# Course Code: 20MCA107

# Course Name: ADVANCED SOFTWARE ENGINEERING

PROF. NEETHU MOHAN
ASSOCIATE PROFESSOR,
DEPARTMENT OF COMPUTER APPLICATIONS,
LOURDES MATHA COLLEGE OF SCIENCE AND TECHNOLOGY.

# Module 1

- Introduction to Software Engineering: What is Software Engineering, Characteristics of Software.
- Life cycle of a software system: software design, development, testing, deployment, Maintenance.
- Project planning phase: project objectives, scope of the software system, empirical estimation models, COCOMO, staffing and personnel planning.

- Software Engineering models: Predictive software engineering models, model approaches, prerequisites, predictive and adaptive waterfall, waterfall with feedback (Sashimi), incremental waterfall, V model; Prototyping and prototyping models.
- Software requirements specification, Eliciting Software requirements, Requirement specifications, Software requirements engineering concepts, Requirements modelling, Requirements documentation. Use cases and User stories.

## TEXT BOOKS REFERRED :-

1. Philip A. Laplante, What Every Engineer Should Know about Software Engineering, CRC Press
   [Socratic form :- that is, in the form of questions and answers.]
2. Pressman, R.S., “Software Engineering: A Practitioner's Approach”, McGraw Hill SE, 7th Edition, (2010).

# Introduction to Software Engineering:

- What is Software Engineering
- Characteristics of Software.

# What is software?

Software is not just the programs but also it includes

1. all associated documentation
2. configuration data

These are needed to make the software correctly

**Software** : It is the set of :

1) Instructions (programs) that when executed provide desired features, functions and performance.

2) Data structures that enable the programs to adequately manipulate information.

3) Documents that describe the use and operation of the programs.

- Software engineers are concerned with developing software products.
- **Software products** means software(set of programs) which can be sold to a customer.
- There are two fundamental types of software:
  1. Generic products
  2. Customized products

**Generic products**

- Developed by an organization or a company and sold on the open market to any customer who is able to buy them.

**Customized products**

- These are developed for a particular customer.
- A software contractor develops the software especially for that customer.

# What is Software Engineering?

- Software engineering is an engineering discipline that is concerned with all aspects of software production.
- That is from the early stages of system specification to maintaining the system after it has gone into use.
- Software engineering is the establishment and use of engineering principles in order to obtain economically feasible software that is reliable and works efficiently on real machines.

# What is Software Engineering?

- Software engineering is “a systematic approach to the analysis, design, assessment, implementation, test, maintenance and reengineering of software.
- In the software engineering approach, several models for the software life cycle are defined, and many methodologies for the definition and assessment of the different phases of a life-cycle model”

# Why Learn Software Engineering ?

- Software engineers play an important role in today's information driven society.
- Software engineering is ***one of the fastest growing professions with excellent job prospects*** predicted throughout the coming decade.
- Software engineering brings together various skills and responsibilities.

**How do software engineers spend their time on the job?**

- Software engineers probably spend less than 10% of their time writing code.
- The other 90% of their time is involved with other activities that are more important than writing code.
- These activities include:

1. Eliciting requirements
2. Analyzing requirements
3. Writing software requirements documents
4. Building and analyzing prototypes
5. Developing software designs

1. Writing software design documents
2. Researching software engineering techniques or obtaining information about the application domain
3. Developing test strategies and test cases
4. Testing the software and recording the results
5. Isolating problems and solving them

1. Learning to use or installing and configuring new software and hardware tools
2. Writing documentation such as users manuals
3. Attending meetings with colleagues, customers, and supervisors
4. Archiving software or readying it for distribution

- Studying software engineering opens up a wide range of career opportunities like Software Developer, Test Engineer, Project Manager, Software Architect, Software Quality Manager, Doctoral Student/Scientist

# Characteristics of Software

- Software can be characterized by any number of qualities.
- **External qualities**, such as usability and reliability, are visible to the user.
- **Internal qualities** are those that may not be necessarily visible to the user, but help the developers to achieve improvement in external qualities.
- For example, good requirements and design documentation might not be seen by the typical user, but these are necessary to achieve improvement in most of the external qualities.

**The most commonly discussed qualities of software are:**

- **Reliability**
- **Correctness**
- **Performance**
- **Usability**
- **Interoperability**
- **Maintainability**
- **Evolvability**
- **Repairability**
- **Portability**
- **Verifiability**
- **Traceability**
- **And few negative qualities like** Fragility, Immobility, Needless complexity, Needless repetition, Opacity, Rigidity, Viscosity **that should be avoided**

# Reliability

- Software reliability can be defined informally in a number of ways.
- For example, can the user "**depend on**" the software?
- Other characterizations of a reliable software system include:
  - The system "stands the test of time."
  - There is an absence of known catastrophic errors (those that disable or destroy the system).
  - The system recovers "gracefully" from errors.
  - The software is robust.
  - Downtime is below a certain threshold.
  - The accuracy of the system is within a certain tolerance.
  - Real-time performance requirements are met consistently

# How do you measure software reliability?

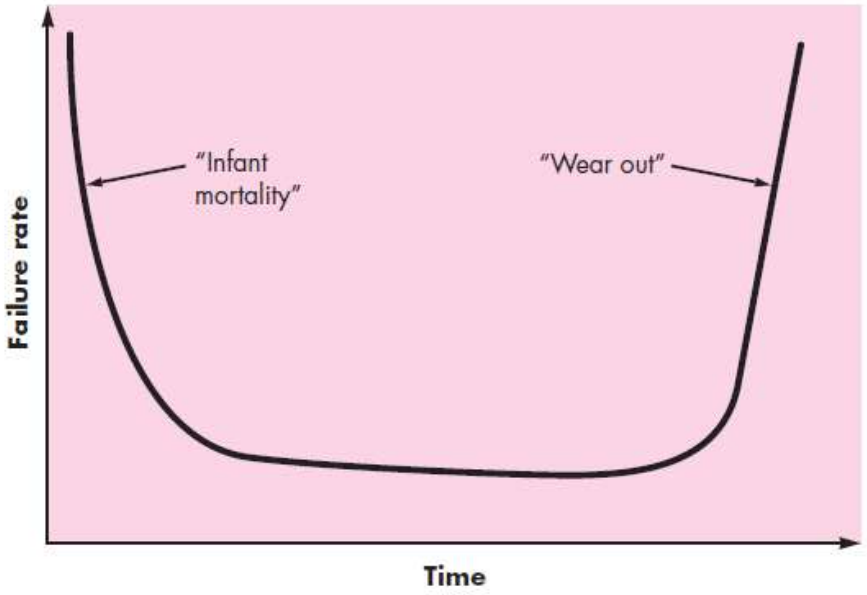
- Software reliability can be defined in terms of statistical behavior;
- that is, the probability that the software will operate as expected over a specified time interval.
- Let *S* be a software system and let *T* be the time of system failure. Then the reliability of *S* at time *t*, denoted *r(t),* is the probability that *T* is greater than *t* ; that is,

$$\mathbf{r(t) = P(T > t)}$$

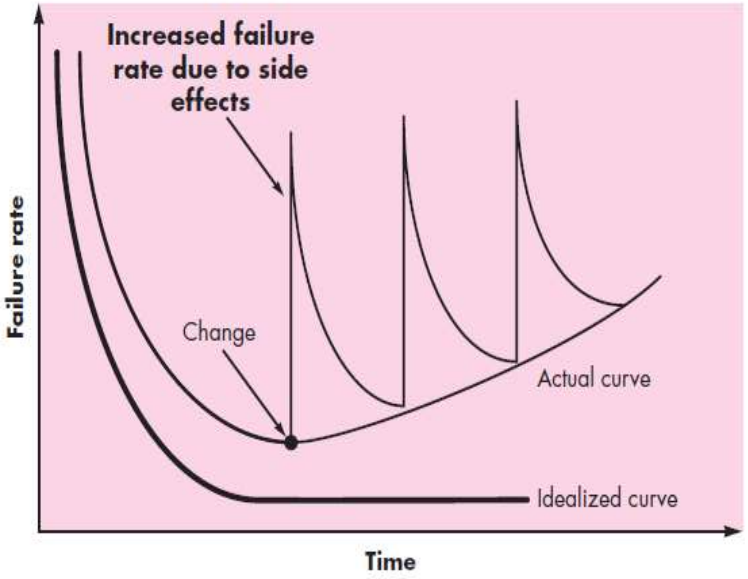
- This is the probability that a software system will operate without failure for a specified period.
- Thus, a system with reliability function $r(t) = 1$ will never fail

- The failure intensity is initially high, as would be expected in new software as faults are detected during testing.
- The number of failures would be expected to decrease with time, presumably as failures are uncovered and repaired.
- The **bathtub curve** is often used to explain the failure function for physical components that wear out, electronics, and even biological systems.
- We expect a large number of failures early in the life of a product (from manufacturing defects) and then a steady decline in failure incidents until later in the life of that product when it has “worn out” or, in the case of biological entities, died.

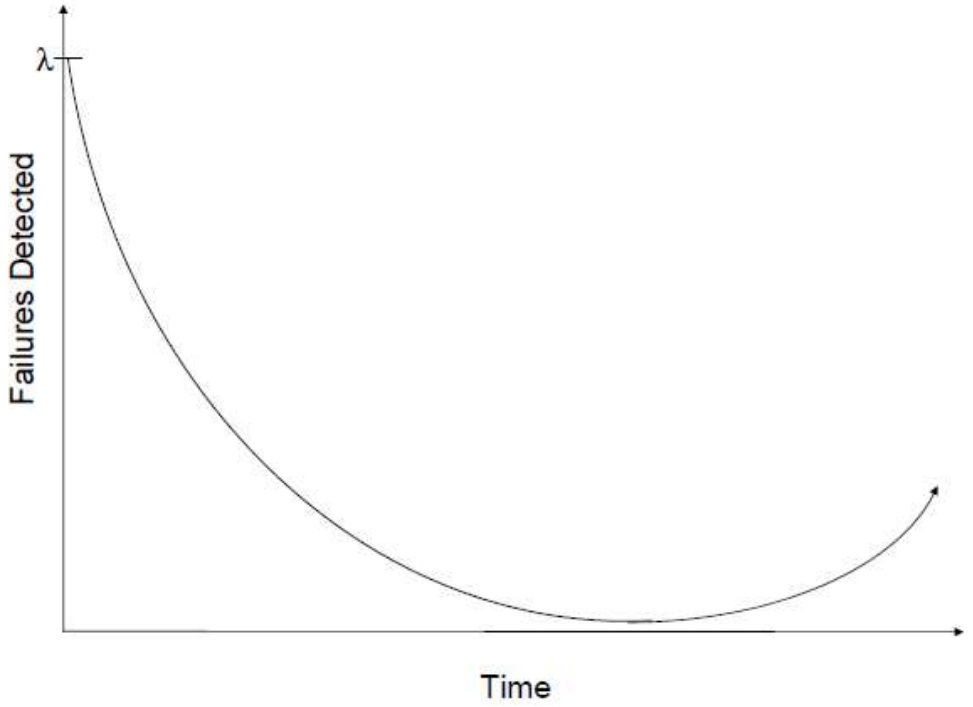
Failure curve for hardware




Failure curve for software




# Bathtub curve



**Correctness**

- Software correctness is closely related to reliability and the terms are often used interchangeably.
- The main difference is that minor deviation from the requirements is strictly considered a failure and hence means the software is incorrect.
- Correctness can be measured in terms of the number of failures detected over time.

## Performance

- Performance is a measure of some required behavior — often with respect to some relative time constraint.
- For example,
  - the baggage inspection system may be required to process 100 pieces of luggage per minute.
  - a photo reproduction system might be required to digitize, clean, and output color copies at a rate of one every two seconds.
- One method of measuring performance is based on mathematical or algorithmic complexity. Another approach involves directly timing the behaviour of the completed system with logic analyzers and similar tools.

**Usability**

- Usability is a measure of how easy the software is for humans to use.
- Software usability is synonymous with ease-of-use, or user-friendliness.
- Usually informal feedback from users, as well as surveys, focus groups, and problem reports are used to measure/determine usability

## Interoperability

- This quality refers to the ability of the software system to coexist and cooperate with other systems.
- In many systems, special software called middleware is written to enhance interoperability. In other cases, standards are used to achieve better interoperability.
- For example, in embedded systems, the software must be able to communicate with various devices using standard bus structures and protocols.
- Interoperability can be measured in terms of compliance with open system standards. These standards are typically specific to the application domain.

- An open system is an extensible collection of independently written applications that cooperate to function as an integrated system.
- This concept is related to interoperability.
- Open systems differ from open source code, which is source code that is made available to the user community for improvement and correction.
- An open system allows the addition of new functionality by independent organizations through the use of interfaces whose characteristics are published.
- Any software engineer can then take advantage of these interfaces, and thereby create software that can communicate using the interface.
- Open systems also permit different applications written by different organizations to interoperate.

**Maintainability, Evolvability and Repairability**

- A software system in which changes are relatively easy to make has a high level of **maintainability**.
- Maintainability can be decomposed into two contributing properties — evolvability and repairability.
- **Evolvability** is a measure of how easily the system can be changed to accommodate new features or modification of existing features.
- **Repairability** is the ability of a software defect to be easily repaired.
- Measuring these qualities is not always easy, and is often based on anecdotal observation. This means that changes and the cost of making them are tracked over time.

**Portability**

- Software is portable if it can run easily in different environments.
- The term environment refers to the hardware on which the system resides, the operating system, or other software in which the system is expected to interact.
- Portability is difficult to measure, other than through anecdotal observation.
- Portability is achieved through a deliberate design strategy in which hardware-dependent code is confined to the fewest code units as possible.

**Verifiability**

- A software system is verifiable if its properties, including all of those previously introduced, can be verified easily.
- One common technique for increasing verifiability is through the insertion of software code that is intended to monitor various qualities such as performance or correctness.
- Modular design, rigorous software engineering practices, and the effective use of an appropriate programming language can also contribute to verifiability.

**Traceability**

- Traceability is concerned with the relationships between requirements, their sources, and the system design.
- A high level of traceability ensures that the software requirements flow down through the design and code and then can be traced back up at every stage of the process.
- Traceability can be obtained by providing links between all documentation and the software code.

- A set of software code **qualities in the negative -** these are qualities of the code that **need to be reduced or avoided altogether**.
- **Fragility** — When changes cause the system to break in places that have no conceptual relationship to the part that was changed. This is a sign of poor design.
- **Immobility** — When the code is hard to reuse.
- **Needless complexity** — When the design is more elaborate than it needs to be. This is sometimes also called “gold plating.”

- **Needless repetition** — This occurs when cut-and-paste (of code) is used too frequently.
- **Opacity** — When the code is not clear.
- **Rigidity** — When the design is hard to change because every time you change something, there are many other changes needed to other parts of the system.
- **Viscosity** — When it is easier to do the wrong thing, such as a quick and dirty fix, than the right thing.

Negative Code Qualities and Their Positives

| Negative Code Quality | Positive Code Quality |
|---|---|
| Fragility | Robustness |
| Immobility | Reusability |
| Needless complexity | Simplicity |
| Needless repetition | Parsimony |
| Opacity | Clarity |
| Rigidity | Flexibility |
| Viscosity | Fluidity |

- Achieving these qualities is a direct result of a good software architecture, solid software design, and effective coding practices.
- There are many software qualities, some mainstream, others more esoteric or application-specific.

# Module - I

## Life cycle of a software system:

- Software Design
- Development
- Testing
- Deployment
- Maintenance.

- Producing a software application is relatively simple in concept: ***Take an idea and turn it in to a useful program.***
- Unfortunately for projects of any real scope, there are countless ways that a simple concept can go wrong.
- Programmers may not understand what users want or need so they build the wrong application.

- The program might be full of bugs that it’s frustrating to use, impossible to fix, and can’t be enhanced over time.
- Software engineering includes techniques for avoiding the many pitfalls.
- It ensures the final application is effective, usable, and maintainable.

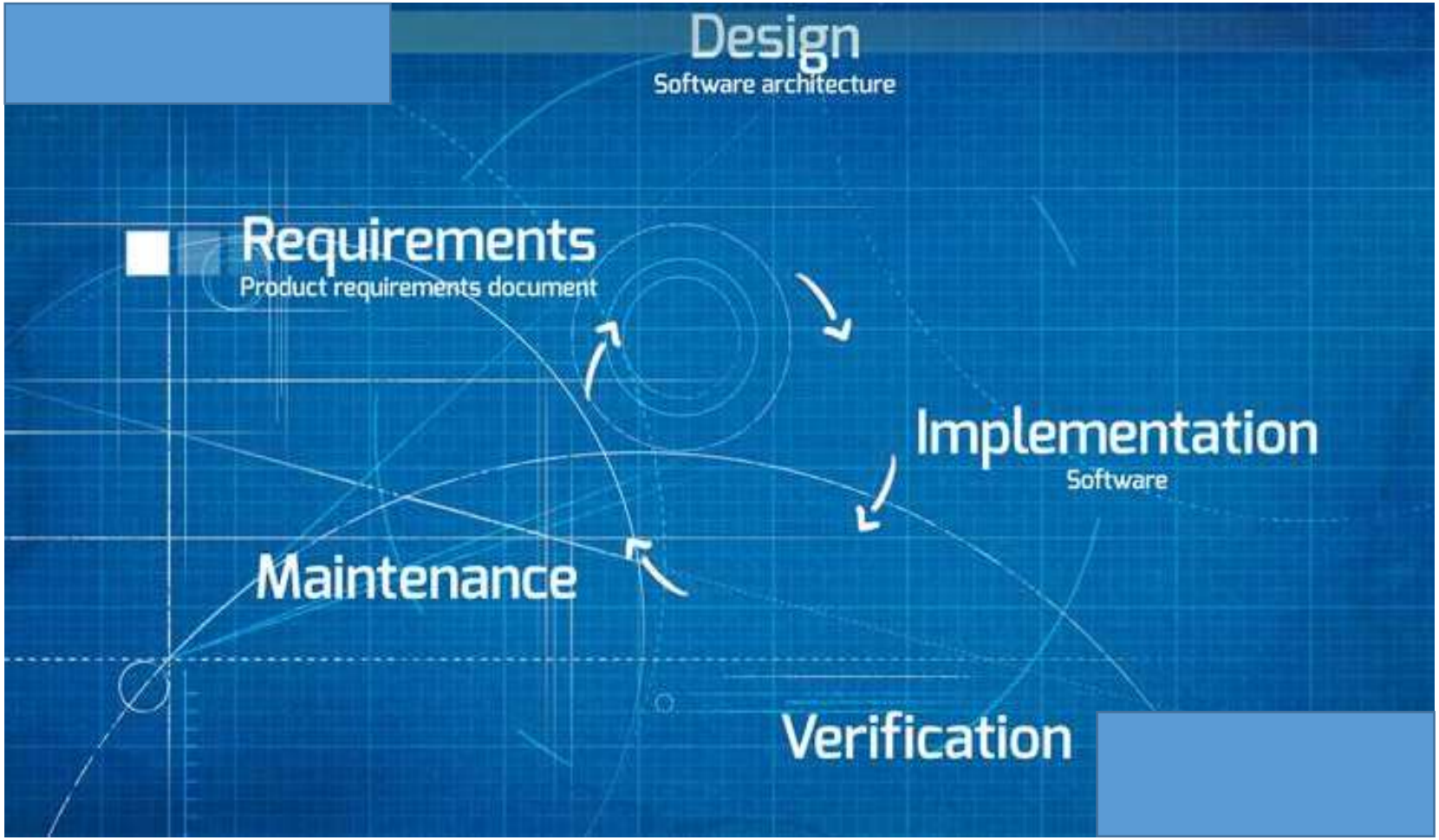
- It helps you meet mile stones on schedule and produce a finished project on time and within budget.
- Perhaps most important, software engineering gives us the flexibility to make changes to meet unexpected demands without completely affecting our schedule and budget constraints.

