



# Unit – 2

## Software Requirements Analysis and Project Management



# Software Requirements

- For large software systems, requirements analysis is the most difficult and error-prone activity.
- In situations where the software is to automate a currently manual process, many of the needs can be understood by observing the current practice.
- No methods exist for systems for which manual processes do not exist.
- For such systems, the requirements problem is complicated by the fact that the needs and requirements of the system have to be visualized and created.



# Software Requirements

- The requirement phase translates the ideas in the minds of the clients (the input), into a formal document.
- The output of the phase is a set of precisely specified requirements.
- The software requirements activity cannot be fully automated.
- Any method for identifying requirements can be at best a set of guidelines.
- This is because, the process of specifying requirements cannot be totally formal.
- Any formal translation process producing a formal output must have a precise and unambiguous input.



# Software Requirements

- IEEE defines a requirements as “(1) A condition or capability needed by a user to solve a problem or achieve an objective; (2) A condition or a capability that must be met or possessed by a system, to satisfy a contract, standard, specification, or other formally imposed document.”
- The goal of the requirements activity is to produce the Software Requirements Specification (SRS), that describes what the proposed software should do without describing how the software will do it.



# Needs for SRS

- There are three major parties interested in a new system: the client, the developer, and the users.
- The requirements for the system that will satisfy the needs of the clients and the concerns of the users have to be communicated to the developer.
- The problem is that the client usually does not understand software or the software development process, and the developer often does not understand the client's problem and application area.
- This causes a communication gap between the parties involved in the development project.

# Needs for SRS

- The basic purpose of software requirements specification is to bridge this communication gap.
- *An SRS establishes the basis for agreement between the client and the supplier, what the software product will do.*
  - Through SRS, the client clearly describes what it expects from the supplier, and the developer clearly understands what capabilities to build in the software.
- *An SRS provides a reference for validation of the final product.*
  - Without a proper SRS, there is no way a client can determine if the software being delivered is what was ordered, and there is no way the developer can convince the client that all the requirements have been fulfilled.

# Needs for SRS

- *A high-quality SRS is a prerequisite to high-quality software.*
  - If the SRS document specifies a wrong system, then even a correct implementation of the SRS will lead to a system that will not satisfy the client.
- A high quality SRS reduces the development cost.
  - With a high-quality SRS, requirement changes that come about due to improperly analyzed requirements should be reduced considerably.



# Requirement Process

- The requirement process is the sequence of activities that need to be performed in the requirements phase.
- The requirements process consists of three basic tasks :
  - problem or requirement analysis
  - requirement specification
  - requirements validation



# Requirement Process

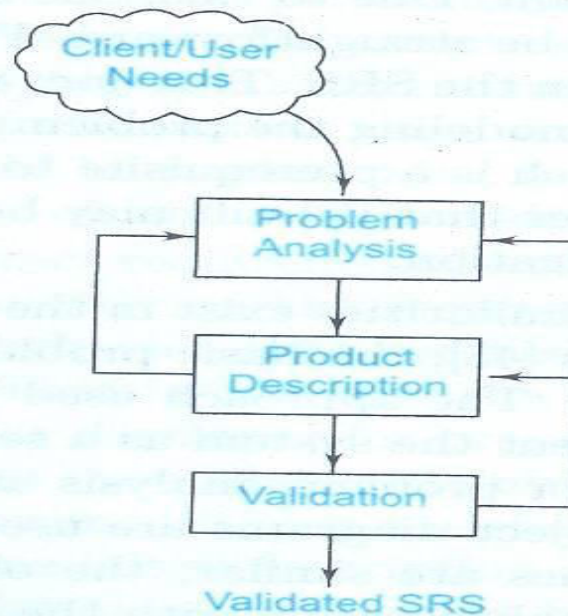


Figure 3.1: The requirement process.



# Requirement Process

- Problem analysis starts with high-level problem statement.
- The basic purpose of this activity is to obtain a thorough understanding of what the software needs to provide.
- The understanding of system forms the basis of requirements specification.
- In requirements specification, the focus is on clearly specifying the requirements in a document.
- Issues such as representation, specification languages, and tools are addressed during this activity.



# Requirement Process

- Requirements validation focuses on ensuring that what has been specified in the SRS are indeed all the requirements of the software and making sure that the SRS is of good quality.
- The requirements process terminates with the production of the validated SRS.
- There is also the issue of the level of detail that the requirement process should aim to uncover and specify.



# Requirements Specification

- The final output is the software requirements specification document (SRS).
- For smaller problems, this activity might come after the entire analysis is complete.
- However, it is more likely that problem analysis and specification are done concurrently.
- An analyst typically will analyze some parts of the problem and then write the requirements for that part.
- In practice, problem analysis and requirements specification activities overlap.



# Characteristics of SRS

To satisfy the basic goals properly, an SRS should have following characteristics. A good SRS is :

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

# Characteristics of SRS

- 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 the software is supposed to do and the responses of the software to all classes of input data are specified in the SRS.
- Correctness ensures that which is specified is done correctly, completeness ensures that everything is indeed specified.
- Unambiguous :- An SRS is unambiguous if and only if every requirement stated has one and only one interpretation.
- Requirements are often written in natural language, so there are chances of ambiguity.

# Characteristics of SRS

- Verifiable :- An SRS is verifiable if and only if every stated requirement is verifiable. A requirement is verifiable if there exists some cost-effective process that can check whether the final software meets that requirement.
- Consistent :- An SRS is consistent if there is no requirement that conflicts with another. For example, suppose a requirement states that an event 'e' is to occur before another event 'f'. But then another set of requirements states that event 'f' should occur before event 'e'.
- Some requirements are “core” requirements which are not likely to change as time passes, while others are more dependent on time.

# Characteristics of SRS

- Ranked for importance and/or stability :- An SRS is ranked for importance and/or stability if for each requirement the importance and the stability of the requirement are indicated.
- Stability of a requirement reflects the chances of it changing in future.
- 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.
- Traceable :- An SRS is traceable if the origin of each of its requirements is clear and if it facilitates the referencing of each requirement in future development.





# Characteristics of SRS

- Forward traceability means that each requirement should be traceable to some design and code elements.
- Backward traceability means that it is possible to trace design and code elements to the requirements they support.



# Components of an SRS

The SRS should have following components.  
They are the system properties that an SRS should specify.

- Functionality
- Performance
- Design constraints imposed on implementation
- External interfaces



# Functional Requirements

- Functional requirement specify which outputs should be produced from the given inputs.
- They describe the relationship between the input and output of the system.
- For each functional requirement, a detailed description of all the data inputs and their source, the units of measure, and the range of valid inputs must be specified.
- All the operations to be performed on the input data to obtain the output should be specified.
- For example, if there is a formula for computing the output, it should be specified, but not whole algorithm.



# Functional Requirements

- The system behavior in abnormal situations, like invalid input or error during computation should also be specified.
- The functional requirement must clearly state what system should do if such situations occur.
- Furthermore, behavior for situations where the input is valid but the normal operation cannot be performed. For example, airline ticket booking.
- In short, the system behavior for all foreseen inputs and all foreseen system states should be specified.



# Performance Requirements

- This part of SRS specifies the performance constraints on the software system.
- All the requirements relating to the performance characteristics of the system must be clearly specified.
- There are two types of performance requirements : Static and Dynamic
- Static requirements are those that do not impose constraint on the execution characteristics of the system.
- These include requirements like the number of simultaneous users to be supported, the number of files that the system has to process and their sizes.
- These are also called capacity requirements of the system.

# Performance Requirements

- Dynamic requirements specify constraints on the execution behavior of the system.
- These include response time and throughput constraints on the system.
- Response time is the expected time for the completion of an operation under specified circumstances.
- Throughput is the expected number of operations that can be performed in a unit time.
  - For e.g., an SRS may specify the number of transactions that must be processed per unit time or what the response time for a particular command should be.
  - These requirements should be stated in measurable terms. Instead of using statements like “response time should be good”, statement like “response time of command x should be less than one second 90% of time” should be used.

