specification is the final work product produced by the requirement engineer.
- It describes the function and performance of a computer based system and the constraints that will be imposed while its development.

6. Validation

- The work products produced as a consequence of requirements engineering are assessed for quality during a validation step.
- Requirements validation examines the specification to ensure that all software requirements have been stated clearly; that inconsistencies, omissions, and errors have been detected and corrected; and that the work products conform to the standards established for the process, the project, and the product.
- The primary requirements validation mechanism is the technical review.
- The review team that validates requirements includes software engineers, customers, users, and other stakeholders who examine the specification looking for errors in content or interpretation, areas where clarification may be required, missing information, inconsistencies (a major problem when large products or systems are engineered), conflicting requirements, or unrealistic (unachievable) requirements.

7. Requirements management

- Requirements management is a set of activities that help the project team identify, control, and track requirements and changes to requirements at any time as the project proceeds.
- Many of these activities are identical to the software configuration management (SCM) techniques.

TYPES OF REQUIREMENTS –

A software requirement can be of 3 types:

- Functional requirements
- Non-functional requirements
- Domain requirements

Functional Requirements: These are the requirements that the end user specifically demands as basic facilities that the system should offer. 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. They are basically the requirements stated by the user which one can see directly in the final product, unlike the non-functional requirements.
For example, in a hospital management system, a doctor should be able to retrieve the information of his patients. Each high-level functional requirement may involve several interactions or dialogues between the system and the outside world. In order to accurately describe the functional requirements, all scenarios must be enumerated.

There are many ways of expressing functional requirements e.g., natural language, a structured or formatted language with no rigorous syntax and formal specification language with proper syntax.

Non-functional requirements: These are basically the quality constraints that the system must satisfy according to the project contract. 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

NFR's are classified into following types:

- Interface constraints
- Performance constraints: response time, security, storage space, etc.
- Operating constraints
- Life cycle constraints: mantainability, portability, etc.
- Economic constraints

The process of specifying non-functional requirements requires the knowledge of the functionality of the system, as well as the knowledge of the context within which the system will operate.

- **Product requirements:** These requirements specify how software product performs. Product requirements comprise the following-

Efficiency requirements: Describe the extent to which the software makes optimal use of resources, the speed with which the system executes, and the memory it consumes for its operation. For example, the system should be able to operate at least three times faster than the existing system.
Reliability requirements: Describe the acceptable failure rate of the software. For example, the software should be able to operate even if a hazard occurs.
Portability requirements: Describe the ease with which the software can be transferred from one platform to another. For example, it should be easy to port the software to a different operating system without the need to redesign the entire software.
Usability requirements: Describe the ease with which users are able to operate the software. For example, the software should be able to provide access to functionality with fewer keystrokes and mouse clicks.

- **Organizational requirements:** These requirements are derived from the policies and procedures of an organization. Organizational requirements comprise the following-

Delivery requirements: Specify when the software and its documentation are to be delivered to the user.
Implementation requirements: Describe requirements such as programming language and design method.
Standards requirements: Describe the process standards to be used during software development. For example, the software should be developed using standards specified by the ISO and IEEE standards.

- **External requirements:** These requirements include all the requirements that affect the software or its development process externally. External requirements comprise the following-

Interoperability requirements: Define the way in which different computer based systems will interact with each other in one or more organizations.
Ethical requirements: Specify the rules and regulations of the software so that they are acceptable to users.
Legislative requirements: Ensure that the software operates within the legal jurisdiction. For example, pirated software should not be sold.

8. Eliciting requirements

Requirements elicitation (also called requirements gathering) combines elements of problem solving, elaboration, negotiation, and specification.