- The different steps that we need to take to keep a software engineering project on track.

- These are more or less the same for any large project although there are some important differences.

- They are:

1. Requirements Gathering (Requirements Analysis)
2. Design
3. Development (coding)
4. Testing
5. Deployment (Implementation)
6. Maintenance


Design
Software architecture
Requirements
Product requirements document
Implementation
Software
Maintenance
Verification

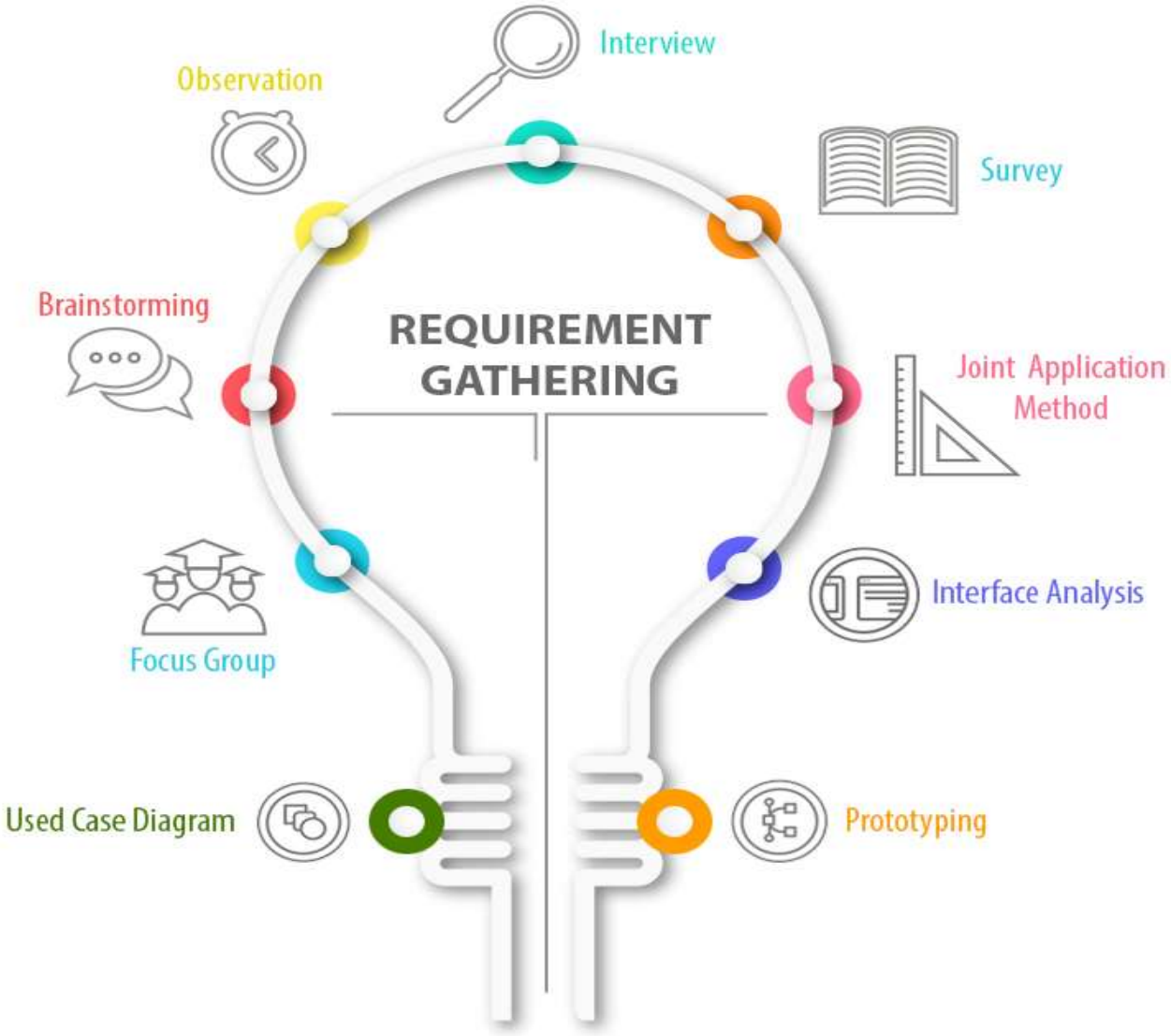
# Requirements Gathering

- No big project can succeed without a plan.
- Sometimes a project doesn’t follow the plan closely, but every big project must have a plan.
- The plan tells project members what they should be doing, when and how long they should be doing it, and most important what the project’s goals are.

- They give the project direction.
- One of the first steps in a software project is figuring out the requirements.
- We need to find out what the customers want and what the customers need.
- Depending on how well defined the user’s needs are, this can be time-consuming.

- Once the customers' wants and needs are clearly specified, then we can turn them into requirements documents.
- Those documents tell the customers what they will be getting, and they tell the project members what they will be building.
- Throughout the project, both customers and team members can refer to the requirements to see if the project is heading in the right direction.

- Requirements are the features that your application must provide.
- At the beginning of the project, we gather requirements from the customers to figure out what we need to build.
- Throughout development, we use the requirements to guide development and ensure that we are heading in the right direction.
- At the end of the project, we use the requirements to verify that the finished application actually does what it’s supposed to do.


Interview
Observation
Survey
Brainstorming
REQUIREMENT GATHERING
Joint Application Method
Interface Analysis
Focus Group
Used Case Diagram
Prototyping

# Characteristics of good requirements

- **Clear**
  - Good requirements are clear, concise, and easy to understand.
  - requirements cannot be vague or ill-defined.
  - Each requirement must state in concrete
- **Unambiguous (**not open to more than one interpretation)
  - As we write requirements, do our best to make sure we can't think of any way to interpret them other than the way we intend.

- **Consistent**
  - That means not only that they cannot contradict each other, but that they also don't provide so many constraints that the problem is unsolvable.
  - Each requirement must also be self-consistent.(it must be possible to achieve.)
- **Prioritized**
  - We might like to include every feature but don't have the time or budget, so something's got to go. At this point, we need to prioritize the requirements.
- **Verifiable**
  - If we can't verify a requirement, how do we know whether we have met it?
  - Being verifiable means the requirements must be limited and precisely defined.

**MOSCOW METHOD** - a common system for prioritizing application features.

- **M - Must.** These are required features that must be included. They are necessary for the project to be considered a success.
- **S - Should.** These are important features that should be included if possible.
- **C - Could.** These are desirable features that can be omitted if they won't fit in the schedule.
- **W - Won't.** These are completely optional features that the customers have agreed l will not be included in the current release.

# REQUIREMENT CATEGORIES

**Audience-Oriented Requirements:**

- These categories focus on different audiences and the different points of view that each audience has.

**Business Requirements:**

- Business requirements layout the project's high-level goals.

**User Requirements:**

- User requirements (which are also called stake holder requirements), describe how the project will be used by the end users.

**Functional Requirements:**

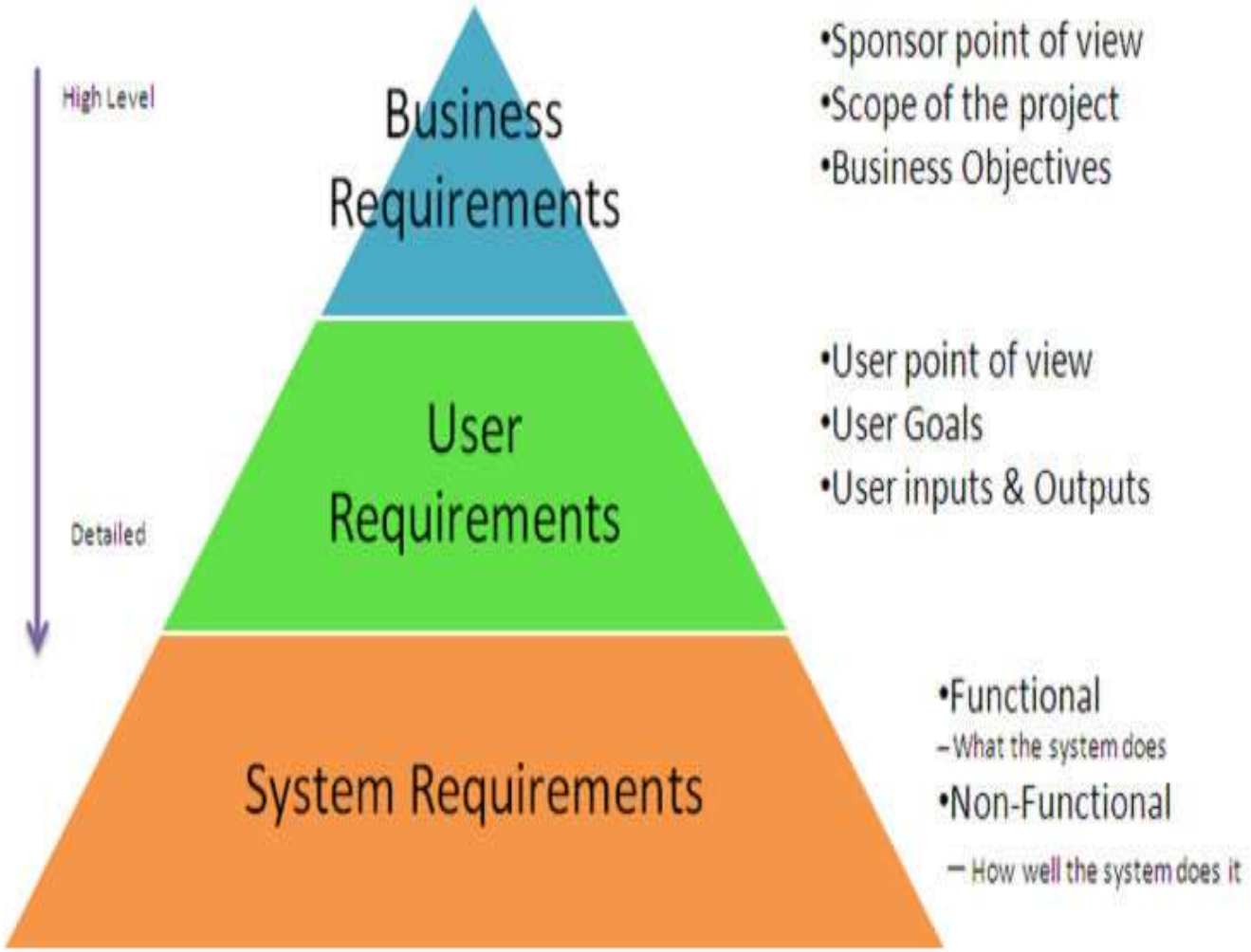
- Functional requirements are detailed statements of the project's desired capabilities.

**Non-functional Requirements:**

- Non-functional requirements are statements about the quality of the application's behavior or constraints on how it produces a desired result.
- They specify things such as the application's performance, reliability, and security characteristics.

**Implementation Requirements:**

- Implementation requirements are temporary features that are needed to transition to using the new system but that will be later discarded.


High Level
Detailed
Business Requirements
User Requirements
System Requirements
•Sponsor point of view
•Scope of the project
•Business Objectives
•User point of view
•User Goals
•User inputs & Outputs
•Functional
– What the system does
•Non-Functional
– How well the system does it

# Systems Development Life Cycle (SDLC)
# Life-Cycle Phases

**Initiation**

Begins when a sponsor identifies a need or an opportunity. Concept Proposal is created.

**System Concept Development**

Defines the scope or boundary of the concept. Includes Systems Boundary Document, Cost Benefit Analysis, Risk Management Plan and Feasibility Study.

**Planning**

Develops a Project Management Plan and other planning documents. Provides the basis for acquiring the resources needed to achieve a solution.

**Requirements Analysis**

Analyzes user needs and develops user requirements. Creates a detailed Functional Requirements Document.

**Design**

Transforms detailed requirements into complete, detailed System Design Document. Focuses on how to deliver the required functionality.

**Development**

Converts a design into a complete information system. Includes acquiring and installing systems environment; creating and testing databases/ preparing test case procedures; preparing test files; coding, compiling, refining programs; performing test readiness review and procurement activities.

**Integration and Test**

Demonstrates that the developed system conforms to requirements as specified in the Functional Requirements Document. Conducted by Quality Assurance staff and users. Produces Test Analysis Reports.

**Implementation**

Includes implementation preparation, implementation of the system into a production environment, and resolution of problems identified in the Integration and Test Phase.

**Operations and Maintenance**

Describes tasks to operate and maintain information systems in a production environment. includes Post-Implementation and In-Process Reviews.

**Disposition**

Describes end-of-system activities. emphasis is given to proper preservation of data.

# Design

- Software design sits at the technical kernel of software engineering
- The importance of design is ***quality***
- Design is the only way that you can accurately translate stakeholder's requirements into a finished software product or system.
- Software design is an iterative process through which requirements are translated into a "blueprint" for constructing the software

# Quality Attributes

- It is commonly known as **FURPS**.
  - *Functionality*
  - *Usability*
  - *Reliability*
  - *Performance*
  - *Supportability*

- ***Functionality*** is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system
- ***Usability*** is assessed by considering human factors
- ***Reliability*** is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the ability to recover from failure
- ***Performance*** is measured by considering processing speed, response time, resource consumption, throughput, and efficiency
- ***Supportability*** combines the ability to extend the program (extensibility), adaptability, serviceability—these three attributes represent a more common term, maintainability

- Design can be of **two types**:- high level design and low level design.

## HIGH-LEVEL DESIGN

- The high-level design includes such things as decisions about what platform to use (such as desktop, laptop, tablet, or phone), what data design to use and the project architecture at a relatively high level.
- We break the project into different modules that handle the project's major areas of functionality.
- We should make sure that the high-level design covers every aspect of the requirements.
- It should specify what the pieces(modules) do and how they should interact, but it should include as few details as possible about how the pieces do their jobs.

## LOW-LEVELDESIGN

- After high-level design breaks the project into pieces, we can assign those pieces to groups within the project so that they can work on low-level designs.
- The low-level design includes information about how that piece of the project should work.
- Better interactions between the different pieces of the project that may require changes here and there.

**High-level design focuses on what.**

**Low-level design begins to focus on how.**

## Three characteristics for a good design

- The design must implement all of the explicit requirements contained in the requirements model, and it must accommodate all of the implicit requirements desired by stakeholders.
- The design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.
- The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

- **Modular design** involves the decomposition of software behavior in encapsulated software units.
- **Modularity** is achieved by grouping together logically related elements, such as statements, procedures, variable declarations, object attributes, and so on in increasingly greater levels of detail
- The main objectives in seeking modularity are to foster **high cohesion and low coupling**

- **Cohesion** relates to the relationship of the elements of a module.
- Seven levels of cohesion in order of strength
  1. Coincidental — parts of the module are not related but are simply bundled into a single module.
  2. Logical — parts that perform similar tasks are put together in a module.
  3. Temporal — tasks that execute within the same time span are brought together.
  4. Procedural — the elements of a module make up a single control sequence.
  5. Communicational — all elements of a module act on the same area of a data structure.
  6. Sequential — the output of one part of a module serves as input for another part.
  7. Functional — each part of the module is necessary for the execution of a single function.

- High cohesion implies that each module represents a single part of the problem solution. Therefore, if the system ever needs modification, then the part that needs to be modified exists in a single place, making it easier to change.
- **Coupling** relates to the relationships between the modules themselves.
- There is great benefit in reducing coupling so that changes made to one code unit do not propagate to others (that is, they are hidden)
- This principle of “information hiding,” also known as Parnas partitioning, is the cornerstone of all software design.
- Low coupling limits the effects of errors in a module (lower “ripple effect”) and reduces the likelihood of data integrity problems.

- Coupling has also been characterized in increasing levels as follows:
  - **No direct coupling -** all modules are completely unrelated.
  - **Data coupling -** when two modules interact with each other by means of passing data
  - Stamp coupling - when a data structure is passed from one module to another, but that module operates on only some of the data elements of the structure. (When multiple modules share common data structure and work on different part of it.)
  - **Control coupling -** one module passes an element of control to another; that is, one module explicitly controls the logic of the other. (Two modules are called control-coupled if one of them decides the function of the other module or changes its flow of execution.)
  - **Common coupling** — if two modules both have access to the same global data.
  - **Content coupling** — one module directly references the contents of another (When a module can directly access or modify or refer to the content of another module)

# Development

- After we have created the high-and low-level designs, it’s time for the programmers to get to work.
- The programmers continue refining the low-level designs until they know how to implement those designs in code.
- As the programmers write the code, they test it to make sure it doesn’t contain any bugs.

- During the Development Phase, the system developer takes the detailed logical information documented in the previous phase and transforms it into machine-executable form, and ensures that all of the individual components of the automated system/application function correctly and interface properly with other components within the system/application.
- The Development Phase includes several activities that are the responsibility of the developer.

**Activities:-**

- The developer places the outputs under configuration control and performs change control.
- The developer also documents and resolves problems and non-conformances found in the software products and tasks.
- The developer selects, tailors, and uses those standards, methods, tools, and computer programming languages that are documented, appropriate, and established by the organization for performing the activities in the Development Phase.
- Plans for conducting the activities of the Development Phase are developed, documented and executed. The plans include specific standards, methods, tools, actions, and responsibility associated with the development and qualification of all requirements including safety and security.
- Verify that the software product covering the documented and baselined requirements in is a sufficient state of readiness for integration and formal testing by an assigned test group (i.e. other than development personnel.)
- During the Development Phase, the final Test Plan is prepared.