# Design Constraints

- There are factors in the client's environment that may restrict the choice of designer.
- Such factors include standards that must be followed, resource limitations, operating environment, reliability and security requirements, etc.
- An SRS should identify and specify all such constraints.
- **Standards Compliance** : This specifies the requirements for the standards the system must follow. The standards may include the report format and accounting procedures.
- **Hardware Limitations** : The software may have to operate on some existing or predetermined hardware, thus imposing restrictions on the design.



# Design Constraints

- **Reliability and Fault Tolerance** : Fault tolerance requirements can place a major constraint on how the system is to be designed. They make the system more complex and expensive.
- **Security** : Security requirements are particularly significant in defense systems and many database systems. They place restrictions on the use of certain commands, control access to data, etc.





# External Interface Requirements

- All the interactions of the software with people, hardware, and other software should be clearly specified.
- A preliminary user manual should be created with all user commands, screen formats, an explanation of how the system will appear to the user, and feedback and error messages.
- Like other specifications these requirements should be precise and verifiable.
- A statement like “the system should be user friendly” should be avoided and statements like “password should not longer than six characters.” should be used.




# Specification Language

- Requirements specification necessitates the use of some specification language.
- The language should support the desired qualities of the SRS. In addition, the language should be easy to learn and use.
- To avoid ambiguity, it is best to use some formal language. But for ease of understanding a natural language might be preferable.
- The major advantage of using a natural language is that both client and supplier understand the language. But it is imprecise and ambiguous.
- To reduce the drawbacks of natural languages, most often natural language is used in a structured fashion.
- In structured English, requirements are broken into sections and paragraphs. Each paragraph is then broken into subparagraphs.



# Structure of Requirements Document

- All the requirements for the system have to be included in a document that is clear and concise.
- For this, it is necessary to organize the requirements document as sections and subsections.
- Many methods and standards have been proposed for organizing an SRS.
- One of the main ideas of standardizing the structure of the document is that with an available standard, each SRS will fit a certain pattern, which will make it easier for others to understand.

- 
- Another role these standards play is that they help to ensure that the analyst does not forget some major property.
  - The IEEE standards recognize the fact that different projects may require their requirements to be organized differently, that is, there is no method that is suitable for all projects.
  - The general structure of an SRS is given in following figure:

# Structure of Requirements Document

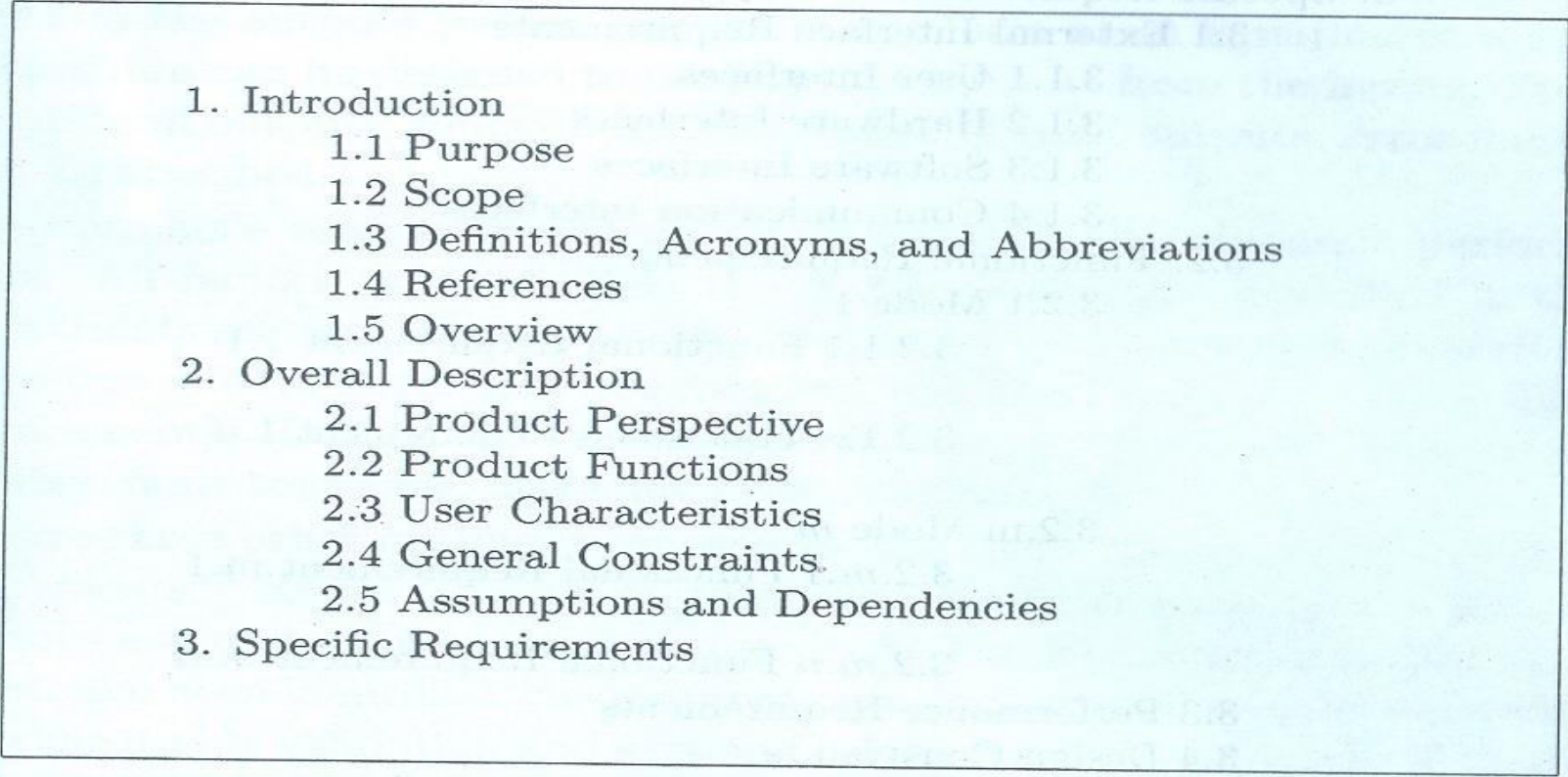
- 
1. Introduction
    - 1.1 Purpose
    - 1.2 Scope
    - 1.3 Definitions, Acronyms, and Abbreviations
    - 1.4 References
    - 1.5 Overview
  2. Overall Description
    - 2.1 Product Perspective
    - 2.2 Product Functions
    - 2.3 User Characteristics
    - 2.4 General Constraints
    - 2.5 Assumptions and Dependencies
  3. Specific Requirements

Figure 3.13: General structure of an SRS.



# Structure of Requirements Document

- The introduction section contains purpose, scope, definitions, references, overview etc.
- Section 2 describes the general factors that affect the product and its requirements.
- Specific requirements are not mentioned but a general overview is presented to make the understanding of the specific requirements easier.
- Product perspective is the relationship of the product to other products; defining if the product is independent or is a part of a larger product, and what the principal interfaces of the product are.
- A general abstract description of the functions to be performed by the product is given in this section.