8.1 Collaborative Requirements Gathering

Many different approaches to collaborative requirements gathering have been proposed. Each makes use of a slightly different scenario, but all apply some variation on the following basic guidelines:

• Meetings are conducted and attended by both software engineers and other stakeholders.
• Rules for preparation and participation are established.
• An agenda is suggested that is formal enough to cover all important points but informal enough to encourage the free flow of ideas.
• A "facilitator" (can be a customer, a developer, or an outsider) controls the meeting
• A "definition mechanism" (can be work sheets, flip charts, or wall stickers or an electronic bulletin board, chat room, or virtual forum) is used.

The goal is to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of solution requirements in an atmosphere that is conducive to the accomplishment of the goal.

8.2 Quality Function Deployment

Quality function deployment (QFD) is a quality management technique that translates the needs of the customer into technical requirements for software. QFD emphasizes an understanding of what is valuable to the customer and then deploys these values throughout the engineering process. QFD identifies three types of requirements:

Normal requirements.

The objectives and goals that are stated for a product or system during meetings with the customer. If these requirements are present, the customer is satisfied. Examples of normal requirements might be requested types of graphical displays, specific system functions, and defined levels of performance.

Expected requirements.

These requirements are implicit to the product or system and may be so fundamental that the customer does not explicitly state them. Their absence will be a cause for significant dissatisfaction. Examples of expected requirements are: ease of human/machine interaction, overall operational correctness and reliability, and ease of software installation.

Exciting requirements.

These features go beyond the customer's expectations and prove to be very satisfying when present. For example, software for a new mobile phone comes with standard features, but is coupled with a set of unexpected capabilities (e.g., multitouch screen, visual voice mail) that delight every user of the product.

8.3 Usage Scenarios

As requirements are gathered, an overall vision of system functions and features begins to materialize. However, it is difficult to move into more technical software engineering activities until you understand how these functions and features will be used by different classes of end users. To accomplish this, developers and users can create a set of scenarios that identify a thread of usage for the system to be constructed. The scenarios, often called use cases, provide a description of how the system will be used.

8.4 Elicitation Work Products

The work products produced as a consequence of requirements elicitation will vary depending on the size of the system or product to be built. For most systems, the work products include

• A statement of need and feasibility.
• A bounded statement of scope for the system or product.
• A list of customers, users, and other stakeholders who participated in requirements elicitation.
• A description of the system's technical environment.
• A list of requirements (preferably organized by function) and the domain constraints that apply to each.
• A set of usage scenarios that provide insight into the use of the system or product under different operating conditions.
• Any prototypes developed to better define requirements.

Each of these work products is reviewed by all people who have participated in requirements elicitation.

9. Dveloping Use cases

A use case tells a stylized story about how an end user (playing one of a number of possible roles) interacts with the system under a specific set of circumstances. The story may be narrative text, an outline of tasks or interactions, a template-based description, or a diagrammatic representation. Regardless of its form, a use case depicts the software or system from the end user's point of view.

The first step in writing a use case is to define the set of "actors" that will be involved in the story. Actors are the different people (or devices) that use the system or product within the context of the function and behavior that is to be described. Actors represent the roles that people (or devices) play as the system operates.

Because requirements elicitation is an evolutionary activity, not all actors are identified during the first iteration. It is possible to identify primary actors during the first iteration and secondary actors as more is learned about the system. **Primary actors** interact to achieve required system function and derive the intended benefit from the system. They work directly and frequently with the software. **Secondary actors** support the system so that primary actors can do their work

Once actors have been identified, use cases can be developed. A number of questions12 that should be answered by a use case:

- Who is the primary actor, the secondary actor(s)?
- What are the actor's goals?
- What preconditions should exist before the story begins?
- What main tasks or functions are performed by the actor?
- What exceptions might be considered as the story is described?
- What variations in the actor's interaction are possible?
- What system information will the actor acquire, produce, or change?
- Will the actor have to inform the system about changes in the external environment?
- What information does the actor desire from the system?
- Does the actor wish to be informed about unexpected changes?

9.1 For example, the following template for detailed descriptions of use cases:

Use case: InitiateMonitoring

Primary actor: Homeowner.