# Programming tips

- Be alert
- Write for people not for computer
- Comment first
- Write self documenting codes
- Keep it small
- Stay focused
- Avoid side effects
- Validate results
- Use exceptions
- Don't repeat code

- The best code starts out with a good design.
- A **code smell** refers to an indicator of poor design or coding.
- **Refactoring** refers to a behavior-preserving code transformation enacted to improve some feature of the software, which is evidenced by the code smell.

# Testing

- Effective software testing will improve software quality.
- Even poorly planned and executed testing will improve software quality if it finds defects.
- Testing is a life-cycle activity; testing activities begin from product inception and continue through delivery of the software and into maintenance.
- Collecting bug reports and assigning them for repair is also a testing activity.
- But as a life-cycle activity, the most valuable testing activities occur at the beginning of the project.

- Even if a particular piece of code is thoroughly tested and contains no (or few) bugs, there's no guarantee that it will work properly with the other parts of the system.
- One way to address the problems like this, is to perform different kinds of tests.
- First developers test their own code. Then testers who didn't write the code test it. After a piece of code seems to work properly, it is integrated into the rest of the project, and the whole thing is tested to see if the new code broke anything.

The terms error, bug, fault, and failure:-

- Use of "**bug**" is that an error crept into the program through no one's action. The preferred term for an error in requirement, design, or code is "**error" or "defect**."
- The manifestation of a defect during the operation of the software system is called a **fault**.
- A fault that causes the software system to fail to meet one of its requirements is called a **failure**.

- **Verification**, or testing, determines whether the products of a given phase of the software development cycle fulfill the requirements established during the previous phase.
- **Verification answers the question "Am I building the product right?"**
- **Validation** determines the correctness of the final program or software with respect to the user's needs and requirements.
- **Validation answers the question "Am I building the right product?"**

# Purpose of Testing

- Testing is the execution of a program or partial program with known inputs and outputs that are both predicted and observed for the purpose of finding faults or deviations from the requirements.
- Testing will **flush out errors**, this is just one of its purposes.
- The other is to **increase trust in the system**.
- Testing must increase faith in the system, even though it may still contain undetected faults, by ensuring that the software meets its requirements.
- A **good test** is one that has a high probability of finding an error. A **successful test** is one that uncovers an error.

# Basic principles of software testing

- These are the most helpful and practical rules for the tester.
  - ➢All tests should be traceable to customer requirements.
  - ➢Tests should be planned long before testing begins.
  - ➢Remember that the Pareto principle applies to software testing.

    (The Pareto principle states that for many outcomes roughly 80% of consequences come from 20% of the causes.)
  - ➢Testing should begin “in the small” and progress toward testing “in the large.”
  - ➢Exhaustive testing is not practical.
  - ➢To be most effective, testing should be conducted by an independent party.

- Testing is a well-planned activity and should not be conducted willy nilly, nor undertaken at the last minute, just as the code is being integrated.
- The most important activity that the test engineer can conduct during requirements engineering is to ensure that each requirement is testable.
- A requirement that cannot be tested cannot be guaranteed and, therefore, must be reworked or eliminated.
- Wide range of testing techniques for unit testing, integration testing, and system level testing.
- Any one of these test techniques can be either insufficient or computationally unfeasible. Therefore, some combination of testing techniques is almost always employed.

# Who Tests the Software?

*developer*

Understands the system

but, will test "gently"

and, is driven by "delivery"

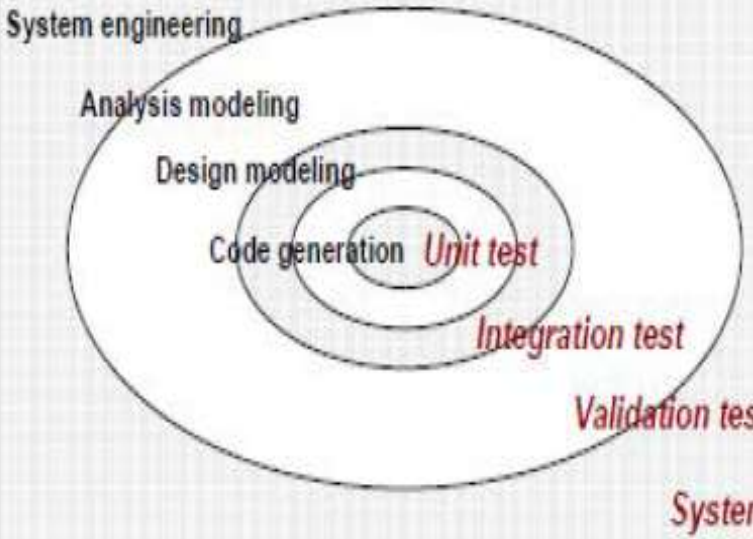
*independent tester*

Must learn about the system,

but, will attempt to break it

and, is driven by quality

# Testing Strategy



# Levels of Testing ( Types of Testing )

- **Unit testing:-** Verifies the correctness of a specific piece of code
  - Since unit test is the first chance to catch errors, it is extremely important
  - Focus on each unit (eg: component, class or web App content object) of the software as implemented in the source code.
  - sometimes called **desk checking.**

# Black Box Testing

- In black box testing, only inputs and outputs of the unit are considered; how the outputs are generated based on a particular set of inputs is ignored.
- Some widely used black box testing techniques include:
  - Exhaustive testing
  - Boundary value testing
  - Random test generation
  - Worst case testing

- **Brute force or exhaustive testing** involves presenting each code unit with every possible input combination.
  - Brute force testing can work well in the case of a small number of inputs each with a limited input range
- **Boundary value or corner case testing** solves the problem of combinatorial explosion by testing some very tiny subset of the input combinations identified as meaningful “boundaries” of input.

- **Random test case generation, or statistically based testing**, can be used for both unit and system level testing.
  - This kind of testing involves subjecting the code unit to many randomly generated test cases over some period of time.
  - The purpose of this approach is to simulate execution of the software under realistic conditions.

- **Equivalence class testing** involves partitioning the space of possible test inputs to a code unit or group of code units into a set of representative inputs.
- **Example:-**

  Auto manufacturers don’t have a crash test dummy representing every possible human being. Instead, they use a handful of representative dummies — small, average, and large adult males; small, average, and large adult females; pregnant female; toddler, etc. These categories represent the equivalence classes.

**Disadvantages to black box testing :-**

- One disadvantage is that it can bypass unreachable or dead code.
- In other words, black box testing only tests what is expected to happen, not what wasn’t intended.
- **White box or clear box testing** techniques can be used to deal with this problem.

# White Box Testing

- White box testing (sometimes called **clear or glass box testing**) seeks to test the structure of the underlying code. For this reason it is also called **structural testing**.
- Black box tests are data driven, White box tests are logic driven.
- White box testing are designed to exercise all paths in the code unit.
- White box testing also has the advantage that it can discover those code paths that cannot be executed.

- The following white box testing strategies
  - ✓ DD path testing
  - ✓ DU path testing
  - ✓ McCabe's basis path method
  - ✓ Code inspections
  - ✓ Formal program proving

- **DD path testing, or decision-to-decision path testing** is based on the control structure of the program.
- In DD testing, a graph representation of the control structure of the program is used to generate test cases that traverse the graph, essentially from one decision branch (for example, if-then statement) to another in well-defined ways.
- Depending on the strength of the testing, different combinations of paths are tested.

- **DU (define-use) path testing** is a data-driven white box testing technique that involves the construction of test cases that exercise all possible definition, change, or use of a variable through the software system.
- DU path testing is actually rather sophisticated because there is a hierarchy of paths involving whether the variable is observed or changed.

- **McCabe's basis path method -** McCabe's metric to determine the complexity of code.
- McCabe's metric can also be used to determine a lower bound on the number of test cases needed to traverse all linearly independent paths in a unit of code.

- McCabe also provides a procedure for determining the linearly independent paths by traversing the program graph.
- This technique is called the **basis path method**
- The basis path method begins with the selection of a baseline path, which should correspond to some “ordinary” case of program execution along one of the programs.
- McCabe advises choosing a path with as many decision nodes as possible.

- **Code Inspections:-** In code inspections, the author of some collection of software presents each line of code to a review group of peer software engineers.
- Code inspections can detect errors as well as discover ways for improving the implementation.
- This audit also provides an opportunity to enforce coding standards.

- **Formal Program Proving :-** Formal program proving is a kind of white box testing using mathematical techniques in which the code is treated as a theorem and some form of calculus is used to prove that the program is correct.
- This form of verification requires a high level of training and is useful, generally, for only limited purposes because of the intensity of activity required.

## Integration Testing:-

- Integration testing involves testing of groups of components integrated to create a system or sub-system.
- The tests are derived from the system specification.
- Verifies that a block of code is working properly
- It checks that the existing code calls the new method correctly, and that new method can call other methods correctly.

- **Incremental Integration Testing:-** This is a strategy that partitions the system in some way to reduce the code tested. Incremental testing strategy includes:
  - **Top-Down Testing**
  - **Bottom-Up Testing**
  - **Other kinds of system partitioning**

  In practice, most integration involves a combination of these strategies.

- **Top-Down Testing** - This kind of testing starts with a high-level system and integrates from the topdown, replacing individual components by stubs (dummy programs) where appropriate.
- **Bottom-Up Testing** - Bottom up testing is the reverse of top-down testing in which we integrate individual components in levels, from the bottom up, until the complete system is created.
- **Other kinds of system partitioning** - These include pair-wise integration, sandwich integration, neighbourhood,integration testing, and interface testing.

## Other kinds of system partitioning

- **Pair-wise integration testing -** pairs of modules or functionality are tested together.
- **Sandwich integration -** combination of top-down and bottom up testing, which falls somewhere in between big-bang testing (one big test case) and testing of individual modules.
- **Neighborhood integration testing -** portions of program functionality that are logically connected somehow are tested in groups.
- **Interface testing** - takes place when modules or subsystems are integrated to create larger systems.

- **Component interface testing:-** Studies the interactions between components
  - ➢Common strategy for component level testing is to think of the interaction between components as one component sending a message to another.
- **Stress testing:** executes a system in a manner that demands resources in abnormal quantity, frequency, or volume.
  - the system is subjected to a large disturbance in the inputs;
  - to see how the system fails (gracefully or catastrophically).
  - with cases and conditions where the system is under heavy load

- **Burn-in testing** is a type of system-level testing done in the factory, which seeks to flush out those failures appearing early in the life of the system, and thus to improve the reliability of the delivered product.
- **Cleanroom testing** is more than a kind of system testing. It is really a testing philosophy.
  - The principal tenant of cleanroom software development is that **given sufficient time and care, error-free software can be written.**
  - Cleanroom software development relies heavily on code inspections and formal program validation.

- **Regression testing:-** Test the program's entire functionality to see if any thing changed when you added new code to the project.
  - Regression testing (which can also be performed at the unit level) is used to validate the updated software against the old set of test cases that have already been passed.
  - Regression testing helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.

- **Acceptance testing:-** Determine whether the finished application meets the customer's requirements.
  - Usually customer or representative sits down with the application and runs through all the user cases (during requirement analysis) to make sure everything works as advertised.
  - Most software builders use a process called alpha and beta testing to uncover errors that only the end user able to find.

- **Alpha test:-**
  - First round testing by selected customers or independent testers.
  - version of the complete software is tested by customer under the supervision of the developer at the developer's site
- **Beta test:-**
  - Second round testing after alpha test.
  - version of the complete software is tested by customer at his or her own site without the developer being present.

- **Software Fault Injection :-** Fault injection is a form of dynamic software testing that acts like “crashtesting” the software by demonstrating the consequences of incorrect code or data.
  - The main benefit of fault injection testing is that it can demonstrate that the software is unlikely to do what it shouldn’t.
  - Eg: type a letter when the input called is for a number
- **Security testing:** verifies that protection mechanisms built into a system will, in fact, protect it from improper penetration.

- **Automated testing:-** Automated testing tools let you define tests and the results they should produce.
  - ➢ After running a test, the testing tool can compare the results it got with expected results.
- **System testing:-** End to end run through of the whole system.
  - ➢ System test exercise every part of the system to discover as many bugs as possible.

- **Performance Testing:** test the run-time performance of software within the context of an integrated system
- **Deployment testing(configuration testing):** exercises the software in each environment in which it is to operate. It also examines the installation procedures and installation software.
- **Recovery testing:** forces the software to fail in a variety of ways and verifies that recovery is properly performed.

- **Compatibility test:-** Focus on compatibility with different environments such as computers running older operating systems versions. Also with older versions of files, databases etc…
- **Destructive test :-** Makes the application fail so that you can study its behaviour when the worst happens.
- **Installation test:-** Make sure that you can successfully install the system on a fresh computer

- **Functional test:-** Deals with features the application provides.
- **Exhaustive testing :-** Testing a method with every possible input.
  - ➢ It proves that a method works correctly under all circumstances, so it's the best you can possibly do.
  - ➢ Because most methods take too many possible inputs, it won't work most of the time.
- **Accessibility test:-** Tests the application for accessibility by those with visual, hearing or other impairments.

**When should you stop testing?**

- There are several criteria that can be used to determine when testing should cease.

1. When you run out of time.
2. When continued testing causes no new failures.
3. When continued testing reveals no new faults.
4. When you can’t think of any new test cases.
5. When you reach a point of “diminishing returns.”
6. When mandated coverage has been attained.
7. When all faults have been removed

# Deployment

- What is deployment?
  - Getting software out of the hands of the developers into the hands of the users.
- The software is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation.

# Key issues around deployment

1. **Business processes:-** Most large software systems require the customer to change the way they work.
2. **Training:-** No point in deploying software if the customers can't use it.
3. **Deployment itself:-** How physically to get the software installed.
4. **Equipment:-** Is the customer's hardware up to the job?
5. **Expertise:-** Does the customer have the IT expertise to install the software?
6. **Integration:-** can it be integrated with other systems of the customer.

# Maintenance

- The software product will *enter a maintenance mode after delivery* in which it will experience many recurring life cycles as errors are detected and corrected and features are added.
- The process of changing a system after it has been delivered.
- Software maintenance is the "…correction of errors, and implementation of modifications needed to allow an existing system to perform new tasks, and to perform old ones under new conditions"

- As soon as the users start pounding away on our software, they will find bugs.
- Of course, when the users find bugs, we need to fix them.
- Fixing a bug sometimes leads to another bug, so now we get to fix that one as well.
- The software maintenance phase activities generally consist of a series of reengineering processes to prolong the life of the system.

- **Re-engineering:-** is the process of taking an old or un-maintainable system and transforming it until it's maintainable.
- If our application is successful, users will use it a lot, and they'll be even more likely to find bugs. They also think up of enhancements, improvements, and new features that they want added immediately.

- Generally maintenance tasks are grouped into the following four categories: (3 types of maintenance)

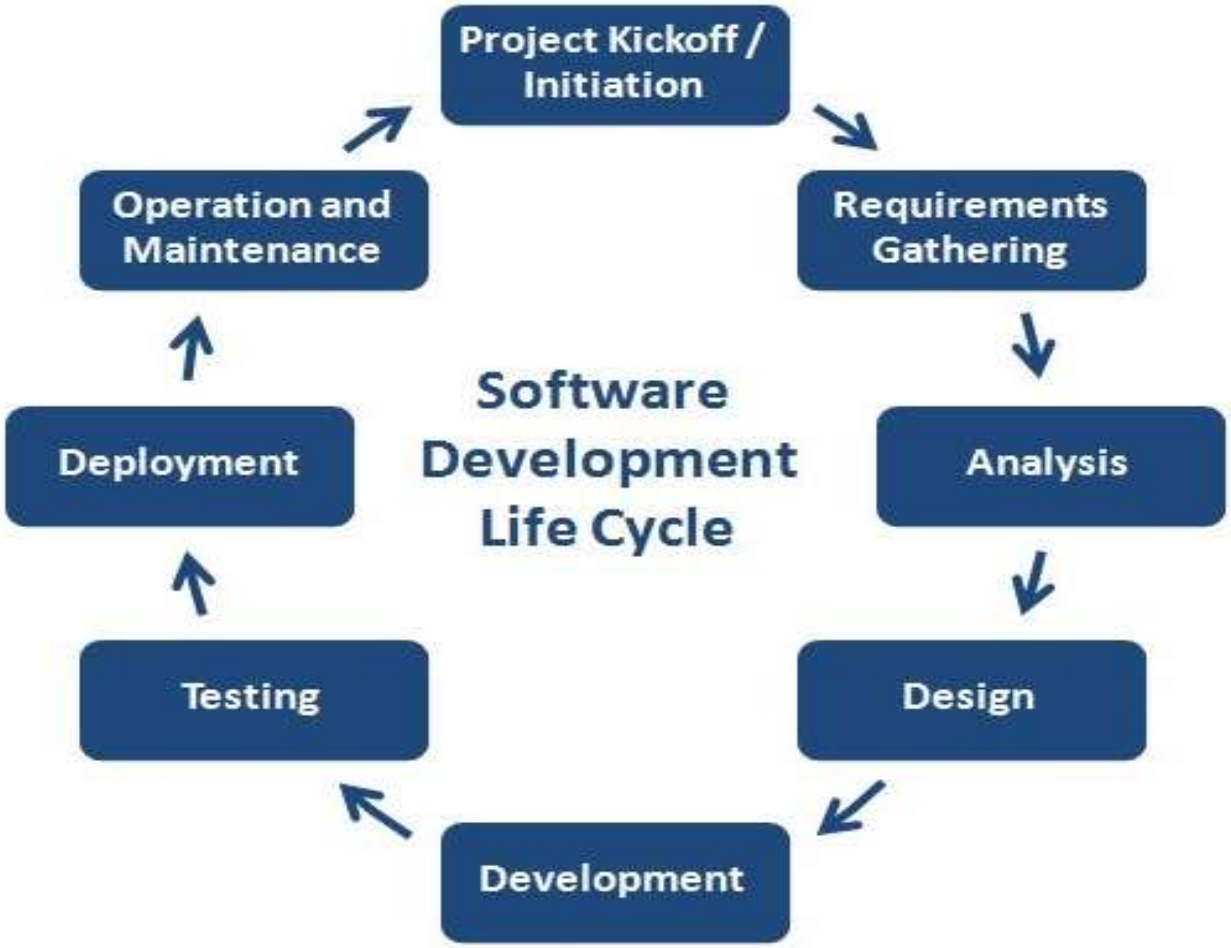
➢ **Corrective—** changes that involve maintenance to correct errors (Fixing bugs)

➢ **Perfective —** all other maintenance including enhancements, documentation changes, efficiency improvements, and so on. (Improving existing features and adding new ones)

➢**Adaptive—** changes that result from external changes to which the system must respond. (Modifying the application to meet changes in the application's environment )

➢ **Preventive—** Restructuring the code to make it more maintainable.