# Structure of Requirements Document

- In this section, typical characteristics of the end user and general constraints are also specified.
- The specific requirements section describes all the details that the software developer needs to know for designing and developing the system.
- This is the largest and most important part of the document.
- Among many methods, one method to organize the specific requirements is to first specify the external interfaces, followed by functional requirements, performance requirements, design constraints, and system attributes. As shown in the figure:

# Structure of Requirements Document

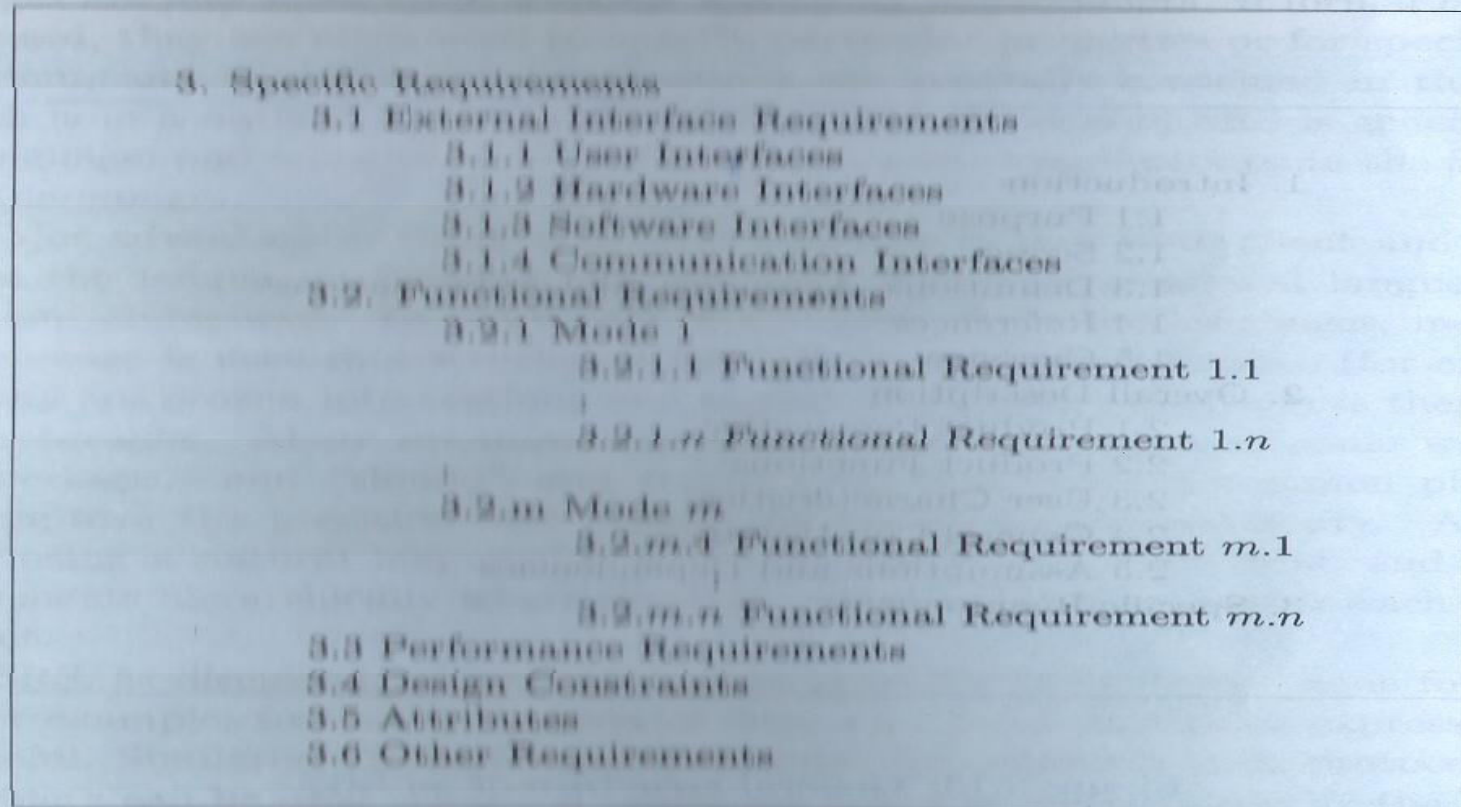


Figure 3.14: One organization for specific requirements.





# Structure of Requirements Document

- The external interface requirements section specifies all the interfaces of the software to people, other software, hardware, and other systems.
- User interfaces specify each human interface the system plans to have, including screen formats, contents of menus, and command structure.
- In hardware interfaces, the logical characteristics of each interface between the software and hardware on which the software can run are specified.



# Structure of Requirements Document

- Any assumptions the software is making about the hardware are listed here.
- In software interfaces, all other software that is needed for this software to run is specified, along with the interfaces.
- Communication interfaces need to be specified if the software communicates with other entities in other machines.



# Functional Specification with Use Cases

- Functional requirements often form the core of a requirements document.
- The traditional approach for specifying functionality is to specify each function that the system should provide.
- Use Cases specify the functionality of a system by specifying the behavior of the system, captured as interactions of the users with the system.

# Basics of Use Case

- **Actor** :- in use case terminology, an actor is a person or a system which uses the system being built for achieving some goal. Actors need not be people only, it may be other system also.
- **Primary Actor** :- A primary actor is the main actor that initiates a use case (UC) for achieving a goal and whose goal satisfaction is the main objective of the use case. The goal of the primary actor is the driving force behind a use case, the use case must also fulfill any goals that other stakeholders might also have for this use case. For example, a use case “withdraw money from ATM”, primary actor is customer, the bank is also a stakeholder.

# Basics of Use Case

- **Scenario** :- For describing interaction, use cases use scenarios. A scenario describes a set of actions that are performed to achieve a goal under some specified conditions.
- The set of actions is generally specified as a sequence, though in actual execution the actions specified may be executed in parallel or in some different order.
- Each step in a scenario is a logically complete action performed either by the actor or the system.
- Generally, a step is some action by the actor (enter information), some logical step that the system performs to progress towards achieving its goals (validate information), or an internal state change by the system to satisfy some goals (update the record).

# Basics of Use Case

- **Main success scenario** :- A use case always has a main success scenario, which describes the interaction if nothing fails and all steps in the scenario succeed.
- **Extension scenario** :- use case has extension scenarios which describe the system behavior if some of the steps in the main scenario do not complete successfully. They are also called exception scenarios.
- A use case is a collection of all the success and extension scenarios related to the goal.
- To achieve the desired goal, a system can divide it into sub-goals.



# Basics of Use Case

- Use cases are naturally textual descriptions, and represent the behavioral requirements of the system.
- This behavior specification can capture most of the functional requirements of the system. Therefore, use cases do not form the complete SRS, but can form a part of it.
- The complete SRS, will need to capture other requirements like performance and design constraints.
- Though the detailed use cases are textual, diagrams can be used to supplement the textual description.



# Example of Use Case

- Let us consider a small on-line auction system is to be built, in which different persons can sell and buy goods.
- The *precondition* of a use case specifies what the system will ensure before allowing the use case to be initiated.



# Example of Use Case

- **Use Case 1: Put an item up for auction**

**Primary Actor:** Seller

**Precondition:** Seller has logged in

**Main Success Scenario:**

1. Seller posts an item (its category, description, picture, etc.) for auction
2. System shows past prices of similar items to seller
3. Seller specifies the starting bid price and a date when auction will close
4. System accepts the item and posts it

**Exception Scenarios:**

- 2 a) There are no past items of this category
  - \* System tells the seller this situation

- **Use Case 2: Make a bid**

**Primary Actor:** Buyer

**Precondition:** The buyer has logged in

**Main Success Scenario:**

1. Buyer searches or browses and selects some item
2. System shows the rating of the seller, the starting bid, the current bids, and the highest bid; asks buyer to make a bid
3. Buyer specifies a bid price, maximum bid price, and an increment
4. System accepts the bid; Blocks funds in bidders account
5. System updates the bid price of other bidders where needed, and updates the records for the item

**Exception Scenarios:**

- 3 a) The bid price is lower than the current highest
  - \* System informs the bidder and asks to rebid
- 4 a) The bidder does not have enough funds in his account
  - \* System cancels the bid, asks the user to get more funds

Figure 3.15: Main use cases in an auction system.