Goal in context: To set the system to monitor sensors when the homeowner leaves the house or remains inside.

Preconditions: System has been programmed for a password and to recognize various sensors.

Trigger: The homeowner decides to "set" the system, i.e., to turn on the alarm functions

Scenario:

1. Homeowner: observes control panel
2. Homeowner: enters password
3. Homeowner: selects "stay" or "away"
4. Homeowner: observes read alarm light to indicate that SafeHome has been armed

Exceptions:

1. Control panel is not ready: homeowner checks all sensors to determine which are open; closes them.
2. Password is incorrect (control panel beeps once): homeowner reenters correct password.
3. Password not recognized: monitoring and response subsystem must be contacted to reprogram password.
4. Stay is selected: control panel beeps twice and a stay light is lit; perimeter sensors are activated.
5. Away is selected: control panel beeps three times and an away light is lit; all sensors are activated.

Priority: Essential, must be implemented

When available: First increment

Frequency of use: Many times per day
Channel to actor: Via control panel interface
Secondary actors: Support technician, sensors
Channels to secondary actors:
Support technician: phone line
Sensors: hardwired and radio frequency interfaces
Open issues:
1. Should there be a way to activate the system without the use of a password or with an abbreviated password?
2. Should the control panel display additional text messages?
3. How much time does the homeowner have to enter the password from the time the first key is pressed?
4. Is there a way to deactivate the system before it actually activates?

UML use case diagram for *SafeHome* home security function

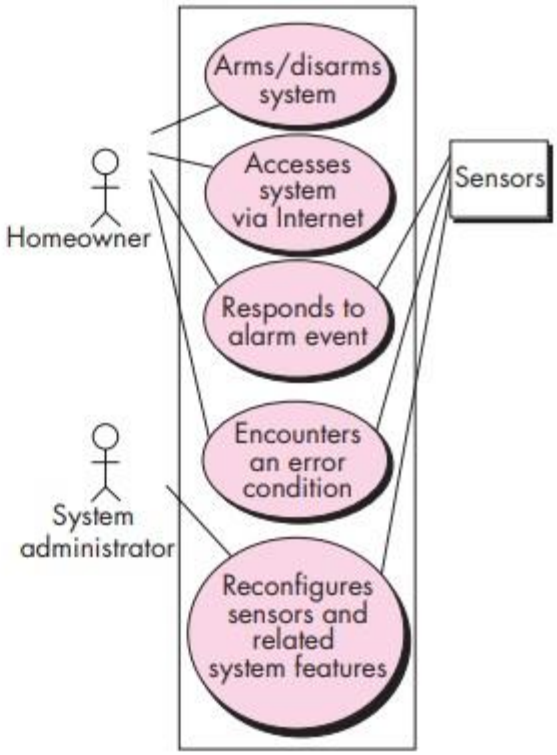

fig-9.1

10. Building a requirements model

The intent of the analysis model is to provide a description of the required informational, functional, and behavioral domains for a computer-based system.

10.1 Elements of the Requirements Model

There are many different ways to look at the requirements for a computer-based system. Some software people argue that it's best to select one mode of representation (e.g., the use case) and apply it to the exclusion of all other modes. Different modes of representation force you to consider requirements from different viewpoints.

Scenario-based elements. The system is described from the user's point of view using a scenario-based approach. For example, basic use cases (Section 9.1) and their corresponding use-case diagrams(use diagram above 9.1) evolve into more elaborate template-based use cases. Scenario-based elements of the requirements model are often the first part of the model that is developed. As such, they serve as input for the creation of other modeling elements.

Figure 10.1 depicts a UML activity diagram17 for eliciting requirements and representing them using

use cases.