- **Fixing bugs and vulnerabilities:-** not only in code, but also design and requirements
- **Adapting to new platforms and software environments:-** e.g. new hardware, new OS, new support software
- **Supporting new features and requirements:-** necessary as operating environments change and in response to competitive pressures


Project Kickoff / Initiation
Requirements Gathering
Analysis
Design
Development
Testing
Deployment
Operation and Maintenance
Software Development Life Cycle

# Module - I

**PLANNING PHASE:–**

- **Project planning objective**
- **Scope of software system**
- **Empirical estimation models**
- **COCOMO**
- **Staffing and personal planning**

# Planning Phase

**Planning Objective:-**

- The objective of software project planning is to provide a framework that enables the manager to make reasonable estimates of resources, cost, and schedule.

- Estimates should attempt to define best-case and worst-case scenarios so that project outcomes can be bounded.

- The plan must be adapted and updated as the project proceeds.

# Software Scope

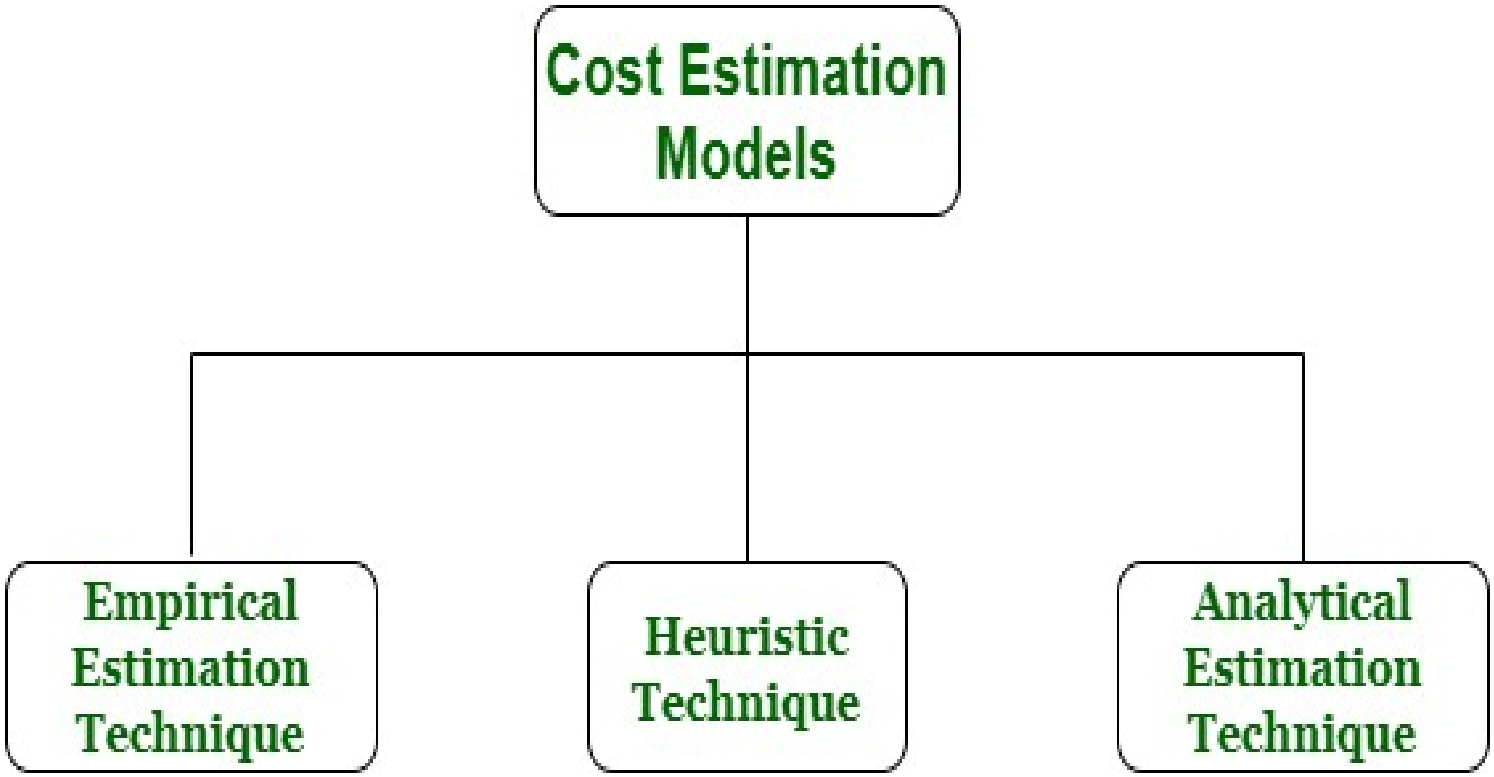
- *Software scope describes the functions and features that are to be delivered to end users*
- The data that are input and output "content" that is presented to users as a consequence of using the software.
- The performance, constraints, interfaces, and reliability that *bound the system*

- Scope is defined using one of two techniques
- A narrative description of software scope is developed after communication with all stakeholders.
- A set of use cases is developed by end users
- Functions described in scope is evaluated for estimation

# Cost Estimation Models in Software Engineering

- **Cost estimation** simply means a technique that is used to find out the cost estimates.
- The cost estimate is the financial spend that is done on the efforts to develop and test software in software engineering.

Various techniques or models are available for cost estimation, also known as Cost Estimation Models as shown below :



- **Empirical estimation** is a technique or model in which empirically derived formulas are used for predicting the data that are a required and essential part of the software project planning step.
- These techniques are usually based on the data that is collected previously from a project and also based on some guesses, prior experience with the development of similar types of projects, and assumptions.
- It uses the size of the software to estimate the effort.
- In this technique, an educated guess of project parameters is made.

- The **heuristic technique** is a technique or model that is used for solving problems, learning, or discovery in the practical methods which are used for achieving immediate goals.
- These techniques are flexible and simple for taking quick decisions through shortcuts and good enough calculations, most probably when working with complex data.
- In this technique, the relationship among different project parameters is expressed using mathematical equations.
- The popular heuristic technique is given by **Constructive Cost Model (COCOMO)**. This technique is also used to increase or speed up the analysis and investment decisions.

- **Analytical estimation** is a type of technique that is used to measure work.
- In this technique, firstly the task is divided or broken down into its basic component operations or elements for analyzing. Second, if the standard time is available from some other source, then these sources are applied to each element or component of work.Third, if there is no such time available, then the work is estimated based on the experience of the work.  Results are derived by making certain basic assumptions about the project.

# What is COCOMO estimation model?

- The **Constructive Cost Model** (COCOMO) is a procedural software cost estimation model developed by Barry W. Boehm.
- Three versions of COCOMO: Basic, Intermediate, and Detailed.
- The model parameters are derived from fitting a regression formula using data from historical (past) projects

- **COCOMO** is a regression model based on LOC, i.e number of Lines of Code.
- It has been commonly used to project costs for a variety of projects and business processes.
- It is a procedural cost estimate model for software projects and often used as a process of reliably predicting the various parameters associated with making a project such as size, effort, cost, time and quality.

- The original COCOMO model was a set of models
- 3 levels (basic, intermediate, and advanced).
- 3 development modes (organic, semi-detached, and embedded)

# COCOMO Model Levels

- **Basic -** predicted software size (lines of code) was used to estimate development effort
- **Intermediate -** predicted software size (lines of code), plus a set of 15 subjectively assessed 'cost drivers' was used to estimate development effort
- **Advanced -** on top of the intermediate model, the advanced model allows phase-based cost driver adjustments and some adjustments at the module, component, and system level

# 15 cost drivers

| COST DRIVER | DESCRIPTION |
|---|---|
| RELY | Required software reliability |
| DATA | Database size |
| CPLX | Product complexity |
| TIME | Execution time constraints |
| STOR | Main storage constraints |
| VIRT | Virtual machine volatility - degree to which the operating system changes |
| TURN | Computer turn around time |
| ACAP | Analyst capability |
| AEXP | Application experience |
| PCAP | Programmer capability |
| VEXP | Virtual machine (i.e. operating system) experience |
| LEXP | Programming language experience |
| MODP | Use of modern programming practices |
| TOOL | Use of software tools |
| SCED | Required development schedule |

## COCOMO - COST DRIVERS

| | COST DRIVER | RATING | | | | | |
|---|---|---|---|---|---|---|---|
| | | V.LOW | LOW | NOMINAL | HIGH | V.HIGH | EX. HIGH |
| (PRODUCT) | RELY | 0.75 | 0.88 | 1.00 | 1.15 | 1.40 | . |
| .. | DATA | . | 0.94 | 1.00 | 1.08 | 1.16 | . |
| .. | CPLX | 0.70 | 0.85 | 1.00 | 1.15 | 1.30 | 1.65 |
| (COMPUTER) | TIME | . | . | 1.00 | 1.11 | 1.30 | 1.66 |
| .. | STOR | . | . | 1.00 | 1.06 | 1.21 | 1.56 |
| .. | VIRT | . | 0.87 | 1.00 | 1.15 | 1.30 | . |
| .. | TURN | . | 0.87 | 1.00 | 1.07 | 1.15 | . |
| (PERSONNEL) | ACAP | 1.46 | 1.19 | 1.00 | 0.86 | 0.71 | . |
| .. | AEXP | 1.29 | 1.13 | 1.00 | 0.91 | 0.82 | . |
| .. | PCAP | 1.42 | 1.17 | 1.00 | 0.86 | 0.70 | . |
| .. | VEXP | 1.21 | 1.10 | 1.00 | 0.90 | . | . |
| .. | LEXP | 1.14 | 1.07 | 1.00 | 0.95 | . | . |
| (PROJECT) | MODP | 1.24 | 1.10 | 1.00 | 0.91 | 0.82 | . |
| .. | TOOL | 1.24 | 1.10 | 1.00 | 0.91 | 0.83 | . |
| .. | SCED | 1.23 | 1.08 | 1.00 | 1.04 | 1.10 | . |

# COCOMO Development Modes

- **Simple(Organic)** - small relatively small, simple software projects in which small teams with good application experience work to a set of flexible requirements
- **Embedded** - the software project has tight software, hardware and operational constraints

- **Semi-detached -** an intermediate (in size and complexity) software project in which teams with mixed experience levels must meet a mix of rigid and less than rigid requirements
- The general formula of the basic COCOMO model is:

$$E = a(S)^b$$

Where:

**E** represents effort in person-months,
**S** is the size of the software development in KLOC and,
**a** and **b** are values dependent on the development mode,

| development mode: | | | |
|---|---|---|---|
| | organic | **a** = 2.4 | **b** = 1.05 |
| | semi-detached | **a** = 3.0 | **b** = 1.12 |
| | embedded | **a** = 3.6 | **b** = 1.20 |

- The intermediate and advanced COCOMO models incorporate 15 'cost drivers'.
- These 'drivers' multiply the effort derived for the basic COCOMO model
- The importance of each driver is assessed and the corresponding value multiplied into the COCOMO equation

$$E = a(S)^b \times \text{product(cost drivers)}$$

As an example of how the intermediate COCOMO model works, the following is a calculation of the estimated effort for a semi-detached project of 56 KLOC. The cost drivers are set as follows:

| | | | |
|---|---|---|---|
| Product cost drivers (from the table) set | **high** | = 1.15 x 1.08 x 1.15 | = 1.43 |
| Computer cost drivers (from the table) set | **nominal** | = 1.00 | |
| Personnel cost drivers (from the table) set | **low** | = 1.19 x 1.13 x 1.17 x 1.10 x 1.07 | = 1.85 |
| Project cost drivers (from the table) set | **high** | = 0.91 x 0.91 x 1.04 | = 0.86 |

hence, **product(cost drivers)** = 1.43 x 1.00 x 1.85 x 0.86 = 2.28

for a semi-detached project of 56KLOC: **a** = 3.0 **b** = 1.12 **S** = 56

$$\mathbf{E} = \mathbf{a(S)b \times product(cost\ drivers)}$$

$$\mathbf{E} = 3.0 \times (56)^{1.12} \times 2.28$$

$$\mathbf{E} = 3.0 \times 90.78 \times 2.28$$

$$\mathbf{E} = 620.94 \text{ person-months}$$

# The COCOMO II Model

- The original COCOMO model became one of the most widely used and discussed software cost estimation models in the industry.
- It has evolved into a more comprehensive estimation model, called COCOMOII.
- COCOMO II is actually a hierarchy of estimation models that address the following areas:

- *Application composition model.* Used during the early stages of software engineering, when prototyping of user interfaces, consideration of software and system interaction, assessment of performance, and evaluation of technology maturity are paramount.
- *Early design stage model.* Used once requirements have been stabilized and basic software architecture has been established.
- *Post-architecture-stage model.* Used during the construction of the software.

- Like all estimation models for software, the COCOMO II models require sizing information.
- Three different sizing options are available as part of the model hierarchy:

1. object points,
2. function points, and
3. lines of source code.

- The object point is an indirect software measure that is computed using counts of the number of

  1. screens (at the user interface),
  2. reports,
  3. components likely to be required to build the application.

- Each object instance (e.g., a screen or report) is classified into one of three complexity levels (i.e., simple, medium, or difficult).
- Complexity is a function of the number and source of the client and server data tables that are required to generate the screen or report and the number of views or sections presented as part of the screen or report.
- Once complexity is determined, the number of screens, reports, and components are weighted.

- The object point count is then determined by multiplying the original number of object instances by the weighting factor in the figure and summing to obtain a total object point count.
- When component-based development or general software reuse is to be applied, the percent of reuse (%reuse) is estimated and the object point count is adjusted

$$\text{NOP} = (\text{object points}) \times [(100 - \%\text{reuse})/100]$$

where NOP is defined as new object points.

# Software Project Staffing

- All the management activities that involve filling and keeping filled the positions that were established in the project organizational structure by well-qualified personnel.

# Major Issues in Staffing

1. Project managers are frequently selected for their ability to program or perform engineering tasks rather than their ability to manage
2. The productivity of programmers, analysts, and software engineers varies greatly from individual to individuals.
3. Universities are not producing a sufficient number of computer science graduates who understand the software engineering process or project management.
4. Training plans for individual software developers are not developed or maintained

# Staffing Activities for Software Projects

| Activity | Definition or Explanation |
|---|---|
| Fill organizational positions | Select, recruit, or promote qualified people for each project position. |
| Assimilate newly assigned personnel | Orient and familiarize new people with the organization, facilities, and tasks to be done on the project. |
| Educate or train personnel | Make up deficiencies in position qualifications through training and education. |
| Provide for general development | Improve knowledge, attitudes, and skills of project personnel. |
| Evaluate and appraise personnel | Record and analyze the quantity and quality of project work as the basis for personnel evaluations. Set performance goals and appraise personnel periodically. |
| Compensate | Provide wages, bonuses, benefits, or other financial remuneration commensurate with project responsibilities and performance. |

# Factors to Consider when Staffing

1. Education
2. Experience
3. Training
4. Motivation

1. Commitment
2. Self-Motivation
3. Group Affinity
4. Intelligence

**Education:**

- Does the candidate have the minimum level of education for the job? Does the candidate have the proper education for future growth in the company?

**Experience:**

- Does the candidate have an acceptable level of experience? Is it the right type and variety of experience?

**Training:**

- Is the candidate trained in the language, methodology, and equipment to be used, and the application area of the software system?

**Motivation:**

- Is the candidate motivated to do the job, work for the project, work for the company, and take on the assignment?

**Commitment:**

- Will the candidate demonstrate loyalty to the project, to the company, and to the decisions made?

**Self-motivation:**

- Is the candidate a self-starter, willing to carry a task through to the end without excessive direction?

**Group affinity:**

- Does the candidate fit in with the current staff? Are there potential conflicts that need to be resolved?

**Intelligence:**

- Does the candidate have the capability to learn, to take difficult assignments, and adapt to changing environments?

# Sources of Qualified Project Individuals

1. Transferring personnel from within the project itself, from task to another task
2. Transfers from other projects within the organization, when other project has ended or is being cancelled
3. New hires from other companies through such methods as job fairs, referrals

4. New college graduates can be recruited either through interviews on campus or through referrals from recent graduates who are now company employees
5. If the project manager is unable to obtain qualified individuals to fill positions, one option is to hire unqualified but motivated individuals and train them for those vacancies

# Personnel Planning

- This includes estimation or allocation of right persons or individuals for the right type of tasks of which they are capable
- The capability of the individuals should always match with the overall objective of the project

# Module I - **Software Engineering Models**

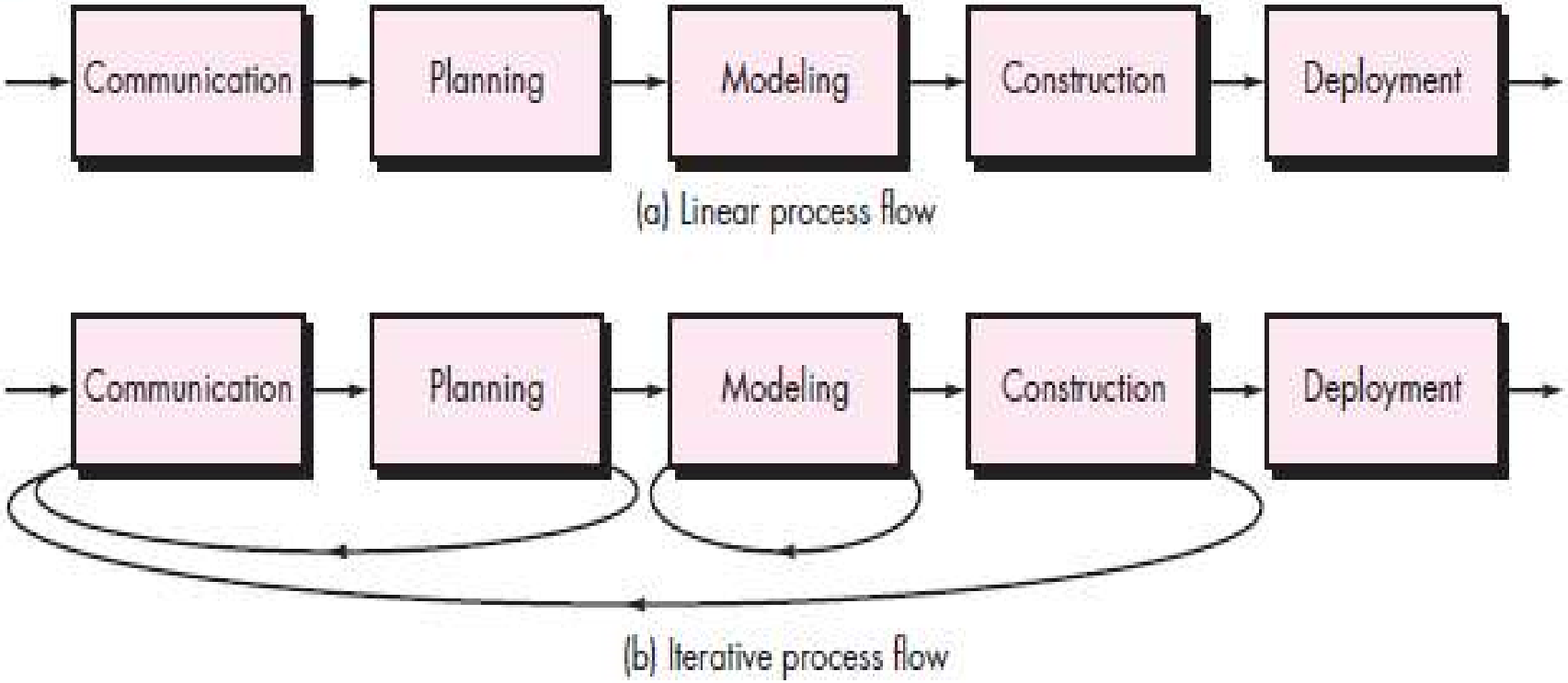
- Predictive software engineering models
- Model Approaches
- Prerequisites
- Predictive and Adaptive
  - waterfall
  - waterfall with feedback
  - Sashimi
  - Incremental waterfall
  - V model
- Prototyping and prototyping models.