# Example of Use Case

- **Use Case 3: Complete auction of an item**

*Primary Actor:* Auction System

*Precondition:* The last date for bidding has been reached

*Main Success Scenario:*

1. Select highest bidder; send email to selected bidder and seller informing final bid price; send email to other bidders also.
2. Debit bidder's account and credit seller's
3. Transfer from seller's acct. commission amt. to organization's acct.
4. Remove item from the site; update records

*Exception Scenarios:* None

Figure 3.15: Main use cases in an auction system (contd.)



# Example of Use Case

- If the system being built has many subsystems, sometimes system use cases may actually be capturing the behavior of some subsystem.
- The enterprise level UCs provide the context in which the systems operate. Hence, sometimes it may be useful to describe some of the key business processes as summary level use cases to provide the context for the system being designed and built.

# Example of Use Case

- **Use Case 0: Auction an item**

**Primary Actor:** Auction system

**Scope :** Auction conducting organization

**Precondition:** None

**Main Success Scenario:**

1. Seller performs *Put an item for auction*
2. Various bidders perform *make a bid*
3. On final date perform *Complete the auction of the item*
4. Get feedback from seller; get feedback from buyer; update records

Figure 3.16: A summary level use case,





# Developing Use Cases

- UCs can be evolved in a stepwise refinement manner with each step adding more details.
- This approach allows UCs to be presented at different levels of abstraction. Though any number of levels of abstractions are possible, four natural levels emerge:

- **Actors and goals.** The actor-goal list enumerates the use cases and specifies the actors for each goal. (The name of the use case is generally the goal.) This table

may be extended by giving a brief description of each of the use cases. At this level, the use cases specify the scope of the system and give an overall view of what it does. Completeness of functionality can be assessed fairly well by reviewing these.

- **Main success scenarios.** For each of the use cases, the main success scenarios are provided at this level. With the main scenarios, the system behavior for each use case is specified. This description can be reviewed to ensure that interests of all the stakeholders are met and that the use case is delivering the desired behavior.
- **Failure conditions.** Once the success scenario is listed, all the possible failure conditions can be identified. At this level, for each step in the main success scenario, the different ways in which a step can fail form the failure conditions. Before deciding what should be done in these failure conditions (which is done at the next level), it is better to enumerate the failure conditions and reviewed for completeness.
- **Failure handling.** This is perhaps the most tricky and difficult part of writing a use case. Often the focus is so much on the main functionality that people do not pay attention to how failures should be handled. Determining what should be the behavior under different failure conditions will often identify new business rules or new actors.



# Developing Use Cases

- The four levels can also guide the analysis activity.
- For writing use cases, general technical writing rules apply.
- Use simple grammar, clearly specify who is performing the step, and keep the overall scenario as simple as possible.
- Also when writing steps, for simplicity, it is better to combine some steps into one logical step, if it makes sense. (“user enters personal information”)



# Validation

- It is important that the requirements specification contains no errors and specifies the client's requirements correctly.
- The basic objective of the requirements validation activity is to ensure that the SRS reflects the actual requirement accurately and clearly.
- A related objective is to check that the SRS document is itself of "good quality".
- There are different types of errors that typically occur in SRS, like omission, inconsistency, incorrect fact and ambiguity.



# Common Errors in SRS

- Omission:- omission is a common error in requirements. In this type of error, some user requirement is simply not included in the SRS. Omission directly affects the external completeness of the SRS.
- Inconsistency:- inconsistency can be due to contradictions within the requirements themselves.
- Incorrect fact:- errors of this type occur when some fact recorded in the SRS is not correct.
- Ambiguity:- errors of this type occurs when there are some requirements that have multiple meanings.



# Validation

- Besides improving the quality of the SRS itself, the validation should focus on uncovering these type of errors.
- Inspections are eminently suitable for requirements validation. Consequently, inspections of the SRS, frequently called requirements review, are the most common method of validation.
- The requirements review team generally consists of client as well as user representatives.
- Requirements review is a review by a group of people to find errors and point out other matters of concern in the requirements specifications of a system.



# Validation

- The review group should include the author of the requirements document, someone who understands the needs of the client, a person of the design team, and the person responsible for maintaining the requirements document.
- The review process is also used to consider factors affecting quality, such as testability and readability.
- Checklists are frequently used in reviews.



# Sample Checklist for Requirements Review

- Are all hardware resources defined?
- Have the response times of functions been specified?
- Have all the functions required by the client been specified?
- Is each requirement testable?
- Is the initial state of the system defined?
- Are possible future modifications specified?



# Software Project Management

- For a successful project, both good project management and good engineering are essential.
- Project management activities can be viewed as having three major phases:
  - Project planning
  - Project monitoring & Control
  - Project termination
- During planning all the activities that management needs to perform are planned
- During project control the plan is executed and updated.



# Software Project Management

- Without a proper plan, no real monitoring or controlling of the project is possible.
- The basic goals of planning are:
  - To look into the future
  - Identify the activities that need to be done to complete the project successfully
  - Plan the scheduling and resources
- The inputs to the planning activity are:
  - the requirements specification
  - the architecture description



# Components of Project Management

- The major issues project planning addresses are:
  - ☐ Process planning
  - ☐ Effort estimation
  - ☐ Schedule and Resource Estimation
  - ☐ Quality plans
  - ☐ Configuration management plans
  - ☐ Risk management
  - ☐ Project monitoring plans

# Process Planning

- During planning, a key activity is ***to plan and specify the process*** that should be used in the project.
- This means specifying;
  - the various stages in the process
  - the entry criteria for each stage
  - the exit criteria
  - the verification activities that will be done at the end of the stage.
- The process planning activity mostly entails selecting a standard process and tailoring it for the project.
- The common tailoring actions are modify a step, omit a step, add a step etc.
- After tailoring, the process specification for the project is available.





# Effort estimation

- The primary reason for cost and schedule estimation is cost-benefit analysis, and project monitoring and control.
- A more practical use of these estimates is bidding for software projects.
- The bulk of the cost of software development is due to the human resources needed.
- By properly including the overheads like cost of h/w, s/w, office space etc. in the cost of a person-month, effort estimates can be converted into cost.
- Estimates can be based on subjective opinion of some person or determined through the use of models.



# Uncertainties in effort estimation

- The accuracy of the estimate will depend on the amount of reliable information we have about the final product.
- When project is in feasibility study phase, we have only some idea of the classes of data the system will get and produce and the major functionality of the system.
- The effort estimation based on this type of information cannot be accurate.
- The obtainable accuracy of the estimates as it varies with the different phases is shown in figure:

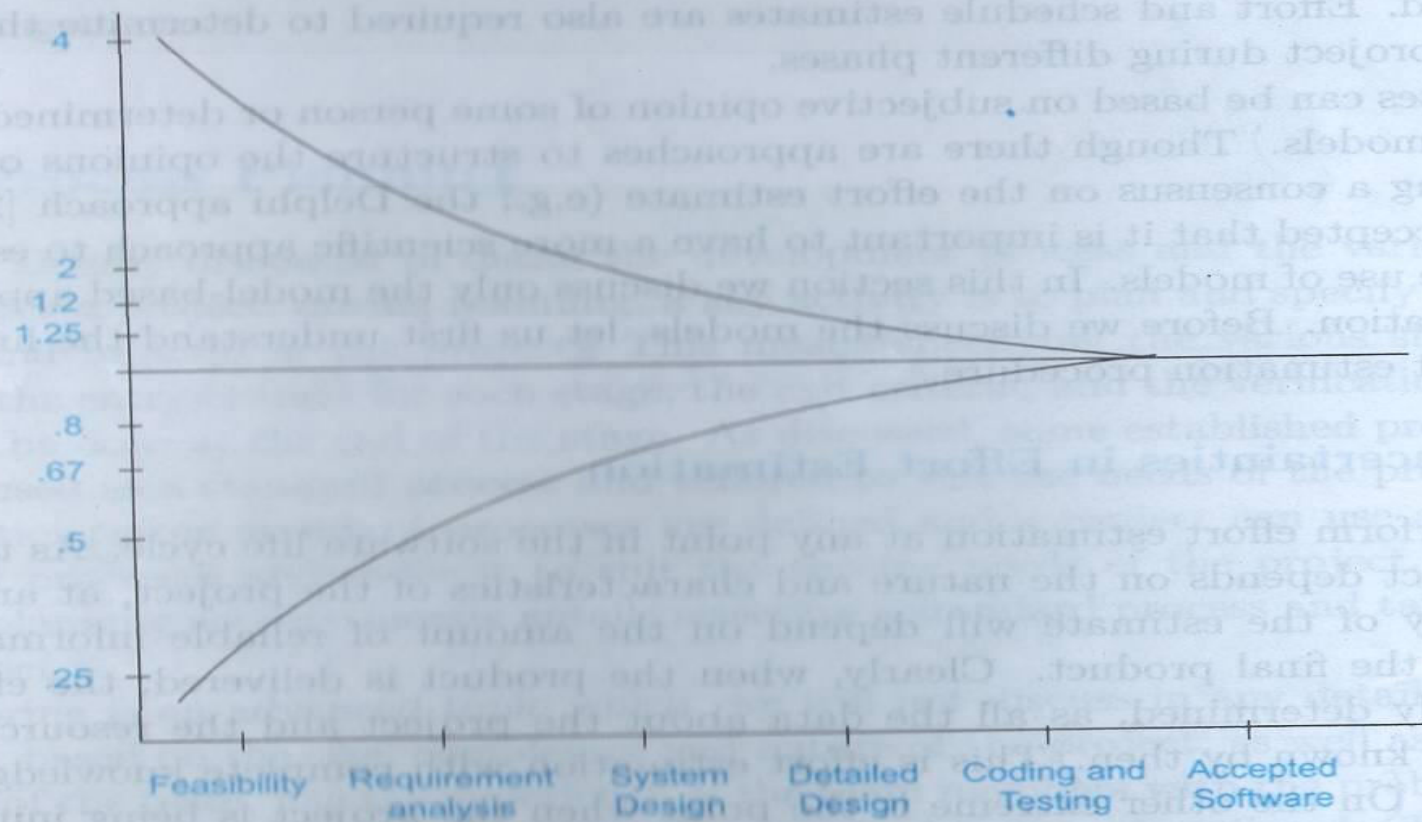


Figure 5.1: Accuracy of effort estimation.



# Building effort estimation models

- An estimation model can be viewed as a “function” that outputs the effort estimate.
- Estimation model cannot work in a vacuum; it needs inputs to produce the effort estimate as output.
- The primary factor that controls the effort is the size of the project.

# Building effort estimation models

- One common approach for estimating effort is to make it a function of project size, and the equation of effort is considered as:

$$\text{EFFORT} = a * \text{SIZE} ^ b$$

Where a and b are constants, and project size is generally in KLOC or function points

- Values for these constants for a particular process are determined through *regression analysis*, which is applied to data about the projects that has been performed in the past.

# Top – Down approach

- The approach of ***determining total effort from the total size*** is referred to as the top-down approach,
- Overall effort is first determined and then from this, the effort for different parts are obtained.
- Once size estimates for components are available, to get the overall size estimate for the system, the estimates of all the components can be added up.
- ⊗ Effort for developing a system is not the sum of effort for developing the components because additional effort is needed for integration and other activities.



# Top – Down approach

- This key feature, that the system property is the sum of the properties of its parts, holds for size but not for effort, and is the main reason that size estimation is considered easier than effort estimation
- With top-down models, if the size estimate is inaccurate, the effort estimate produced by the models will also be inaccurate.
- When estimating software size, the best way may be to get as much detail as possible about the software.



# Bottom-Up estimation approach

- In this approach, the project is first divided into tasks and then estimates for the different tasks of the project are first obtained.
- Once the project is partitioned into smaller tasks, it is possible to directly estimate the effort required for them, specially if tasks are relatively small.
- A risk of bottom-up methods is that one may omit some important activities in the list of tasks.
- Top-down approach requires the information of size of project and bottom-up approach requires list of tasks in the project.





# Steps of Bottom-Up approach

1. Identify modules in the system and classify them as simple, medium or complex.
2. Determine the average coding effort for modules
3. Get the total coding effort using the coding effort of different types of modules and the counts for them
4. Using the effort distribution for similar projects, estimate the effort for other tasks and the total effort
5. Refine the estimates based on project-specific factors.



# Bottom-Up estimation approach

- Bottom-up approach lends itself to a judicious mixture of experience and data.
- If suitable past data are not available, one can estimate the coding effort using experience.
- For small projects, many people find this approach natural and comfortable.



# COCOMO Model

- A top-down model can depend on many different factors, giving rise to multivariable models.
- One approach for building multivariable models is to start with an initial estimate determined by using the static single-variable model equations, which depend on size, and then adjusting the estimates based on other variables.
- This approach implies that size is the primary factor for cost.

# COCOMO Model

- The COnstructive COst MOdel also estimates the total effort in terms of person-months. The basic steps in this model are:
  1. Obtain an initial estimate of the development effort from the estimate of thousands of delivered lines of source code (KLOC).
  2. Determine a set of 15 multiplying factors from different attributes of the project.
  3. Adjust the effort estimate by multiplying the initial estimate with all the multiplying factors.

# COCOMO Model

- The initial estimate (nominal estimate) is determined by an equation of the form used in the static single-variable models, using KLOC as the measure of size.
- To determine the initial effort  $E_i$  in person-months the equation used is of type:  
$$E_i = a * (KLOC)^b$$
- In COCOMO, projects are categorized into three types – Organic, semidetached, and embedded.

# COCOMO Model

- Organic projects are those that are relatively straightforward and developed by a small team.
- Embedded projects are those that ambitious and novel, with stringent constraints from the environment and high requirements for such aspects as interfacing and reliability.
- The constants  $a$  and  $b$  for different systems are:

System	$a$	$b$
Organic	3.2	1.05
Semidetached	3.0	1.12
Embedded	2.8	1.20



# COCOMO Model

- There are 15 different attributes, called cost driver attributes, that determine the multiplying factors.
- These factors depend on product, computer, personnel and technology attributes. For example, product complexity, Analyst's capability, modern tools etc.
- Each cost driver has a rating scale, and for each rating, a multiplying factor is provided as shown in following table:

# COCOMO Model

Cost Drivers	Rating				
	Very Low	Low	Nominal	High	Very High
<b>Product Attributes</b>					
RELY, required reliability	.75	.88	1.00	1.15	1.40
DATA, database size		.94	1.00	1.08	1.16
CPLX, product complexity	.70	.85	1.00	1.15	1.30
<b>Computer Attributes</b>					
TIME, execution time constraint			1.00	1.11	1.30
STOR, main storage constraint			1.00	1.06	1.21
VITR, virtual machine volatility		.87	1.00	1.15	1.30
TURN, computer turnaround time		.87	1.00	1.07	1.15
<b>Personnel Attributes</b>					
ACAP, analyst capability	1.46	1.19	1.00	.86	.71
AEXP, application exp.	1.29	1.13	1.00	.91	.82
PCAP, programmer capability	1.42	1.17	1.00	.86	.70
VEXP, virtual machine exp.	1.21	1.10	1.00	.90	
LEXP, prog. language exp.	1.14	1.07	1.00	.95	
<b>Project Attributes</b>					
MODP, modern prog. practices	1.24	1.10	1.00	.91	.82
TOOL, use of SW tools	1.24	1.10	1.00	.91	.83
SCHED, development schedule	1.23	1.08	1.00	1.04	1.10

Table 5.1: Effort multipliers for different cost drivers.