UML activity diagrams for eliciting requirements

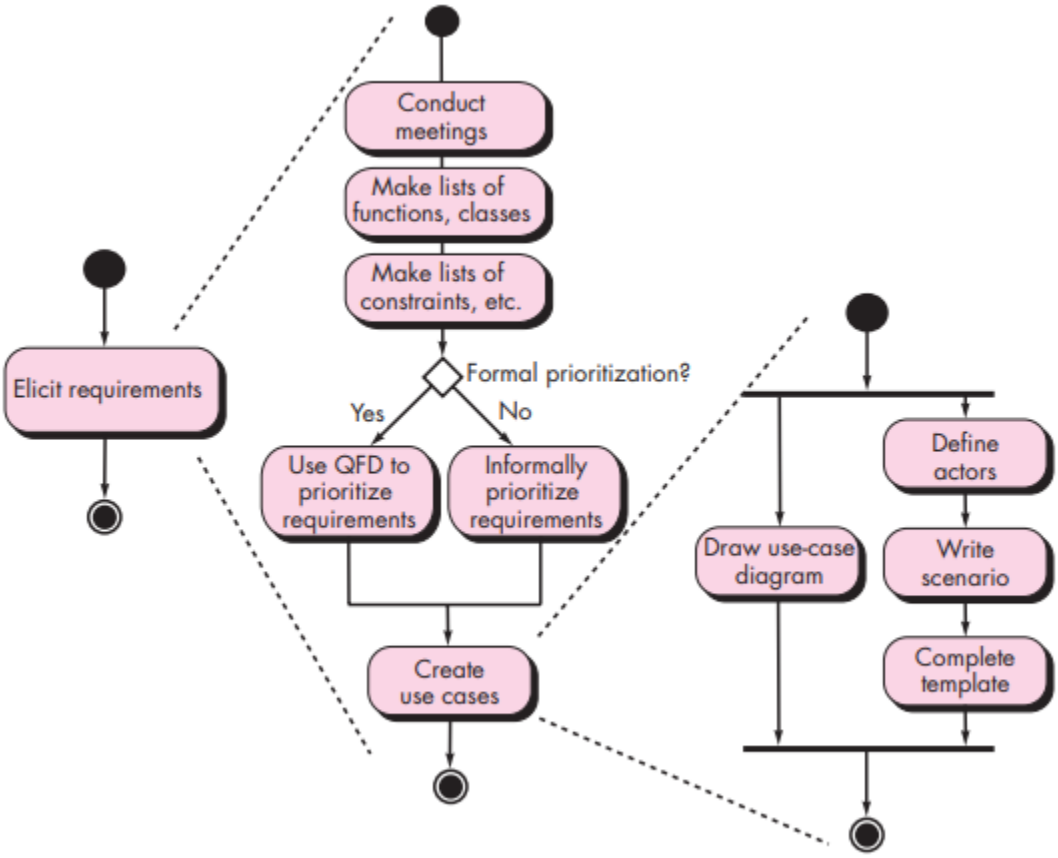

Fig-10.1

Class-based elements. Each usage scenario implies a set of objects that are manipulated as an actor interacts with the system. These objects are categorized into classes—a collection of things that have similar attributes and common behaviors. For example, a UML class diagram can be used to depict a Sensor class for the SafeHome security function (Figure 10.2). Note that the diagram lists the attributes of sensors (e.g., name, type) and the operations (e.g., identify, enable) that can be applied to modify these attributes.

Class diagram for sensor

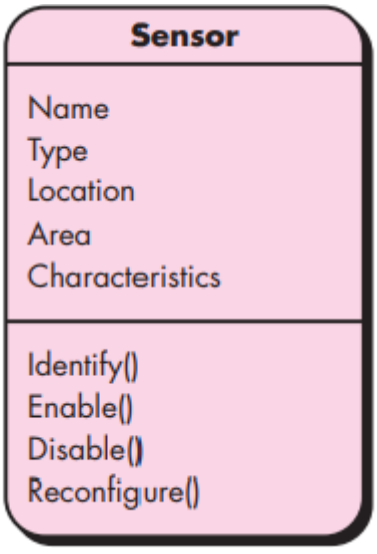

Fig 10.2

Behavioral elements. The state diagram is one method for representing the behavior of a system by depicting its states and the events that cause the system to change state. A state is any externally observable mode of behavior. In addition, the state diagram indicates actions (e.g., process activation) taken as a consequence of a particular event. A simplified UML state diagram is shown in Figure 10.3