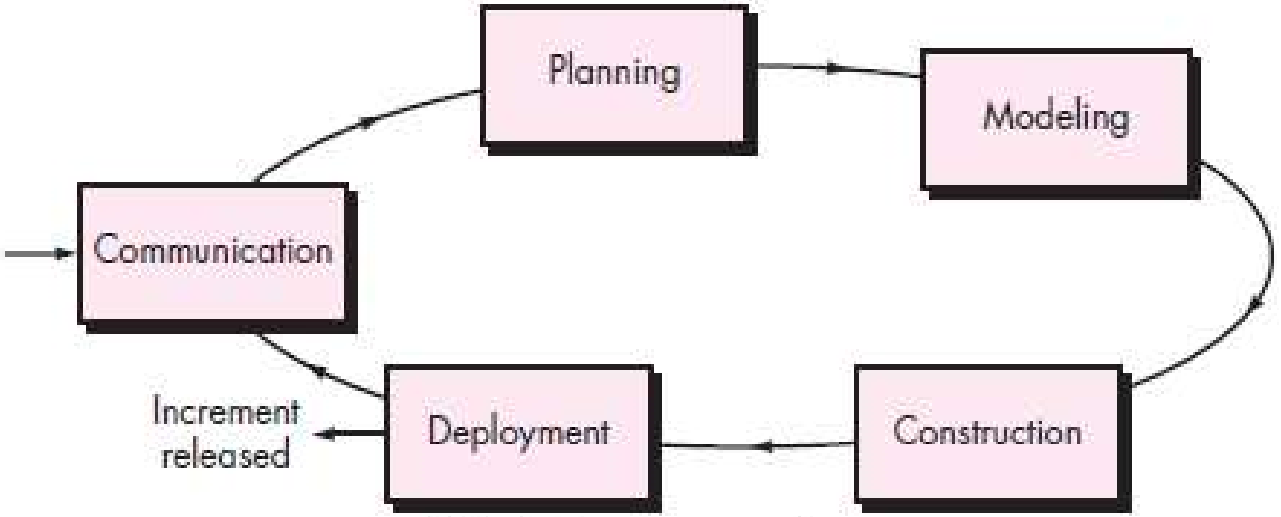
# Introduction – Process Model

- A software process as a framework for the activities, actions, and tasks that are required to build high-quality software.
- A generic process framework for software engineering defines five framework activities—communication, planning, modeling, construction, and deployment.
- In addition, a set of umbrella activities—project tracking and control, risk management, quality assurance, configuration management, technical reviews, and others—are applied throughout the process.

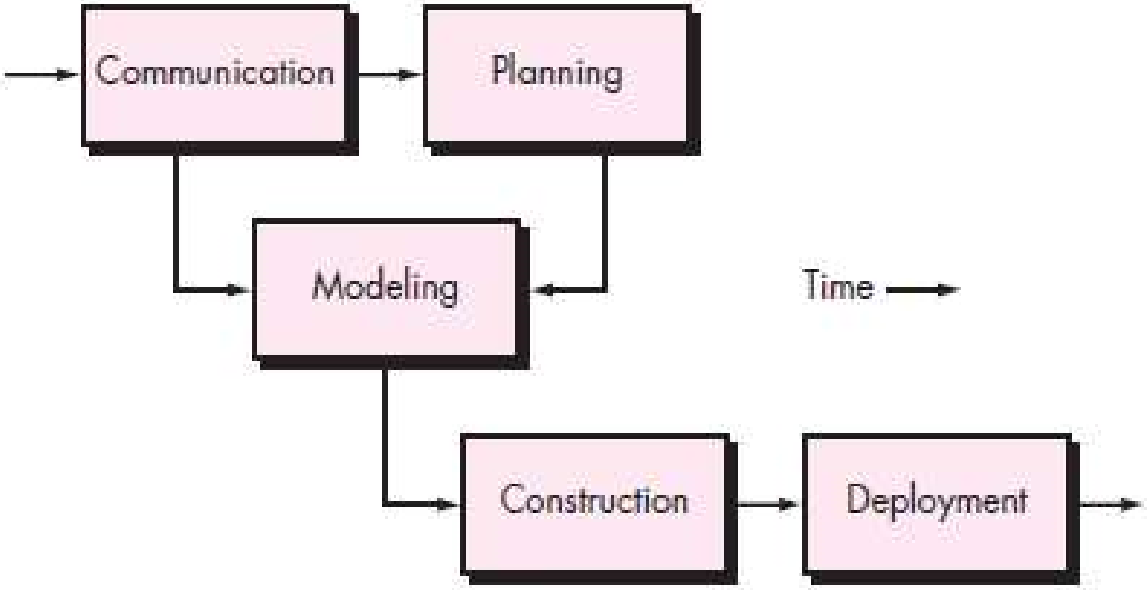
- ***Process flow*** — describes how the framework activities and the actions and tasks that occur within each framework activity are organized with respect to sequence and time

**FIGURE 2.2** Process flow






(c) Evolutionary process flow




(d) Parallel process flow

# Model Approaches

- There are a lot of different development models.
- Each model has its own set of rules, philosophy and list of important principles.

- ***Prescriptive process models*** have been applied for many years in an effort to bring order and structure to software development.
- Each of these models suggests a somewhat different process flow, but all perform the same set of generic framework activities: communication, planning, modeling, construction, and deployment.

- **Prescriptive process models define a prescribed set of process elements and a predictable process work flow.**
- "Prescriptive " because they prescribe a set of process elements—framework activities, software engineering actions, tasks, work products, quality assurance, and change control mechanisms for each project.

- **Sequential process models**, such as the waterfall and V models, are the oldest software engineering paradigms.
- They suggest a linear process flow that is often inconsistent with modern realities (e.g., continuous change, evolving systems, tight time lines) in the software world. However, they have applicability in situations where requirements are well defined and stable

- ***Incremental process models*** are iterative in nature and produce working versions of software quite rapidly.
- ***Evolutionary process models*** recognize the iterative, incremental nature of most software engineering projects and are designed to accommodate change.
- Evolutionary models, such as prototyping and the spiral model, produce incremental work products (or working versions of the software) quickly.

- These models can be adopted to apply across all software engineering activities— from concept development to long-term system maintenance.
- **The concurrent process model** allows a software team to represent iterative and concurrent elements of any process model.
- **The Unified Process** is a “use case driven, architecture-centric, iterative and incremental” software process designed as a framework for UML methods and tools.

- **Agile methods -** allow a project's goal to change over time to track changing customer needs.

- **Each model has its own set of rules, philosophy and list of important principles.**

# Prerequisites

- Before we start using a particular model, every one in the team has to understand the model properly.
- Every one needs to agree on what the rules are and what procedures we will use to make sure the rules are followed.

# PREDICTIVE AND ADAPTIVE

# ADAPTIVE DEVELOPMENT MODEL

- Enables you to change the project's goals if necessary during the development.
- Periodically revaluate and decide whether need to change the direction.
- Even then, we cannot say that an adaptive model is always better than a predictive one.
- For example: predictive models work well when the project is relatively small; we know exactly what you need to do, and the time scale is short enough that the requirement won't change during development.

# PREDICTIVE DEVELOPMENT MODEL

- Predict in advance **what needs to be done.**
- Based on past experience, it is easy to predict the time to build.
- It’s often hard to predict exactly what a software application needs to do and how we build it ahead of time.
- Sometimes, may not be familiar with the new programming tool.
- In changing business situations ,customer’s needs also changes as time goes.

- ***Predictive models*** are useful primarily because they give a lot of structure to a project.
- It is Empirical Software Engineering.
- The goal of such methods is ***repeatable, refutable (and possibly improvable) results*** in software engineering.
- Some of the predictive software development models are ***waterfall model***, ***waterfall with feedback***, ***sashimi model, incremental waterfall model***, ***V-model*** and ***software development life cycle.***

# Success and Failure Indicators

**Success indicators**

- User involvement
- Clear vision
- Limited size
- Experienced team
- Realistic
- Established technology

**Failure indicators**

- Incomplete requirements
- Unclear requirements
- Changing requirements
- No resources

# Advantages of Predictive Model

- Predictability
- Stability
- Cost-savings
- Detailed design
- Less refactoring
- Fix bugs early
- Better documentation
- Easy maintenance

- **Predictability –** If everything goes according to plan, we can know exactly when different stages will occur and when we will be finished.
- **Stability–** customers know exactly what they are getting.
- **Cost-savings–** if the design is clear and correct, we won't waste time.
- **Detailed design–** if we design everything up front, we shouldn't waste time making a lot of decisions during development.

- **Less refactoring –** Adaptive projects tend to require refactoring.
  - A programmer writes some code, sometimes later, the requirements change and the code needs to be modified.
  - These problems don't occur as often in predictive projects.
- **Fix bugs early –** If the requirements and design are correct and complete, we can fix the bugs they would have caused later.

- **Better documentation –** Predictive models require a lot of documentation. This documentation helps the new people to understand the projects in a better way.
- **Easy maintenance –** in the predictive model we can create a more elegant design that's more consistent and easier to maintain.

# Disadvantages of Predictive Model

- **Inflexible:** If the requirements change, it is very hard to implement**.**
- **Later initial release:** Many adaptive models enables us to give the users a program as soon as it does something useful. With a predictive model, the users don’t get anything until development is finished.
- **Big Design Up Front (BDUF):** we need to define everything up front. We can’t start development until we know exactly everything what we are going to do.

# ITERATIVE VERSUS PREDICTIVE

- Problem with **predictive model:** Not suited to handle unexpected changes.
- Can deal with small changes; don't handle big changes well.
- Spend a lot effort at the beginning, figuring out exactly what they will do.
- Don’t handle fuzzy requirements well.
- **Iterative models** address those problems by building the application incrementally.

# ITERATIVE MODEL

- Start by building the smallest program that is reasonably useful.
- Then use a series of increments to add more features to the program until it is finished.
- Each increment has relatively small duration compared to predictive project.
- It handles fuzzy requirements reasonably well.
- Useful if we are unsure of some of the requirements.

# Comparison

- Suppose we are working on a project that provides three features.
- We use fidelity (degree of exactness) to describe different development approaches.

**Predictive:**

- Provides all three features at the same time with full fidelity.

**Iterative:**

- Initially provides all three features at a low (but usable) fidelity. Later iteration provide higher and higher fidelity until all the features are provided with full fidelity.

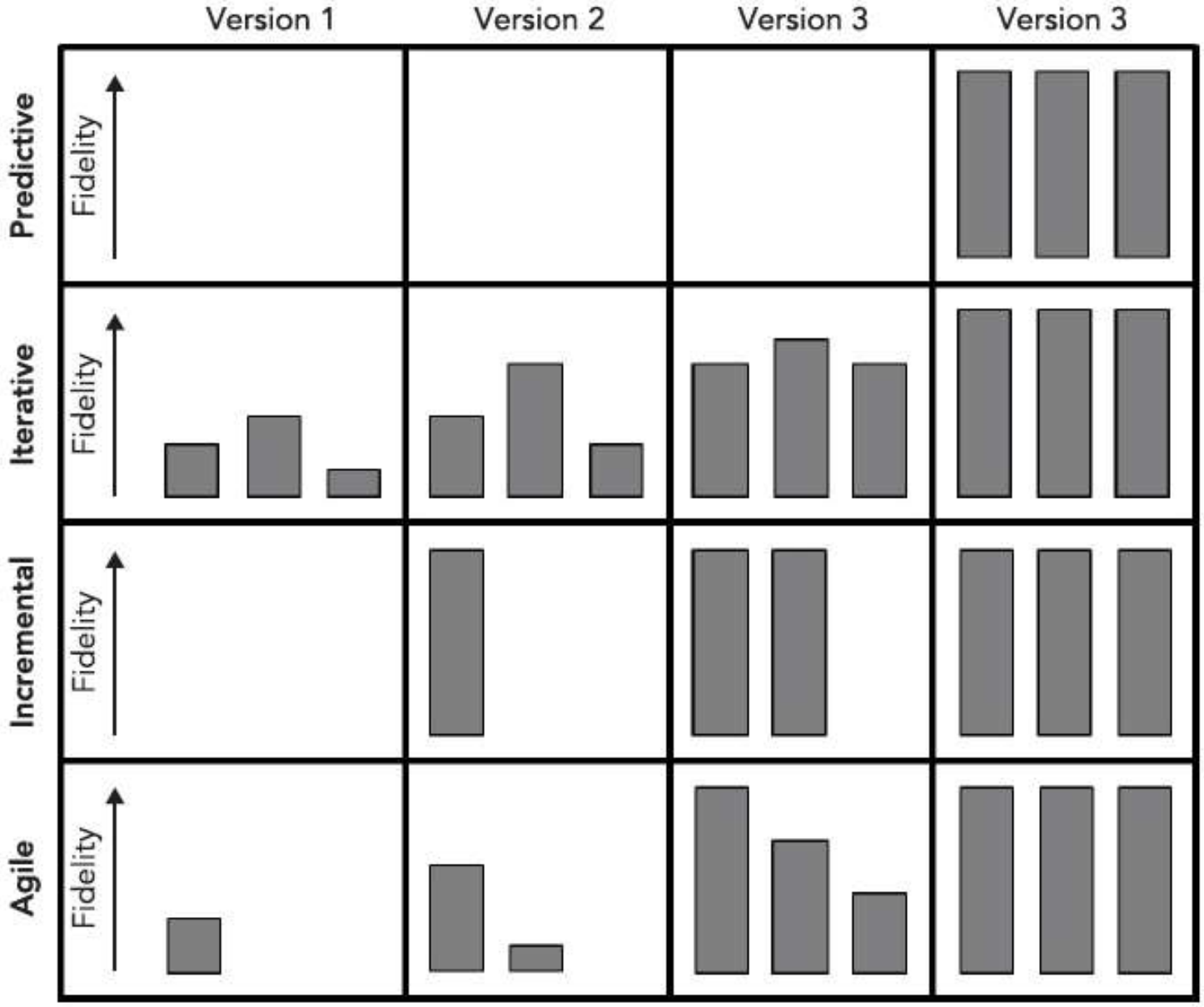
**Increment:**

- Initially provides the fewest possible features for a usable application, but all the features present are provided with full fidelity. Later versions add more features, always at full fidelity.

**Agile:**

- Initially provides the fewest possible features at low fidelity. Later versions improve the fidelity of existing features and add new features. Eventually all the features are provided at full fidelity.

All four of those approaches end with an application that includes all the features at full fidelity. It's the routes they take to get to their final solutions that differ.



# WATERFALL MODEL

The *waterfall model,* sometimes called the ***classic life cycle****, suggests a systematic,* sequential approach to software development that begins with customer specification of requirements and progresses through planning, modeling, construction, and deployment, culminating in ongoing support of the completed software

# WATERFALL MODEL

- Predictive model
- The waterfall model is the oldest paradigm for software engineering.
- Finish each step completely and thoroughly before move on to the next step.
- The waterfall analogy can be explained like this:
  - The water represents information and the stages act like buckets.
  - When one bucket is full, the information floods from that bucket into the next so that it can direct the following task.