# COCOMO Model

- The multiplying factors for all 15 cost drivers are multiplied to get the effort adjustment factor (EAF).
- The final effort estimate  $E$  is obtained by multiplying the initial estimate by the EAF.

$$E = EAF * E_i$$

- By this method, the overall cost of the project can be estimated.
- Effort for a phase is a defined percentage of the overall effort.

# COCOMO Model

- The percentage of total effort spent in a phase varies with the type and size of the project.
- The percentages for an organic software project are

Phase	Size			
	Small	Intermediate	Medium	Large
	2 KLOC	8 KLOC	32 KLOC	128 KLOC
Product design	16	16	16	16
Detailed design	26	25	24	23
Code and unit test	42	40	38	36
Integration and test	16	19	22	25

Table 5.2: Phase-wise distribution of effort.

# COCOMO Model

- If the total effort estimate for an organic software system is 20PM and the size estimate is 20KLOC, then the percentage effort for the coding and unit testing phase will be,  
$$40 + (38-40) / (32-8) * 20 = 39\%$$
- In COCOMO, the detailed design and code and unit testing are sometimes combined into one phase called the programming phase.



# Project scheduling and staffing

- Once the effort is estimated, various schedules are possible, depending on the number of resources put on the project.
- For example, for a project whose effort estimate is 56 person-months, a total schedule of 8 months is possible with 7 people, or 7 months with 8 people, or 9 months with 6 people.
- Manpower and months are not fully interchangeable in a software project.



# Project scheduling and staffing

- Brooks Law: “ man and months are interchangeable only for activities that require no communication among men.”
- In a project, the scheduling activity can be broken into two sub activities:
  1. Determining the overall schedule i.e. project duration with major milestones, and
  2. Developing the detailed schedule of the various tasks.

# Overall scheduling

- One method to determine the normal overall schedule is to determine it as a function of effort
- The IBM Federal Systems Division found that the total duration,  $M$ , in calendar months can be estimated by,  $M = 4.1 * E^{.36}$
- In COCOMO, the equation for schedule for an organic type of software is,  $M = 2.5 * E^{.38}$
- These methods can be used as guidelines, but other factors can also be considered.



# Overall scheduling

- **Rule of Square root check:** the proposed schedule can be around the square root of the total effort in person-months.
- To determine the milestones, we must first understand the manpower ramp-up that usually takes place in a project.
- That is, in the beginning and end, few people work on the project. The human resources requirement peaks during coding and unit testing.



# Overall scheduling

- For ease of scheduling, particularly for smaller projects, often the required people are assigned together around the start of the project.
- Some people being can be unoccupied at the start and toward the end, but they can be assigned to supporting project activities like training and documentation.
- Given the effort estimate for a phase, we can determine the duration of the phase if we know the manpower ramp-up.





Peak Team  
Size

Design

Build

Test

Figure 5.2: Manpower ramp-up in a typical project.



# Detailed scheduling

- Once the milestones and the resources are fixed, it is time to set the detailed scheduling.
- For detailed schedules, the major tasks fixed while planning the milestones are broken into small schedulable activities in a hierarchical manner.
- At each level of refinement, the project manager determines the effort for the overall task from the detailed schedule and checks it against the effort estimates.
- If this detailed schedule is not consistent with the overall schedule and effort estimates, the detailed schedule must be changed.
- If it is found that the best detailed schedule cannot match the milestone effort and schedule, then the earlier estimates must be revised. Thus, scheduling is an iterative process.



# Detailed scheduling

- Rarely will a project manager complete the detailed schedule of the entire project all at once.
- Once the overall schedule is fixed, detailing for a phase may only be done at the start of that phase.
- For detailed scheduling, tools like Microsoft Project or a spreadsheet can be very useful.
- A detailed project schedule is never static. Changes are done as and when the need arises.
- The final schedule is often the most live project plan document.



# Team Structure

- The number of resources is fixed when schedule is being planned.
- Detailed scheduling is done only after actual assignment of people has been done.
- **Chief Programmer Team**:-In hierarchical organization, the project manager is responsible for all major technical decisions of the project.
- He does most of the design and assigns coding of the different parts of the design to the programmers.
- The team typically consists of programmers, testers, a configuration controller and possibly a librarian for documentation.



# Team Structure

- There may be other roles like database manager, network manager, backup project manager or backup configuration controller.
- For a small project, a one-level hierarchy is enough. For large projects, this organization can be extended easily by partitioning the project into modules.
- We can have module leaders who are responsible for all tasks related to their module and have a team with them for performing these tasks.

# Team Structure

- **Egoless Team**:- Egoless teams consists of ten or fewer programmers. The goals of the group are set and input from every member is taken for major decisions.
- Group leadership rotates among the group members.
- Egoless teams are sometimes called democratic teams. This structure allows input from all members, which can lead to better decisions for difficult problems
- This structure is well suited for long-term research projects, that do not have time constraint.



# Team Structure

- For very large product developments, another structure has emerged.
- This structure recognizes that there are three main task categories in software development : management related, development related, and testing related.
- It is often desirable to have the test and development team be relatively independent.
- In this structure, there is an overall unit manager, under whom there are three small hierarchic organizations – for program management, for development, and for testing.



# Team Structure

- The primary job of developers is to write code and they work under a development manager.
- The responsibility of the testers is to test the code and they work under a test manager.
- The program managers provides the specifications for what is being built, and ensure that development and testing are properly coordinated.





# Quality Plan

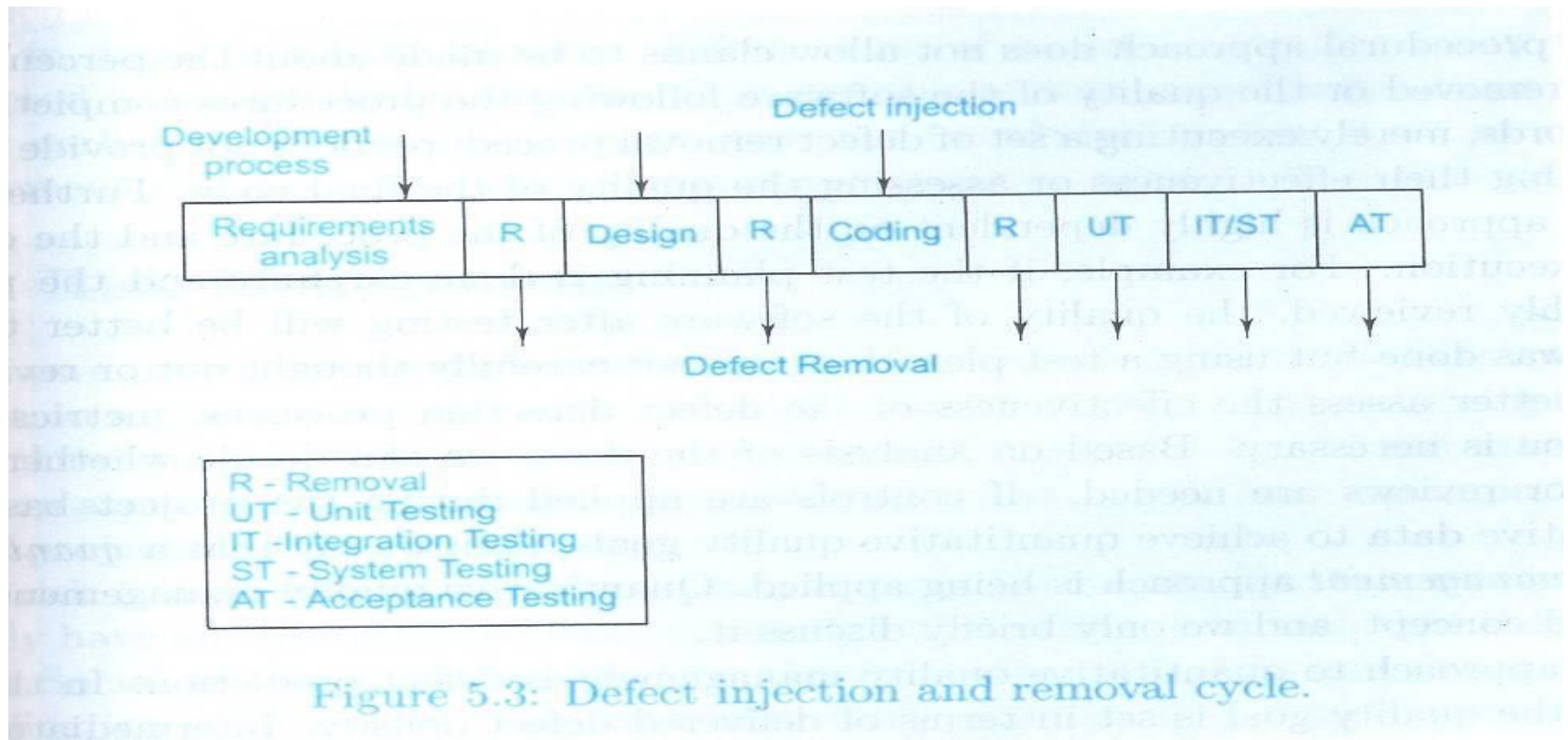
- “delivered defect density” – the number of defects per unit size in the delivered software is the definition of the quality.
- By defect we mean something in software that causes the software to behave in a manner that is inconsistent with the requirements or needs of the customer.
- A QC task is one whose main purpose is to identify defects.
- The purpose of a quality plan in a project is to specify the activities that need to be performed for identifying and removing defect and the tools and methods that may be used for that purpose.