UML state diagram notation

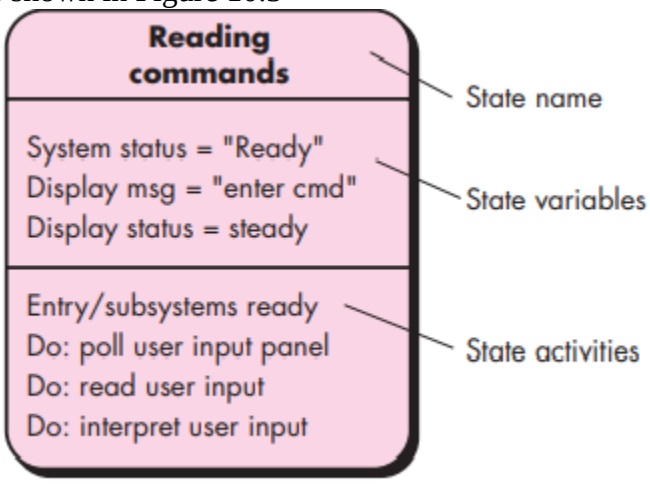

fig 10.3

Flow-oriented elements. Information is transformed as it flows through a computer-based system. The system accepts input in a variety of forms, applies functions to transform it, and produces output in a variety of forms.

10.2 Analysis Patterns

Certain problems reoccur across all projects within a specific application domain. These analysis patterns suggest solutions (e.g., a class, a function, a behavior) within the application domain that can be reused when modeling many applications. Analysis patterns are integrated into the analysis model by reference to the pattern name. They are also stored in a repository so that requirements engineers can use search facilities to find and apply them.

FOR MORE EXMAPLES OF UML DIAGRAMS GO TO APPENDIX 1 OF THE BOOK

11. Negotiating requirements

You may have to enter into a negotiation with one or more stakeholders. In most cases, stakeholders are asked to balance functionality, performance, and other product or system characteristics against cost and time-to-market. The intent of this negotiation is to develop a project plan that meets stakeholder needs while at the same time reflecting the real-world constraints (e.g., time, people, budget) that have been placed on the software team.

The best negotiations strive for a “win-win” result.20 That is, stakeholders win by getting the system or product that satisfies the majority of their needs and you (as a member of the software team) win by working to realistic and achievable budgets and deadlines.

Boehm [Boe98] defines a set of negotiation activities at the beginning of each software process iteration.

Rather than a single customer communication activity, the following activities are defined:

1. Identification of the system or subsystem's key stakeholders.
2. Determination of the stakeholders' "win conditions."
3. Negotiation of the stakeholders' win conditions to reconcile them into a set of win-win conditions for all concerned (including the software team).

12. Validating requirements

As each element of the requirements model is created, it is examined for inconsistency, omissions, and ambiguity.

A review of the requirements model addresses the following questions:

- Is each requirement consistent with the overall objectives for the system/product?
- Have all requirements been specified at the proper level of abstraction? That is, do some requirements provide a level of technical detail that is inappropriate at this stage?
- Is the requirement really necessary or does it represent an add-on feature that may not be essential to the objective of the system?
- Is each requirement bounded and unambiguous?
- Does each requirement have attribution? That is, is a source (generally, a specific individual) noted for each requirement?
- Do any requirements conflict with other requirements?
- Is each requirement achievable in the technical environment that will house the system or product?
- Is each requirement testable, once implemented?
- Does the requirements model properly reflect the information, function, and behavior of the system to be built?
- Has the requirements model been "partitioned" in a way that exposes progressively more detailed information about the system?
- Have requirements patterns been used to simplify the requirements model? Have all patterns been properly validated? Are all patterns consistent with customer requirements?