# WATERFALL MODEL

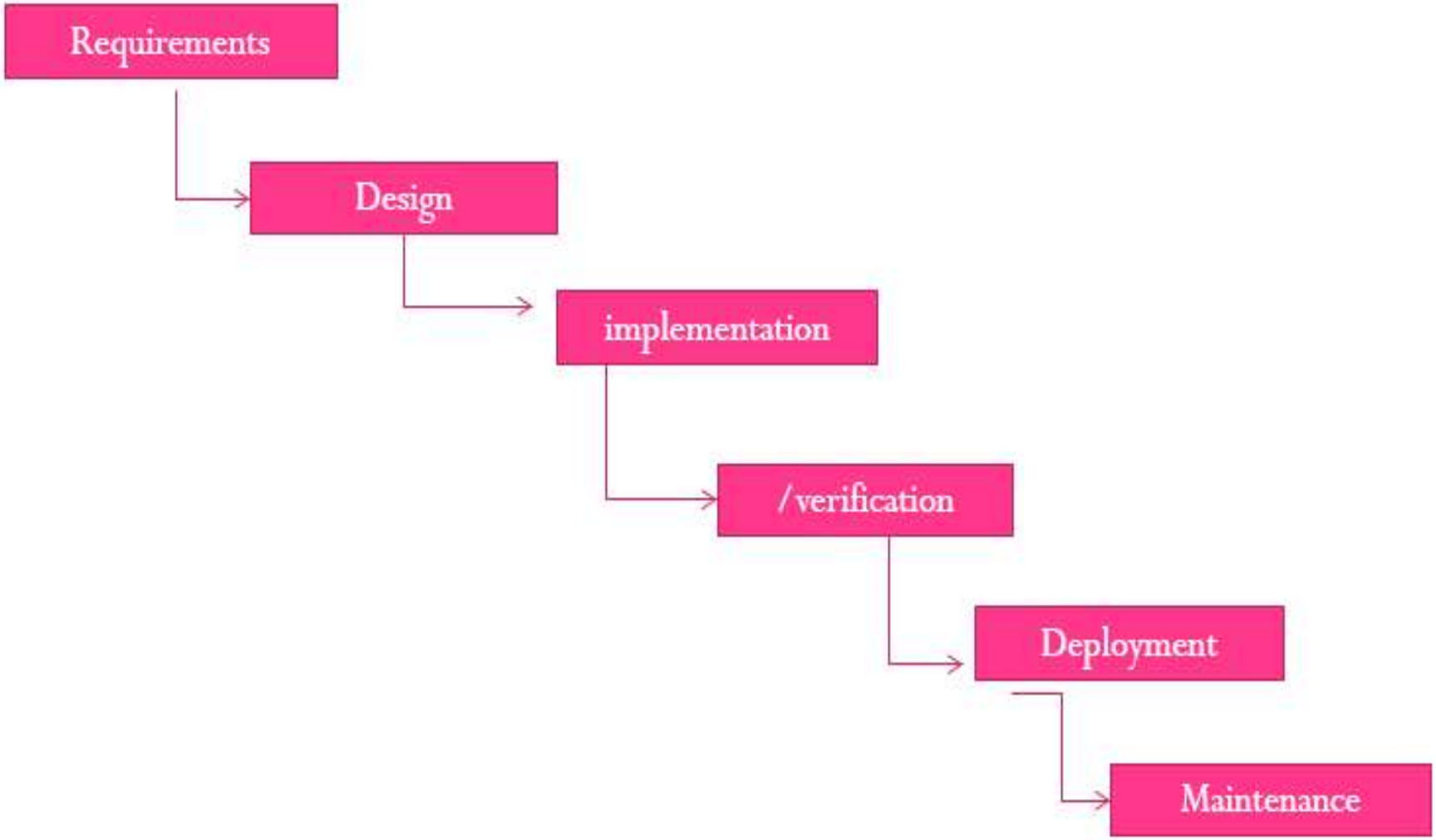
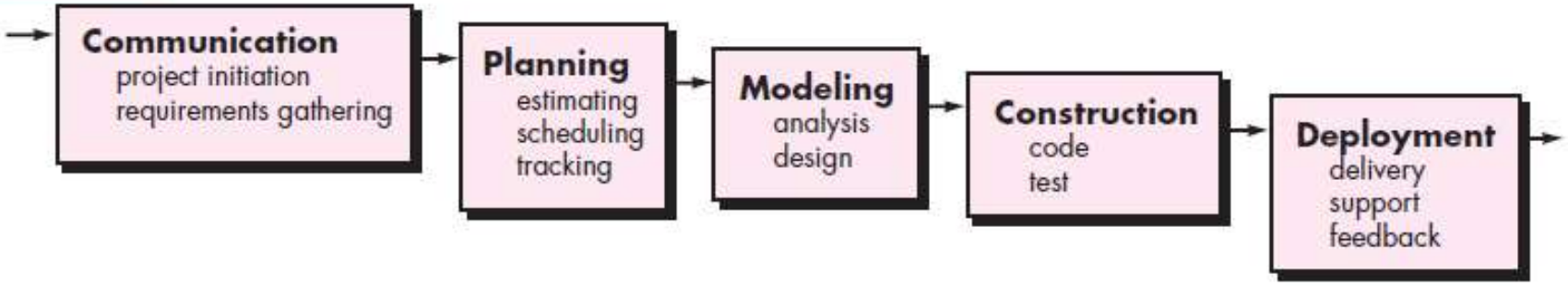


**FIGURE 2.3** The waterfall model



- **The model can work reasonable well if all the following assumptions are satisfied:**

1. The requirements are precisely known in advance
2. The requirements include no unresolved high risk items
3. The requirements won't change much during development.
4. The team has previous experience with similar projects so that they know what is involved in building the application.
5. There is enough time to do everything sequentially.

- You can add additional steps or split steps to give more details if you like.
- You can also elaborate on a step .
- A variation in the representation of the waterfall model is called the ***V-model.***

# Waterfall Strengths

- **Easy to understand**, easy to use
- **Provides structure** to inexperienced staff
- **Milestones are well understood**
- Sets requirements **stability**
- Good for **management control** (plan, staff, track)

# Waterfall Deficiencies

- All **requirements must be known** upfront
- Deliverables created for each phase are considered **frozen**
- Can give a **false impression** of progress
- Does not reflect **problem-solving nature of software** development – iterations of phases
- **Integration** is one big bang at the end
- **Little opportunity for customer** to preview the system (until it may be too late)

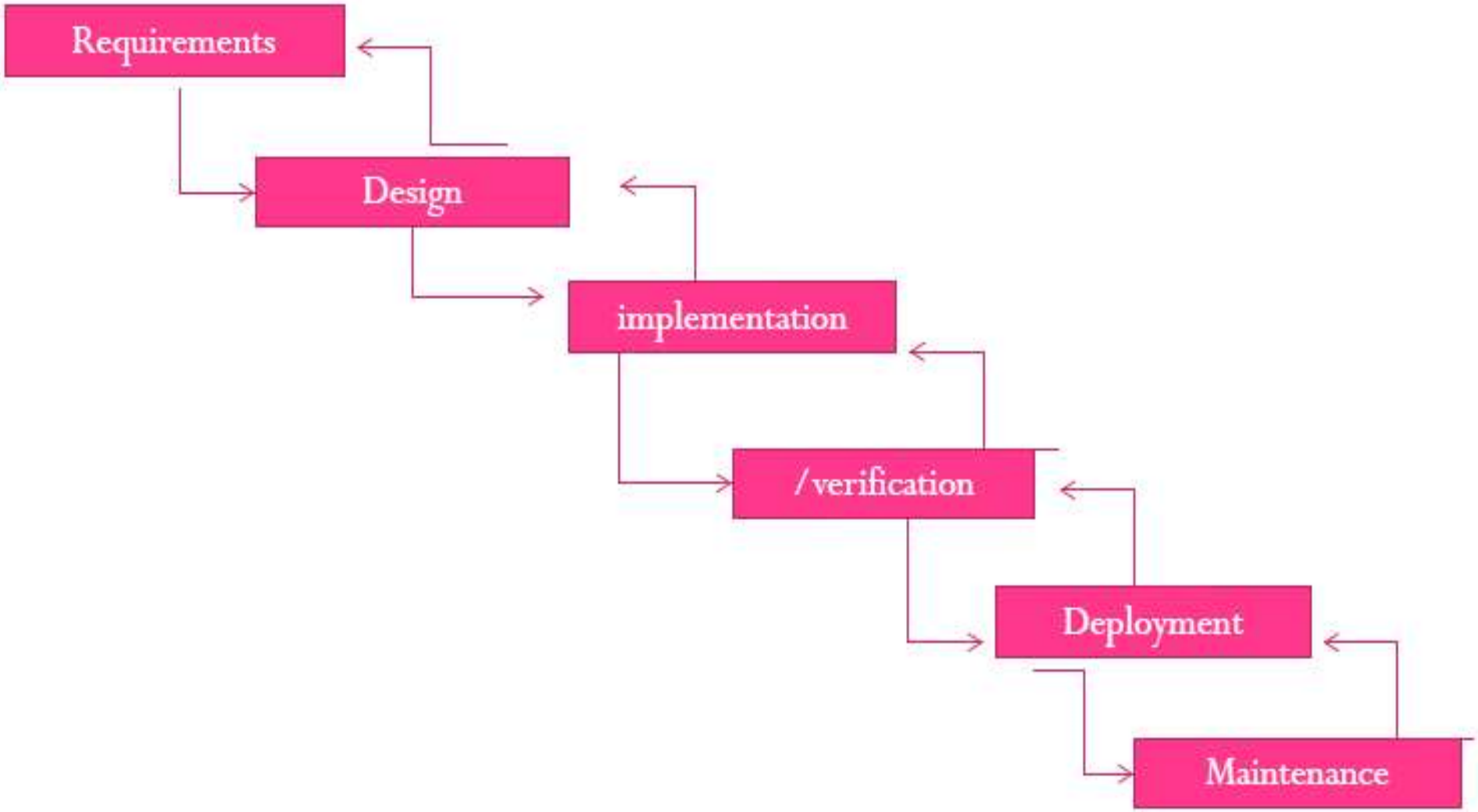
# When to use the Waterfall Model

- Requirements are **very well known**
- Product definition is **stable**
- Technology is **understood**
- New **version of an existing product**
- **Porting an existing product** to a new platform.

# Why does the waterfall model sometimes fail?

1. Real projects rarely follow the sequential flow that the model proposes.
2. It is often difficult for the customer to state all requirements explicitly.
3. The customer must have patience. A working version of the program(s) will not be available until late in the project time span

# WATERFALL MODEL WITH FEEDBACK



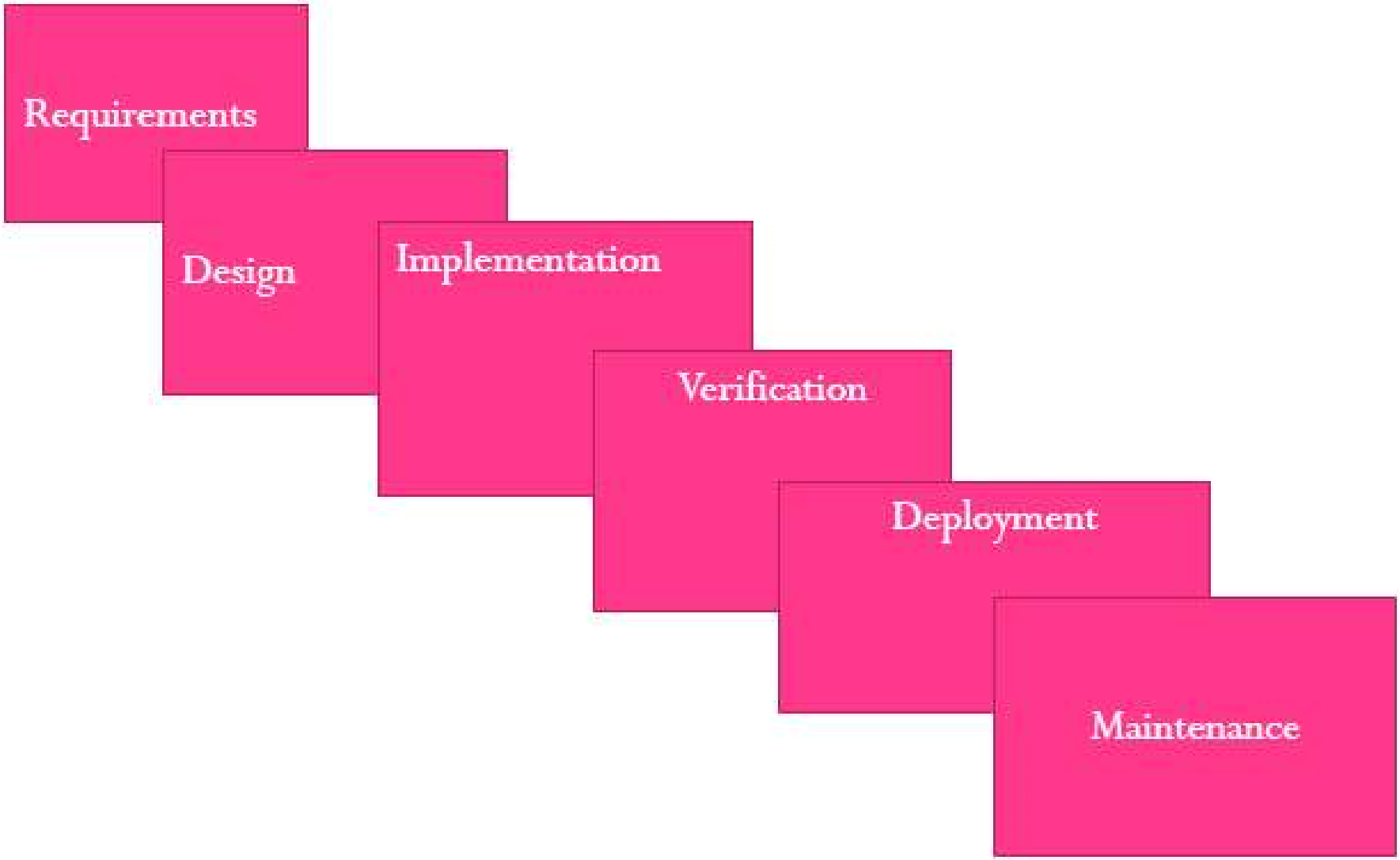
- Enables the developer to move backward from one step to the previous step.
- If we are working on design and discover that there was a problem in the requirements ,we can briefly go back to the requirements and fix them.
- In short, the waterfall model with feedback enables us to move backward from one step to the previous step.

- The farther you have to go back up the cascade, the harder it is.
  - For example, if we are working on implementation and discover a problem in the requirements, it's hard to skip back up two stages to fix the problem.
- It's also less meaningful to move back up the cascade for the later steps.
  - For example, if we find a problem during maintenance, then we should probably treat it as a maintenance task instead of moving back into the deployment stage.

# SASHIMI

- Also called ***sashimi waterfall*** or ***waterfall with overlapping phase.***
- A modified version of the waterfall model.
- Is similar to the waterfall except the steps are allowed to overlap.
- Introducing feedback into the classical waterfall model.

# SASHIMI


Requirements
Design
Implementation
Verification
Deployment
Maintenance

## Advantages :-

- In a project's first phase, some requirements will be defined while you are still working on others.
  - For example, some of the team members can start designing the defined features while others continue working on the remaining requirements.
- You can allow greater overlap between project phases.
  - For example, some developers can start writing code for the designed parts of the system while others continue working on the rest of the design tasks, and may be on remaining requirements.

- We can allow greater overlap between project phases.
  - If we have people working on requirements, design, implementation, testing all at the same time.
  - People with different skills can focus on their specialities without waiting for others.
  - It lets you perform a spike or deep dive into a particular topic to learn more about it.
- It lets later phases modify earlier phases.
  - If we discover during design that the requirements are impossible or need alterations, we can make the necessary changes.

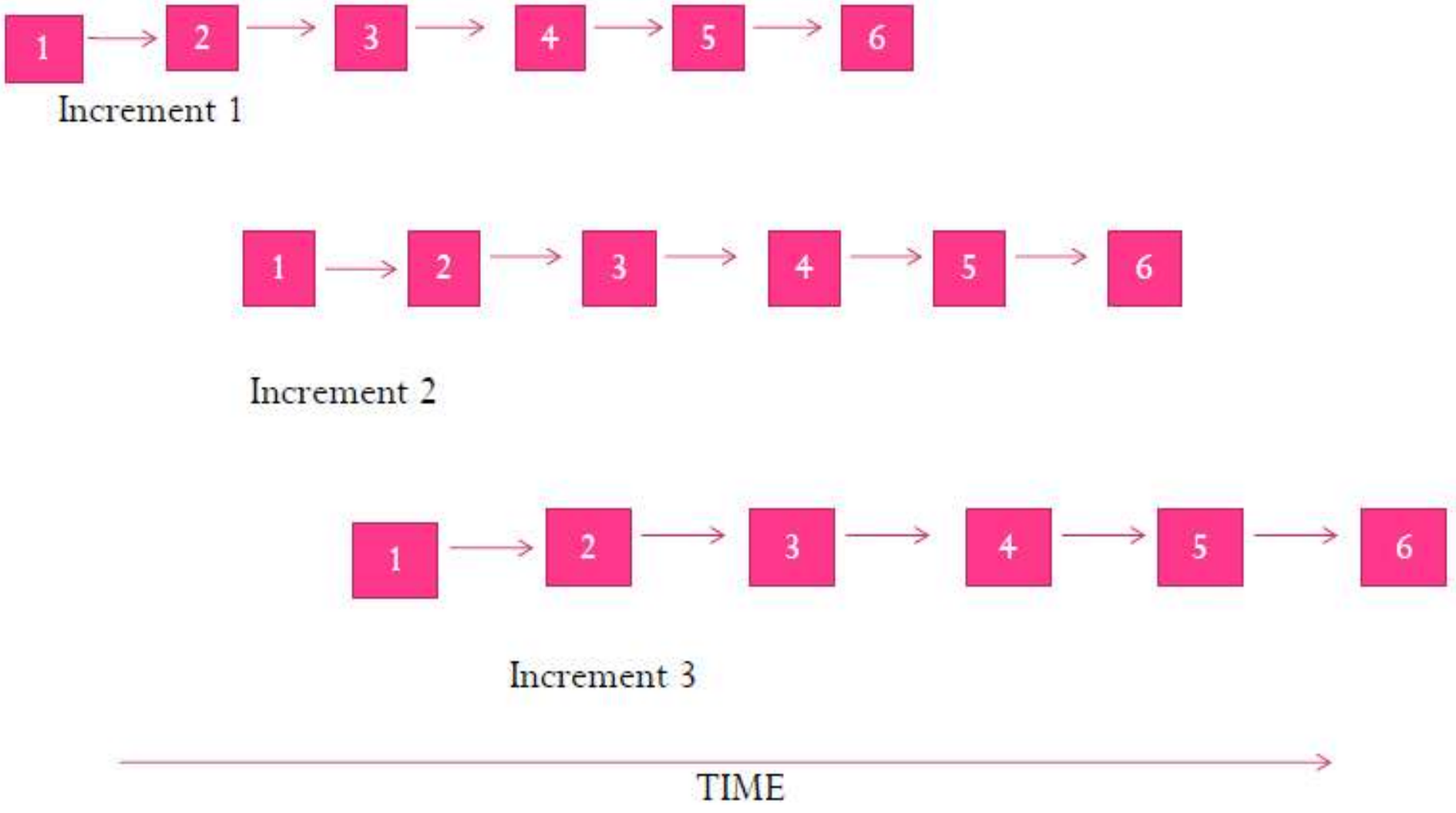
**Disadvantage:**

- some parts of the application will be more or less finished but the other parts of the system won’t be.

# Incremental waterfall

- Also called multi-waterfall model.
- Uses a series of separate waterfall cascades.
- Each cascade ends with the delivery of a usable application called an increment
- Each increment includes more features than the previous one.
- So in incremental waterfall model we are building the final application incrementally.