# Quality Plan

- To ensure that the delivered software is of good quality, it is essential to ensure that all work products like the requirements specification, design and test plan are also of good quality.

# Defect Injection and Removal Cycle



- 
- Software development is a highly people-oriented activity and hence it is error-prone.
  - Defect can be injected in software at any stage during its evolution. These injection stages are primarily the requirements specification, the high-level design, the detailed design and coding.
  - For high-quality software, the final product should have as few defects as possible.
  - For delivery of high-quality software, active removal of defects through the quality control activities is necessary.
  - The QC activities for defect removal include requirements reviews, design reviews, code reviews, unit testing, integration testing, system testing, acceptance testing.



# Verification v/s Validation

- The task of quality management is to plan suitable quality control activities and then to properly execute and control them so the projects quality goals are achieved.
- With respect to quality control the terms verification and validation are often used.
- **Verification** is the process of determining whether or not the products of a given phase of software development fulfill the specification established during previous phase.
- **Validation** is the process of evaluating software at the end of the software development to ensure compliance with the software requirements.
- For high reliability we need to perform both activities. Together they are called **V&V activities**.



# Approaches to Quality Management

- Reviews and testing are two most common QC activities.
- Procedural approach:- in the procedural approach to quality management, procedures and guidelines for the review and testing activities are planned. During project execution, they are carried out according to the defined procedures.
- Defect Prediction:- in this approach, the quality goal is set in terms of delivered defect density. Then the actual number of defects are compared to the estimated defect levels.
- Other two approaches are Quantitative Quality management approach and Statistical process control (SPC).



# Quality Plan

- The quality plan for a project is what drives the quality activities in the project.
- The quality plan specifies the quality control tasks that will be performed in the project.
- Much of the quality plan revolves around testing and reviews.



# Risk Management

- Risk management is an attempt to minimize the chances of failure caused by unplanned events.
- The aim of risk management is to minimize the impact of risks in the projects that are undertaken.
- A Risk is a probabilistic event – it may or may not occur.





# Risk Management Concepts

- Risk is defined as an exposure to the chance of injury or loss.
- A risk implies that there is a possibility that something negative may happen. In the context of software projects, negative implies that there is an adverse effect on cost, quality or schedule.
- Risk Management is the area that tries to ensure that the impact of risks on cost, quality and schedule is minimal.
- Risk management begins where normal project management ends.
- It deals with events that are infrequent, somewhat out of the control of the project management, and which can have a major impact on the project.



# Risk Management Concepts

- Risk management has to deal with identifying the undesirable events that can occur, the probability of their occurring, and the loss if an undesirable event does occur.
- Once this is known, strategies can be formulated for either reducing the probability of the risk materializing or reducing the effect of risk materializing.
- Risk management revolves around ***risk assessment*** and ***risk control***.

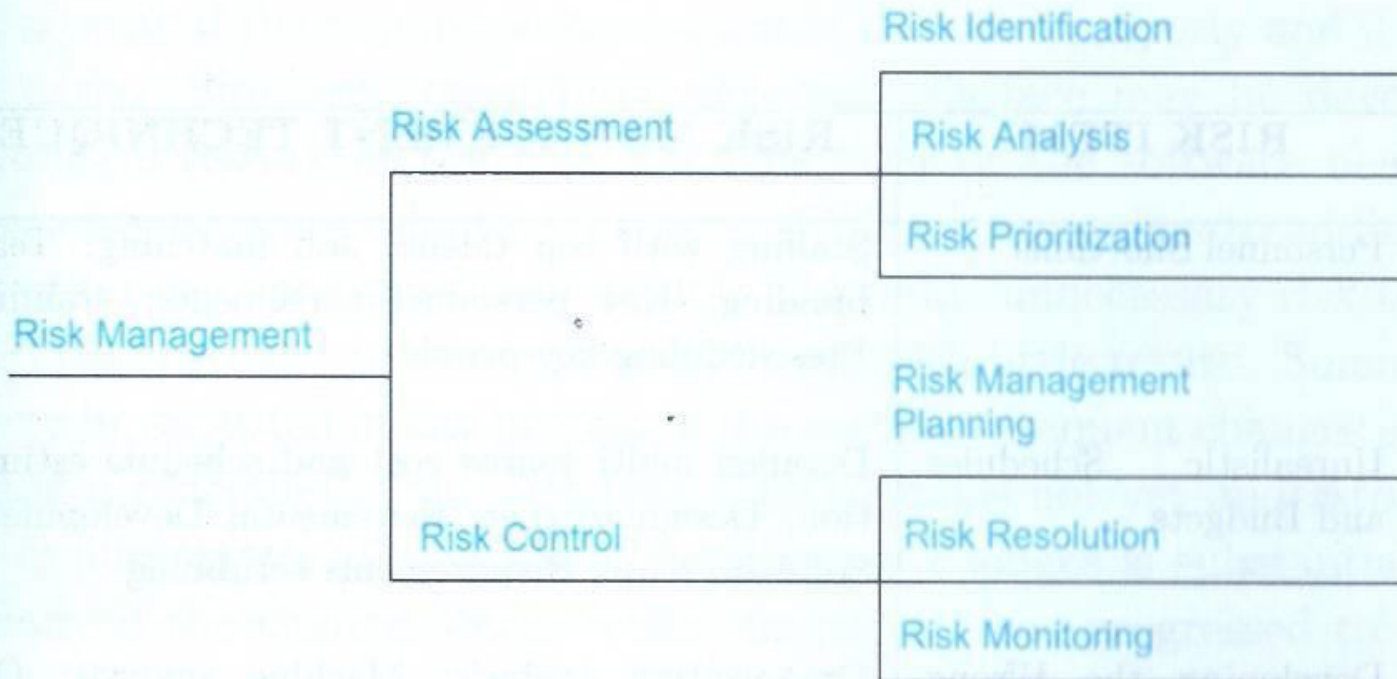




Figure 5.4: Risk management activities.




# Risk Assessment

- Risk assessment is an activity that must be undertaken during project planning.
- This involves identifying the risks, analyzing them, and prioritizing them on the basis of the analysis.
- The goal of risk assessment is to prioritize the risks so that attention and resources can be focused on the more risky items.