13. Software Requirements Specification

1. Concept:

- A software requirements specification (SRS) is a document that is created when a detailed description of all aspects of the software to be built that must be specified before the project is to commence.
- It is a primary document for development of software.
- It is written by Business Analysts who interact with client and gather the requirements to build the software.

2. Need/Importance of SRS:

- **It establishes the basis for agreement between the customers and the suppliers on what the software product is to do.**

The complete description of the functions performed by the software specified in the SRS will assist the users to determine if the software meets their needs.

- **Reduces the development effort.**

The preparation of the SRS forces the concerned groups to consider all of the requirements before design begins and reduces later redesign, recoding, and retesting.

- **Provide a basis for estimating cost and schedules.**

The description of the product to be developed given in SRS is a realistic basis for estimating project costs and prices.

- **Provide a baseline for verification and validation.**

Organizations can develop their validation and verification plans much more productively from a good SRS.

- **Facilitate Transfer.**

The SRS makes it easier to transfer the software product to new users or new machines.

- **Serve as basis for enhancement.**

Because the SRS discusses the product but not just the project that is developed, the SRS serves as a basis for later enhancement of the finished product.

3. Characteristics of SRS

The desirable characteristics of an SRS are following:

- Correct: An SRS is correct if every requirement included in the SRS represents something required in the final system.
- Complete: An SRS is complete if everything software is supposed to do and the responses of the software to all classes of input data are specified in the SRS.
- Unambiguous: An SRS is unambiguous if and only if every requirement stated has one and only one interpretation.
- Verifiable: An SRS is verifiable if and only if every specified requirement is verifiable i.e. there exists a procedure to check that final software meets the requirement.
- Consistent: An SRS is consistent if there is no requirement that conflicts with another.
- Traceable: An SRS is traceable if each requirement in it must be uniquely identified to a source.
- Modifiable: An SRS is modifiable if its structure and style are such that any necessary change can be made easily while preserving completeness and consistency.
- Ranked: An SRS is ranked for importance and/or stability if for each requirement the importance and the stability of the requirements are indicated.

SRS links -

https://image.slidesharecdn.com/softwareengineeringbypankajjalote-120826045222-phpapp02/95/software-engineering-by-pankaj-jalote-50-728.jpg?cb=1346070951

https://image.slidesharecdn.com/softwareengineeringbypankajjalote-120826045222-phpapp02/95/software-engineering-by-pankaj-jalote-53-728.jpg?cb=1346070951

https://image.slidesharecdn.com/softwareengineeringbypankajjalote-120826045222-phpapp02/95/software-engineering-by-pankaj-jalote-54-728.jpg?cb=1346070951

https://image.slidesharecdn.com/softwareengineeringbypankajjalote-120826045222-phpapp02/95/software-engineering-by-pankaj-jalote-55-728.jpg?cb=1346070951

https://image.slidesharecdn.com/softwareengineeringbypankajjalote-120826045222-phpapp02/95/software-engineering-by-pankaj-jalote-56-728.jpg?cb=1346070951

https://image.slidesharecdn.com/softwareengineeringbypankajjalote-120826045222-phpapp02/95/software-engineering-by-pankaj-jalote-57-728.jpg?cb=1346070951
https://image.slidesharecdn.com/softwareengineeringbypankajjalote-120826045222-phpapp02/95/software-engineering-by-pankaj-jalote-58-728.jpg?cb=1346070951

https://image.slidesharecdn.com/softwareengineeringbypankajjalote-120826045222-phpapp02/95/software-engineering-by-pankaj-jalote-59-728.jpg?cb=1346070951

https://image.slidesharecdn.com/softwareengineeringbypankajjalote-120826045222-phpapp02/95/software-engineering-by-pankaj-jalote-60-728.jpg?cb=1346070951