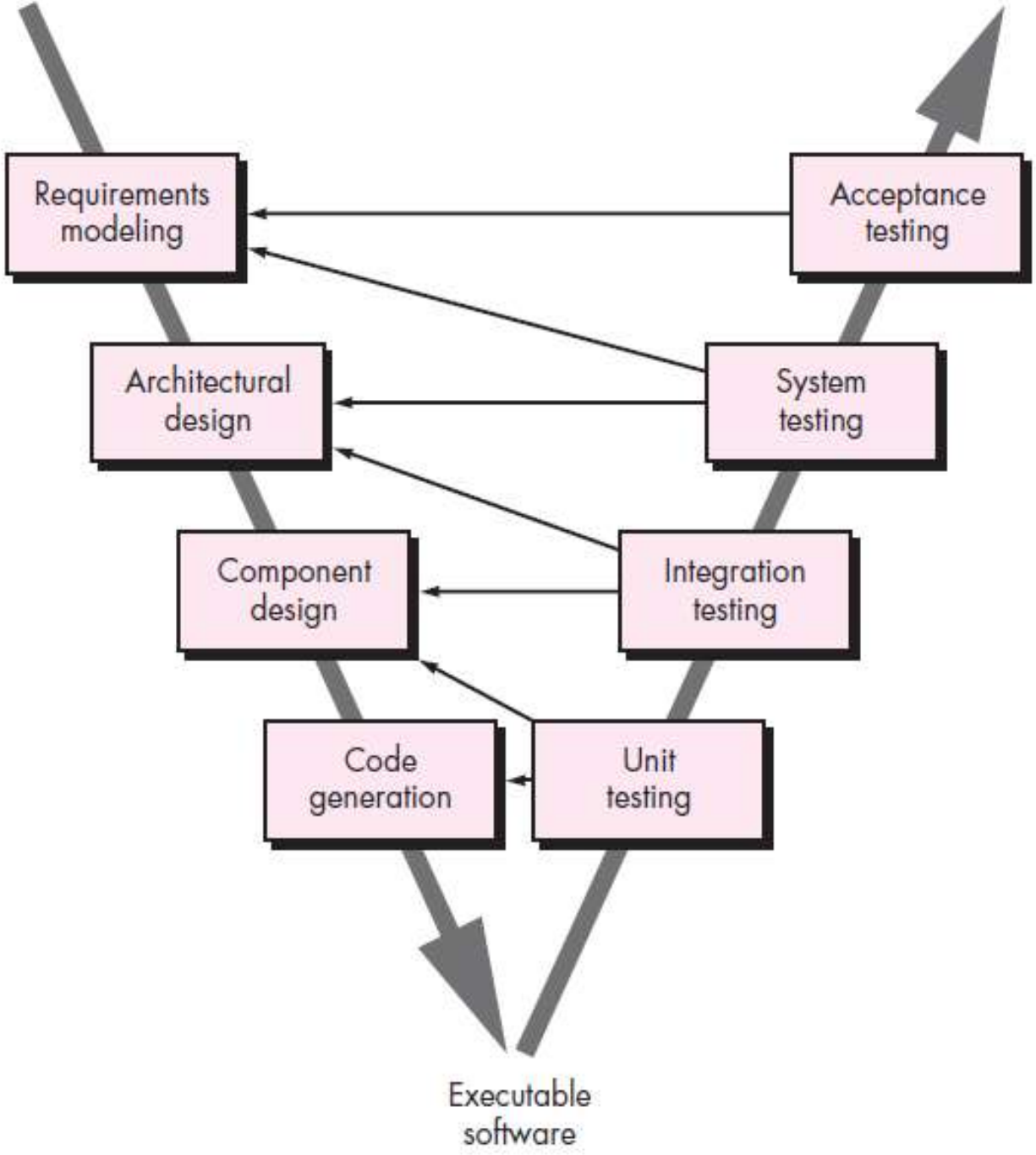
# Incremental waterfall



- During each increment you will get a better understanding of what the final application should look like.
- We learn what did and didn’t work well in the previous increment.
- If we understand what we need to do in the next iteration, we don’t need to wait until the current iteration is completely finished.

- The incremental waterfall is **somewhat adaptive model because its lets you to re-evaluate the direction at the start of each increment.**
- The incremental waterfall model usually take long time to complete one iteration.
- We can change direction when we start a new increment, but within each increment the model runs predictively. In that sense, it is **not all that adaptive.**

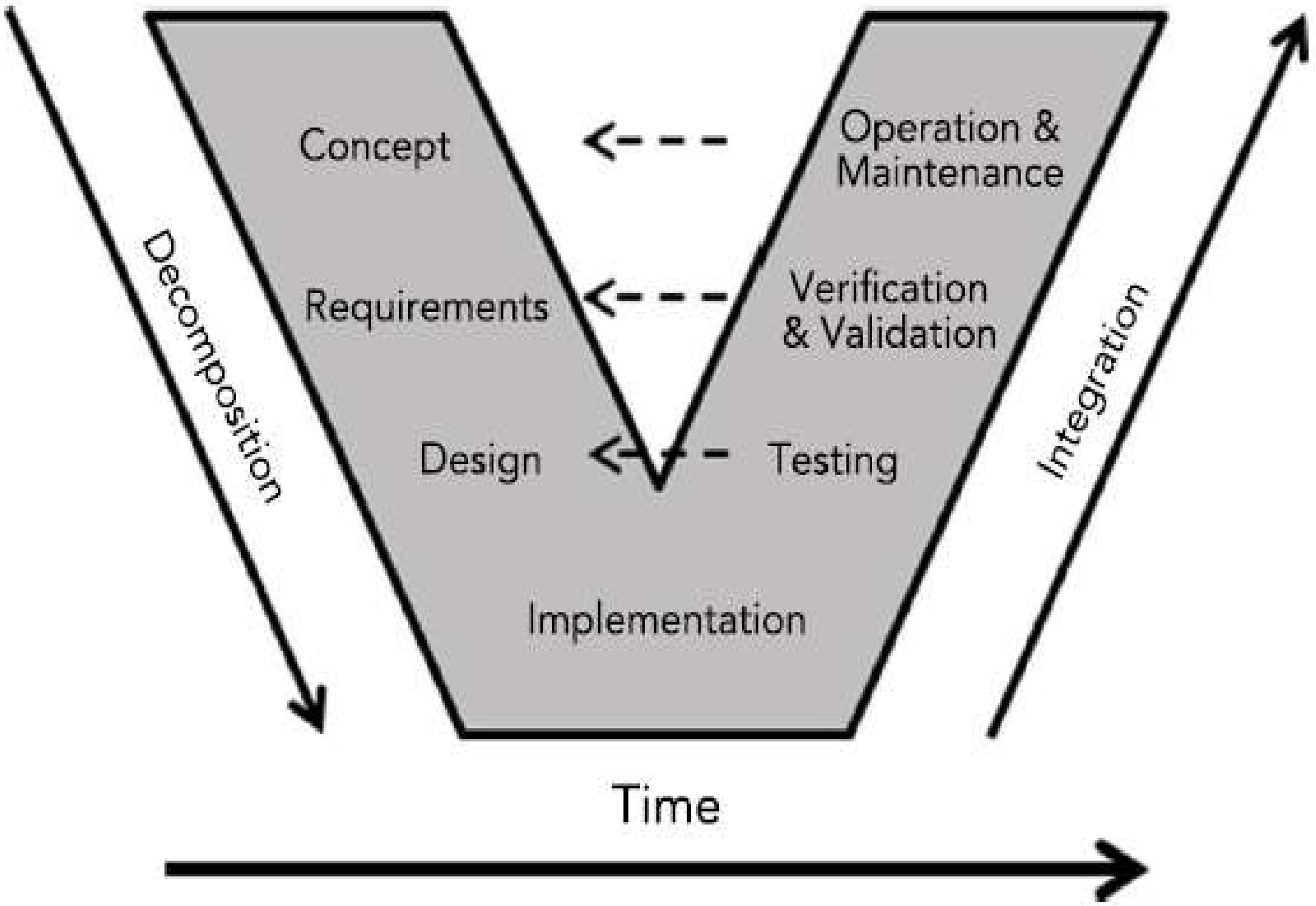
# V-Model


Requirements modeling
Acceptance testing
Architectural design
System testing
Component design
Integration testing
Code generation
Unit testing
Executable software

- Basically a waterfall model that's been bent into a V-shape **Or** A variation in the representation of the waterfall model is called the *V-model.*
- As a software team moves down the left side of the V, basic problem requirements are refined into progressively more detailed and technical representations of the problem and its solution.

- Once code has been generated, the team moves up the right side of the V, essentially performing a series of tests (quality assurance actions) that validate each of the models created as the team moved down the left side.
- The task on the left side of the V break the application down from its highest conceptual level into more and more detailed tasks.
- The process of breaking the application down into pieces is called **Decomposition.**

- The tasks on the right side of the V consider the finished application at greater and greater levels of abstraction.
- At the lowest level, ***Testing verifies that the code works.***
- At the next level, ***verification confirms that the application satisfies the requirements, and validation confirms that the application meets customer needs.***
- The process of working backup to the conceptual top of the application is called **Integration.**


Concept
Requirements
Design
Implementation
Testing
Verification & Validation
Operation & Maintenance
Decomposition
Integration
Time

- Each of the tasks on the left corresponds to a task on the right with a similar level of abstraction.
- At the highest level, the initial concept corresponds to operation and maintenance.
- At the next level, the requirements correspond to verification and validation.
- Testing confirms that the design worked.

- In reality, there is no fundamental difference between the classic life cycle and the V-model.
- The V-model provides a way of visualizing how verification and validation actions are applied to earlier engineering work.

# Prototypes

- It can be useful in iterative development.
- Prototype is a simplified model that demonstrates some behaviour of the project.
- Prototype does not work exactly the same way the finished application will work.
- However it lets the customer to see what the application will look like.
- After the customers experiment with the prototype, they can give us feedback to help refine the requirements.

# Types of prototypes

1. Throw away prototype
2. Evolutionary prototype
3. Incremental prototype

- In a **throw away prototype**, we use the prototype to study some aspect of the system and then we throw it away and write code from scratch.

- In an **evolutionary prototype**, the proto type demonstrates some of the application's features.
- As the project progresses, we refine those features and add new ones until the prototype morphs into the finished application.

- In **incremental prototyping**, we build a collection of prototypes of that separately demonstrate the finished application's features.
- We then combine the prototypes to build the finished application.

## HORIZONTAL AND VERTICAL PROTOTYPES

A *horizontal prototype* is one that demonstrates a lot of the application's features but with little depth. For example, the prototype described earlier that lets a user pretend to navigate through customer orders is a horizontal prototype. Horizontal prototypes are often user interface prototypes that let customers see what the finished application will look like.

In contrast, a *vertical prototype* is one that has little breadth but great depth. The example described earlier that has no user interface and generates an invoice for a single customer is a vertical prototype.

# Advantages of Prototype

1. Improved requirements
2. Common vision
3. Better design

## Improved requirements

- Prototypes allow customers to see what the finished application will look like.
- That lets them provide feedback to modify the requirements early in the project.
- Often customers can spot problems and request changes earlier so the finished result is more useful to users.

## Common vision

- Prototypes let the customers and developers see the same preview of the finished application, so they are more likely to have a common vision of what the application should do and what it should look like.

## Better design

- Prototypes let the developers quickly explore specific pieces of the application to learn what they involve.
- It also help them to improve the design and make the final code more elegant and robust

# Disadvantages of Prototype

1. Narrowing vision
2. Customer impatient
3. Scheduled pressure
4. Raised expectation
5. Attachment to code
6. Never-ending prototype

## Narrowing vision

- People (customers and developers) tend to focus on a prototype's specific approach rather than on the problem it addresses.

## Customer impatience

- A good prototype can make customers think that the finished application is just around the corner.

**Scheduled pressure**

- If customers see a prototype that they think is mostly done, they may not understand that we need another year to finish and may pressure to shorten the schedule.

**Raised expectation**

- Sometimes, a prototype may demonstrate features that won't be included in the application.

**Attachment to code**

- Sometimes, developers become attached to the prototype's code.
- Initial code might have low quality.

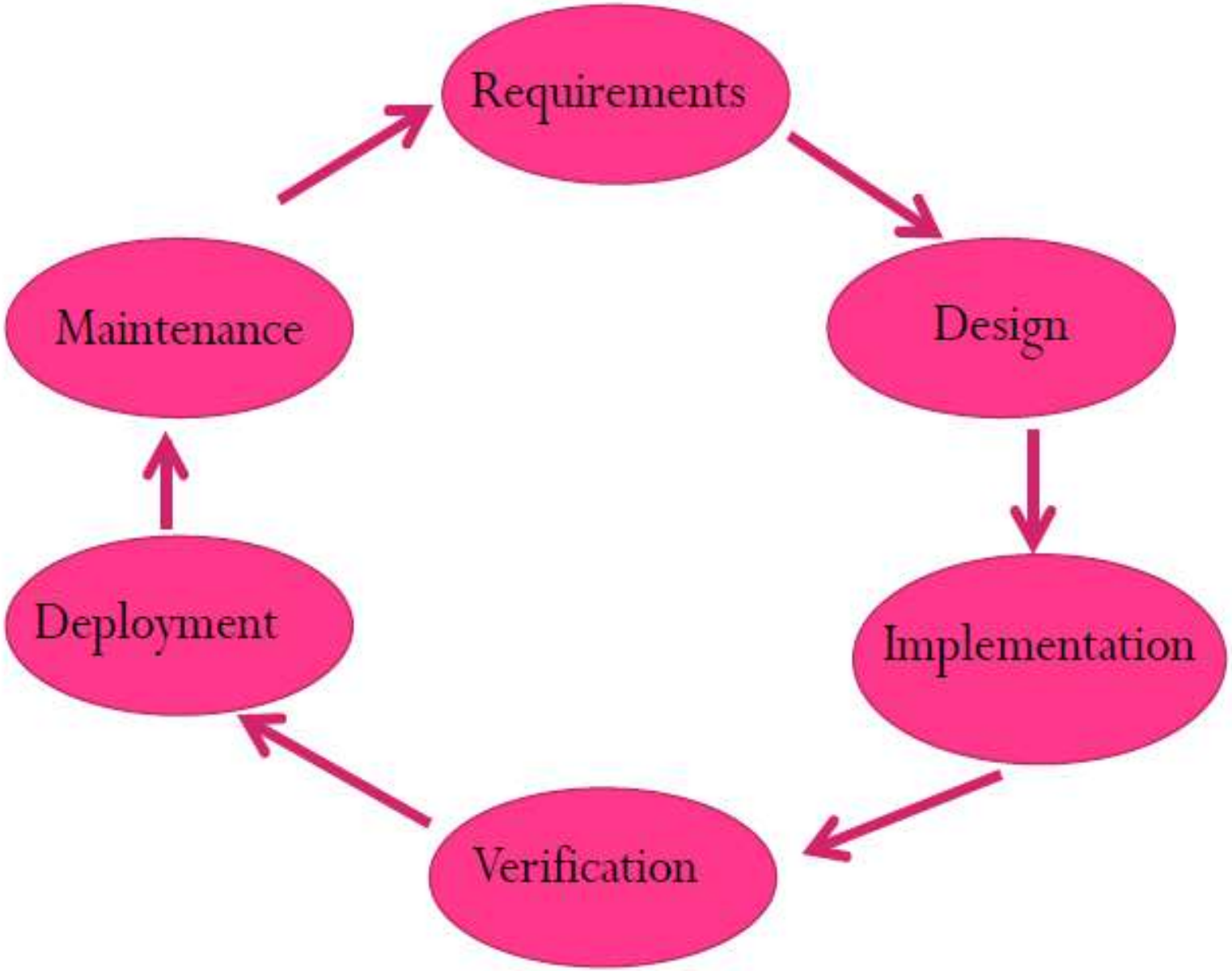
**Never- ending prototype**

- Sometimes, developers spend far too much time refining a prototype to make it look better and include more features that aren't actually necessary.

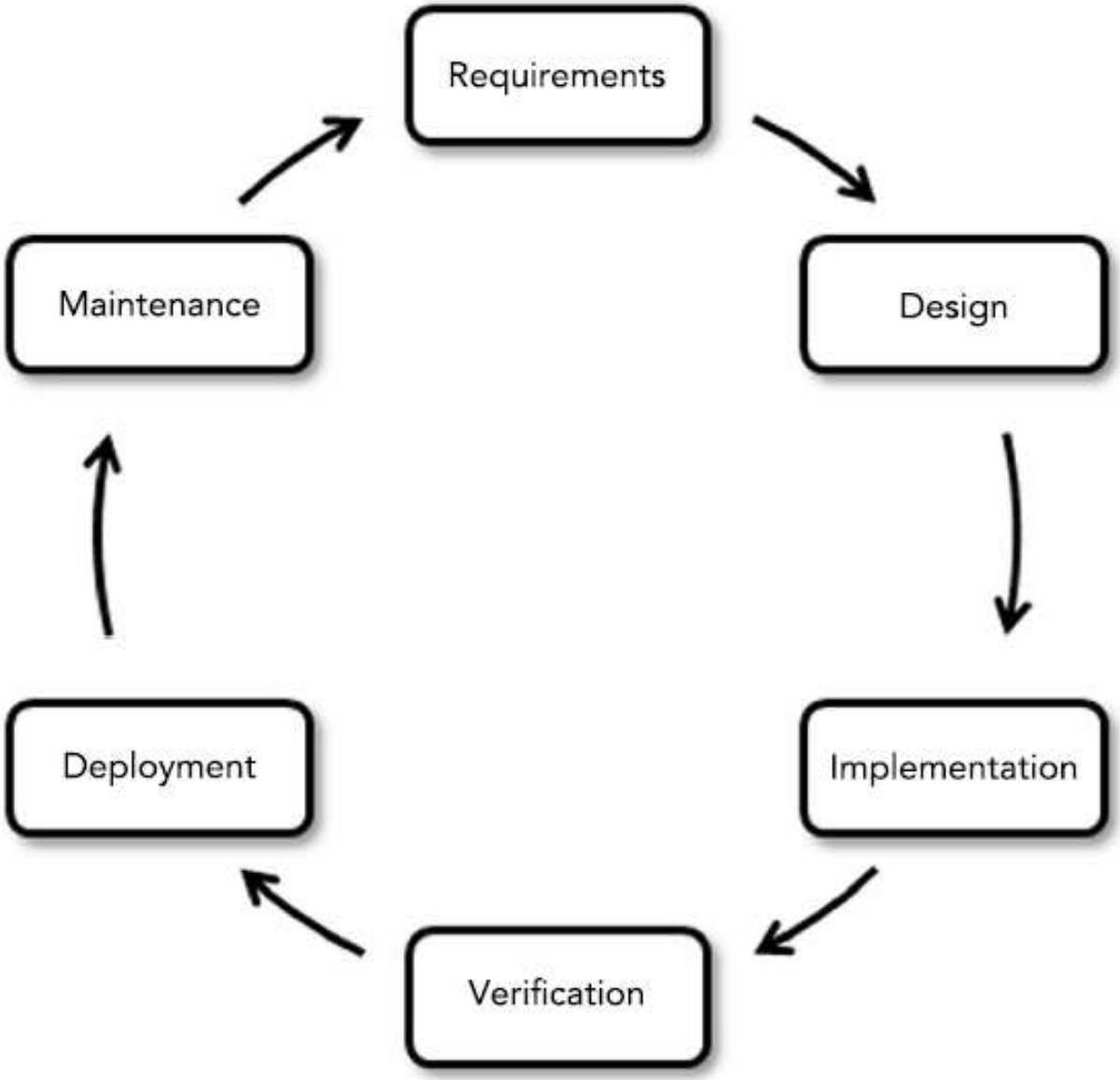
# System Development life cycle (SDLC )

- Also called **Application Development Life Cycle.**
- It covers all the tasks that go into a software engineering project from start to finish-like waterfall model.
- The end of one project feed directly into the next project in an ever ending cycle.