- 
- Methods that can aid ***risk identification*** include checklists of possible risks, surveys, meetings and brainstorming, and reviews of plans, processes, and work products.
  - Checklists of frequently occurring risks are probably the most common tool for risk identification.
  - The other methods are decision driver analysis, assumption analysis, and decomposition.
  - Decision driver analysis involves questioning and analyzing all the major decisions taken for the project.
  - If a decision has been driven by factors other than technical and management reasons, it is likely to be a source of risk in the project

- 
- In ***risk analysis***, the probability of occurrence of a risk has to be estimated, along with the loss that will occur if the risk does materialize.
  - This is often done through discussion, using experience and understanding of the situation.
  - Cost models are used for this.
  - The other approaches for risk analysis include
    - Decision analysis: studying the probability and the outcome of possible decisions.
    - Network analysis: understanding the task dependencies to decide critical activities and the probability and cost of their not being completed on time.
    - Quality Factor analysis: risk on the various quality factors like reliability and usability.
    - Performance analysis: valuating performance early through simulations.

- 
- Once the probabilities of risks materializing and losses due to materialization of different risks have been analyzed, they can be prioritized.
  - One approach for **prioritization** is through the concept of risk exposure (RE), which is sometimes called risk impact.

$$RE = Prob(UO) * Loss(UO)$$

- where  $Prob(UO)$  is the probability of the risk materializing (Undesirable Outcome) and  $Loss(UO)$  is the total loss incurred due to the unsatisfactory outcome.
- For risk prioritization using RE, the higher the RE, the higher the priority of the risk item.



# The Project Plan

- The project plan sets out the resources available to the project, the work breakdown and a schedule for carrying out the work.
- In some organizations, the project plan is a single document that includes the different types of plan. That is, Quality plan, validation plan, configuration management plan, maintenance plan, staff development plan etc.
- In other cases, the project plan is solely concerned with the development process. That is, separate plans for separate processes.





# Sections of Plan

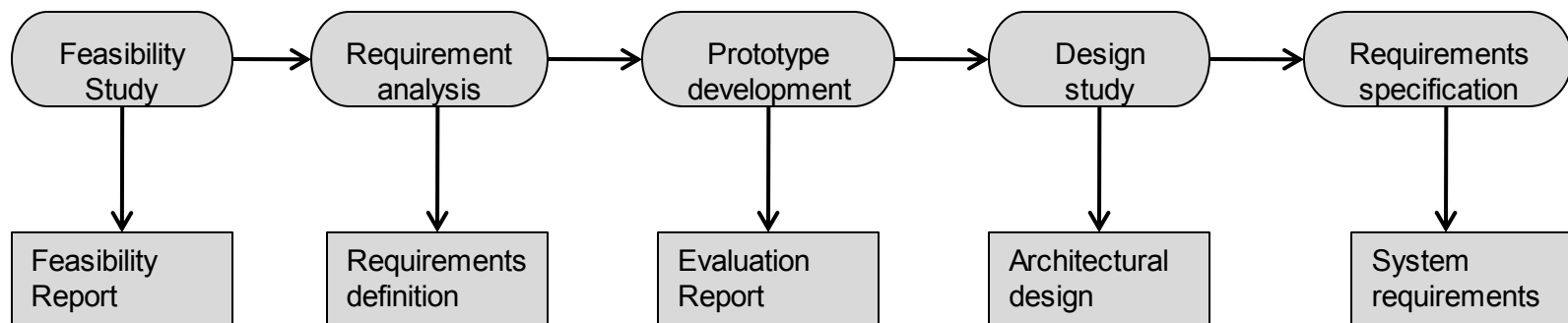
- **Introduction:** this briefly describes the objectives of the project and sets out the constraints like budget, time that affect the project management.
- **Project organization:** this describes the way in which the development team is organized, the people involved and their roles in the team.
- **Risk analysis:**
- **Hardware and software resource requirements:**
- **Work breakdown:** this sets out the breakdown of the project into activities and identifies the milestones and deliverables associated with each activity.
- **Project schedule:**
- **Monitoring and reporting mechanisms:** this defines the management reports that should be produced, when these should be produced and the project monitoring mechanisms used.



# Milestones and Deliverables

- Milestone is a recognizable end-point of a software process activity.
- At each milestone, there should be a formal output, such as a report, that can be presented to management.
- Milestone reports may simply be a short report.
- Milestones should represent the end of a distinct, logical stage in the project.
- Indefinite milestones such as 'Coding is 80% complete' are useless for project management.
- A deliverable is a project result that is delivered to the customer. It is usually delivered at the end of some major project phase such as specification or design.

- Deliverables are usually milestones, but milestones need not be deliverables.
- Milestones may be internal project results that are used by the project manager to check project progress but which are not delivered to the customer.
- To establish milestones, the software process must be broken down into basic activities with associated outputs.
- Following figure shows activities (top row) and milestones.





# Software Architecture

- For building the specified software system, designing the software architecture is a key step.
- Any complex system is composed of sub systems that interact under the control of system design such that the system provides the expected behaviour.
- While designing such a system, therefore, the logical approach is to identify the sub-systems that should compose the system, the interfaces of these subsystems, and the rules for interaction between the subsystems.
- This is what software architecture aims to do.



# What is Software Architecture?

- It is during the architecture design where choices like using some type of middleware, or some type of back end database, or some type of server, or some type of security component are made.
- At a top level, architecture is a design of a system which gives a very high level view of the parts of the system and how they are related to form the whole system.
- ***the software architecture of a system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.***



# Uses of Software Architecture

- Understanding and Communication
- Reuse
- Construction and Evolution
- Analysis



# Understanding and Communication

- An architecture description is primarily to communicate the architecture to its various stakeholders.
- The stakeholders are; the users, the clients, the builders, and the architects.
- An architecture description is an important means of communication between these various stakeholders.
- Through this description the stakeholders gain an understanding of some macro properties of the system and how the system intends to fulfil the functional and quality requirements.
- An architecture description of the proposed system describes how the system will be composed, when it is built (Understanding of system).



# Reuse

- Reuse is considered one of the main techniques by which productivity can be improved, thereby reducing the cost of software.
- Architecture also facilitates reuse among products that are similar and building product families such that the common parts of these different but similar products can be reused.
- It is very hard to achieve this type of reuse at a detail level.





# Construction and Evolution

- An architecture partitions the system into parts.
- The parts specified in an architecture are relatively independent, they can be built independently.
- The dependence between parts coming through their relationship.
- a software system also evolves with time. During evolution, often new features are added to the system.
- The architecture of the system can help in deciding where to add the new features with minimum complexity and effort, and what the impact on the rest of the system might be of adding the new features.



# Analysis

- Many engineering disciplines use models to analyse design of a product for its cost, reliability, performance, etc.
- Architecture opens such possibilities for software also.
- It is possible to analyse or predict the properties of the system being built from its architecture.
- Such an analysis can help determine whether the system will meet the quality and performance requirements, and if not, what needs to be done to meet the requirements.



# WBS

- A work breakdown structure (WBS), in project management and systems engineering, is a deliverable oriented decomposition of a project into smaller components.
- It defines and groups a project's discrete work elements in a way that helps organize and define the total work scope of the project.
- A “work breakdown structure element” may be a product, data, a service, or any combination.
- A WBS also provides
  - Necessary framework for detailed cost estimating and control
  - Guidance for schedule development and control

# WBS

- The work breakdown structure is a tree structure, which shows a subdivision of effort required to achieve an objective.
  - In a project or contract, the WBS is developed by
    - starting with the end objective and
    - successively subdividing it into manageable components in terms of:
      - size,
      - duration, and
      - responsibility
- (e.g., systems, subsystems, components, tasks, subtasks, and work packages)
- which include all steps necessary to achieve the objective.

# WBS Example