- The incremental waterfall model is basically just a series of SDLCs and with some overlap, so one project starts before the previous one is completely finished.
- Incremental waterfall is a version of SDLC
- We can break down the basic steps in a lot more detail if we like.




In SDLC, project phases feed into each other in a potentially infinite loop.


Requirements
Design
Implementation
Verification
Deployment
Maintenance

- Incremental waterfall is a version of SDLC
  - The incremental waterfall model is basically just a series of SDLCs and with some overlap, so one project starts before the previous one is completely finished.
- You can break down the basic steps in a lot more detail if you like.

# Tasks involved in SDLC

- **Initiation –** An initiator comes up with the initial idea.
- **Concept development –** Initial project definition, a feasibility analysis, a cost-benefit analysis and a risk analysis
- **Preliminary planning –**
  - Project Manager and technical lead are assigned to the project, and they start planning.
  - Breaking into small units and team leaders are assigned.
  - Leaders make preliminary planning on necessary resources like computers, staffing, network and development tools etc…

- **Requirement analysis -** The team studies the user's needs and creates requirement documents.
- It includes UML diagrams, use cases, prototypes and so on.
- **High level design** – The team creates high-level design data flow, data base needs.
- **Low level design** – The team creates low-level designs that explain how to build the different modules.
- **Development** – The team writes the program code.

- **Acceptance testing –** The customers get a chance to take the application for a test drive in its final form.
- **Deployment –** The team rolls out the application**.**
- **Maintenance –** The team continues to track the application's usefulness throughout its lifetime to determine whether it needs repair, enhancement or replacement with a new version**.**
- **Review –** The team uses metrics to assess the project and decide whether the development process can be improved in the future.
- **Disposal –** Eventually, the application's usefulness comes to an end. So we may remove or replace with something else.

# Predictive Model - Characteristics

- They give a lot of structure to a project
- Have a fully developed plan that can follow throughout the project's life time.
- Predictive project make scheduling simpler, includes documentation, costless etc
- But will not handle changes

# Module - 1

- Software requirements specification
- Eliciting Software requirements
- Requirement specifications
- Software requirements engineering concepts
- Requirements modelling
- Use cases and User stories.
- Requirements documentation.

# Software Requirements Specification (SRS)

- A **requirement** can range from a high-level, abstract statement of a service or constraint to a detailed, formal specification.
- SRS is the set of activities designed to capture behavioral and non-behavioral aspects of the system in the SRS document.
- The goal of the SRS activity, and the resultant documentation, is to provide a complete description of the system's behavior without describing the internal structure.
- This aspect is easily stated, but difficult to achieve.

- Software specifications provide the basis for analyzing the requirements, validating that they are the stakeholder's intentions, defining what the designers have to build, and verifying that they have done so correctly.
- SRSs allow us to know the motivation for development of the software system.
- Software requirements also help software engineers manage the evolution of the software over time and across families of related software products.

# Requirements Engineering Activities

- First, we must **elicit the requirements**, which is a form of discovery.
- The term “gathering” is also used to describe the process of collecting software requirements.
- Often, requirements are deeply hidden, expressed incorrectly by stakeholders, contradictory, and complex.
- Eliciting requirements is one of the hardest jobs for the requirements engineer.

- **Modeling requirements** involves representing the requirements in some form.
- Words, pictures, and mathematical formulas can all be used but it is never easy to effectively model requirements.
- **Analyzing requirements** involves determining if the requirements are correct or have certain other properties such as consistency, completeness, sufficient detail, and so on.
- Finally, requirements change all the time and the requirements engineer must be equipped to deal with this eventuality.

- SRSs are usually classified in terms of their level of abstraction:

    1. user requirements

    2. system requirements

    3. software design specifications

- **User requirements specifications** are usually used to attract bidders, often in the form of a request for proposal (RFP).
  - User requirements may contain abstract statements in natural language with accompanying informal diagrams.
  - User requirements specify functional and non-functional requirements as they pertain to externally visible behavior in a form understandable by clients and system users.

- **System level requirements**, or SRSs mentioned previously, are detailed descriptions of the services and constraints.
  - Systems requirements are derived from analysis of the user requirements.
  - Systems requirements should be structured and precise because they act as a contract between client and contractor (and can literally be enforced in court).

- **Software design specifications** are usually the most detailed level requirements specifications that are written and used by software engineers as a basis for the system's architecture and design.

- **The user needs are usually called "functional requirements" and the external constraints are called "non-functional requirements."**

- **Functional requirements** describe the services the system should provide.
- Sometimes the functional requirements state what the system should not do.
- Functional requirements can be high-level and general or detailed, expressing inputs, outputs, exceptions, and so on.
- **Non-functional requirements** are imposed by the environment in which the system is to exist.
- These requirements could include timing constraints, quality properties, standard adherence, programming languages to be used, compliance with laws, and so on.

**Functional requirements**

- ***Interface specifications***
- ***Performance requirements***
- ***Logical database requirements***
- ***System attribute requirements***

**Non-Functional requirements**

- ***Domain requirements***
- ***Design constraint requirements***

- **Interface specifications** are functional software requirements specified in terms of interfaces to operating units.
- Most systems must operate with other systems and the operating interfaces must be specified as part of the requirements.
- There are three types of interface that may have to be defined: procedural interfaces, data structures that are exchanged , and  data representations.
- **System attribute requirements** are functional requirements that include reliability, availability, security, maintainability, and portability.

- **Performance requirements** are functional requirements that include the static and dynamic numerical requirements placed on the software or on human interaction with the software as a whole.
- **Logical database requirements** are functional requirements that include the types of information used by various functions such as frequency of use, accessing capabilities, data entities and their relationships, integrity constraints, and data retention requirements.

- **Domain requirements** are a type of non-functional requirement from which the application domain dictates or derives.
  - For example, in the baggage inspection system, industry standards and restrictions on baggage size and shape will place certain constraints on the system.
- **Design constraint requirements** are non-functional requirements that are related to standards compliance and hardware limitations.

**What is a feasibility study and what is its role?**

- A feasibility study is a short focused study that checks if the system contributes to organizational objectives the system can be engineered using current technology and within budget the system can be integrated with other systems that are used.
- A feasibility study is used to decide if the proposed system is worthwhile.

- Feasibility studies can also help answer some of the following questions.

    1. What if the system wasn’t implemented?
    2. What are current process problems?
    3. How will the proposed system help?
    4. What will be the integration problems?
    5. Is new technology needed? What skills?
    6. What facilities must be supported by the proposed system?

# Requirements elicitation

- Requirements elicitation involves working with customers to determine the application domain, the services that the system should provide, and the operational constraints of the system.
- Elicitation may involve end-users, managers, engineers involved in maintenance, domain experts, trade unions, and so on. These people are collectively called **stakeholders**.
- But stakeholders don’t always know what they really want.

- The software engineer has to be sensitive to the needs of the stakeholders and aware of the problems that stakeholders can create including: expressing requirements in their own terms
  - providing conflicting requirements
  - introducing organizational and political factors, which may influence the system requirements
  - changing requirements during the analysis process due to new stakeholders who may emerge and changes to the business environment
- The software engineer must monitor and control these factors throughout the requirements engineering process.

The following three approaches to requirements elicitation will be discussed in detail:

- **Joint application design (JAD)**
- **Quality function deployment (QFD)**
- **Designer as apprentice**

# JAD

- **Joint application design (**JAD) involves highly structured group meetings or mini-retreats with system users, system owners, and analysts in a single room for an extended period.
- These meetings occur four to eight hours per day and over a period lasting one day to a couple of weeks.

Software engineers can use JAD for:

i. eliciting requirements and for the SRS
ii. design and software design description
iii. code
iv. tests and test plans
v. user manuals

- Planning for a review or audit session involves three steps:
  i. selecting participants
  ii. preparing the agenda
  iii. selecting a location

# Rules for JAD sessions

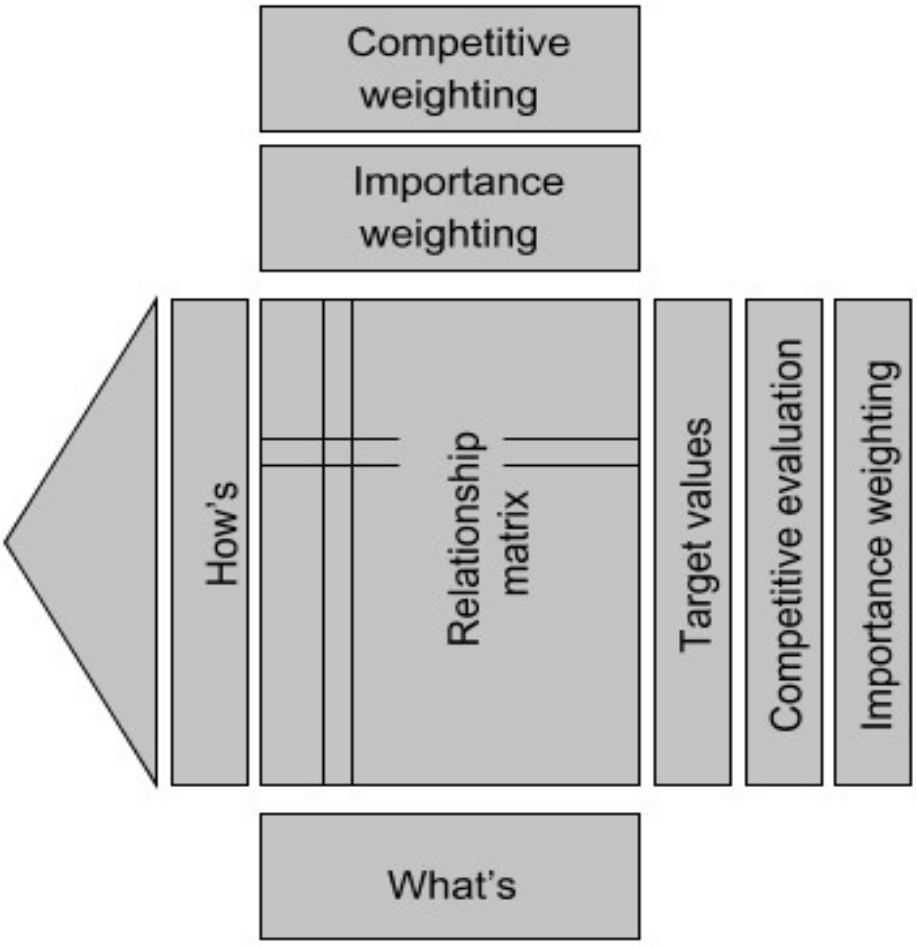
The session leader must make every effort to ensure that these practices are implemented.

- Stick to the agenda.
- Stay on schedule
- Ensure that the scribe is able to take notes.
- Avoid technical jargon.
- Resolve conflicts.
- Encourage group consensus.
- Encourage user and management participation without allowing individuals to dominate the session.
- Keep the meeting impersonal.

# Quality function deployment (QFD)

- It is a technique for determining customer requirements and defining major quality assurance points to be used through-out the production phase.
- QFD provides a structure for ensuring that customers' wants and needs are carefully heard, and then directly translated into a company's internal technical requirements from analysis through implementation to deployment.

- The basic idea of QFD is to construct relationship matrices between customer needs, technical requirements, priorities, and competitor assessment.
- These relationship matrices are often represented as the roof, ceiling, and sides of a house, QFD is sometimes referred to as the “house of quality”


Competitive weighting
Importance weighting
How's
Relationship matrix
Target values
Competitive evaluation
Importance weighting
What's

# Advantages of QFD

- QFD improves the involvement of users and mangers.
- It shortens the development life cycle and improves overall project development.
- QFD supports team involvement by structuring communication processes.
- It provides a preventive tool to avoid the loss of information.

# Designer as apprentice

- Designer as apprentice is a requirements discovery technique in which the requirements engineer "looks over the shoulder" of the customer to enable the engineer to learn enough about the work of the customer to understand
  his needs.
- The relationship between customer and designer is like that between a master craftsman and apprentice.
- The apprentice learns a skill from the master just as we want the requirements engineer (the designer) to learn about the work from the customer.
- The apprentice is there to learn whatever the master knows.

- The designer must understand the structure and implication of the work, including:
    a. the strategy to get work done
    b. constraints that get in the way
    c. the structure of the physical environment as it supports work
    d. the way work is divided
    e. recurring patterns of activity
    f. the implications the above has on any potential system
- **Both customer and designer learn during this process; the customer learns what may be possible and the designer expands his understanding of the work.**

- **Requirements modeling** involves the techniques needed to express requirements in a way that can capture user needs.
- There are a number of ways to model software requirements; these include natural languages, informal and semiformal techniques, user stories, use case diagrams, structured diagrams, object-oriented techniques, formal methods, and more.
- English, or any other natural language, have many problems for requirements communication.
- These problems include lack of clarity and precision, mixing of functional and non-functional requirements, ambiguity, overflexibility, and lack of modularization.
- Every clear SRS must have a great deal of narrative in clear and concise natural language. But when it comes to expressing complex behavior, it is best to use formal or semiformal methods, clear diagrams or tables, and narrative as needed to tie these elements together.

# Use Cases

- Use cases are an essential artifact in object-oriented requirements elicitation and analysis and are described graphically using any of several techniques.
- Artifact – something observed in a scientific investigation or experiment that is not naturally present but occurs as a result of the preparative or investigative procedure.
- One representation for the use case is the use case diagram, which depicts the interactions of the software system with its external environment.

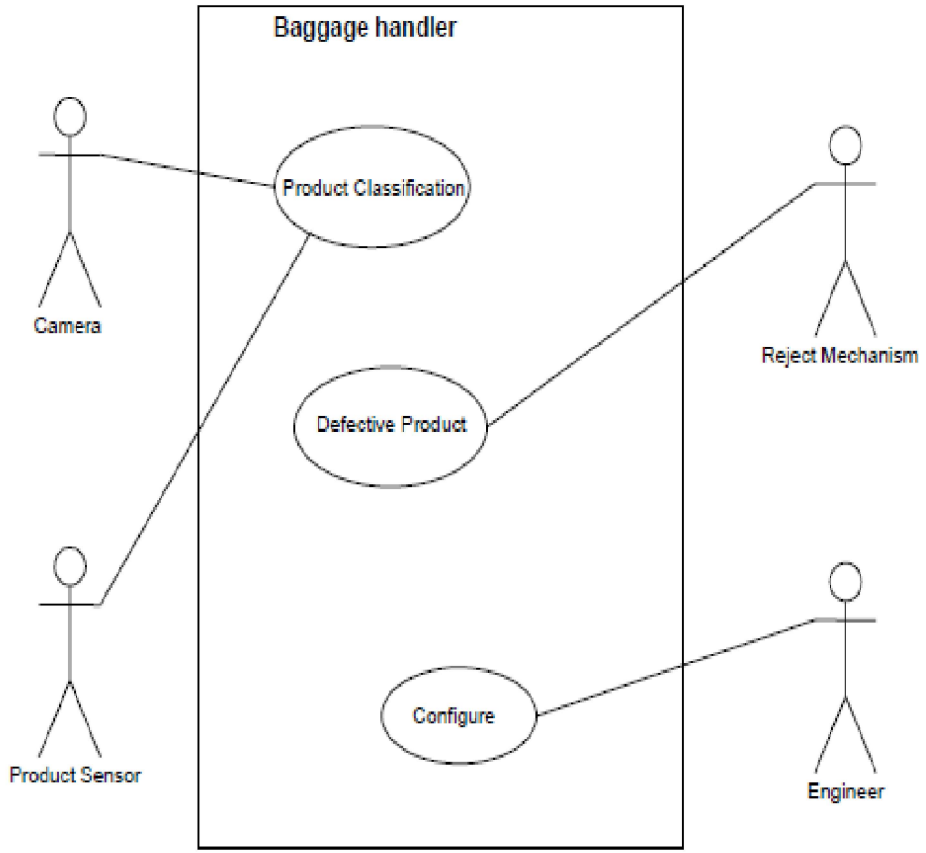



**FIGURE 3.3**
Use case diagram of the baggage inspection system.

In a use case diagram, the box represents the system itself.

The stick figures represent “actors” that designate external entities that interact with the system.

The actors can be humans, other systems, or device inputs.

Internal ellipses represent each activity of use for each of the actors (use cases).

The solid lines associate actors with each use.

- Each use case is a document that describes scenarios of operation of the system under consideration as well as pre- and post conditions and exceptions.
- In an iterative development life cycle, these use cases will become increasingly refined and detailed as the analysis and design workflows progress.

# User Stories

- User stories are short conversational texts that are used for initial requirements discovery and project planning.
- User stories are widely used in conjunction with agile methodologies.
- User stories are written by the customers in their own “voice,” in terms of what the system needs to do for them.
- User stories usually consist of two to four sentences written by the customer in his own terminology, usually on a three-by-five inch card.

- The appropriate amount of user stories for one system increment or evolution is about 80, but the appropriate number will vary widely depending upon the application size and scope
- User stories should provide only enough detail to make a reasonably low risk estimate of how long the story will take to implement.
- When the time comes to implement the story, developers will meet with the customer to flesh out the details.
- User stories also form the basis of acceptance testing.
  - For example, one or more automated acceptance tests can be created to verify the user story has been correctly implemented.

# Requirements Documentation

- The SRS document is the official statement of what is required of the system developers.
- The SRS should include both a definition and a specification of requirements.
- SRS is *not* a design document. It should be a set of *what* the system should do rather than *how* it should do it.
- A variety of stakeholders uses the software requirements throughout the software life cycle.

- Stakeholders include customers (these might be external customers or internal customers such as the marketing department), managers, developers, testers, and those who maintain the system.
- Each stakeholder has a different perspective on and use for the SRS.

Software Stakeholders and Their Uses of the SRS

| Stakeholder | Use |
|---|---|
| Customers | Express how their needs can be met. They continue to do this throughout the process as their perceptions of their own needs change. |
| Managers | Bid on the system and then control the software production process. |
| Developers | Create a software design that will meet the requirements. |
| Test engineers | A basis for verifying that the system performs as required. |
| Maintenance engineers | Understand what the system was intended to do as it evolves over time. |

- **Requirements traceability** is concerned with the relationships between requirements, their sources, and the system design.
- During the requirements engineering process, the requirements must be identified to assist in planning for the many changes that will occur throughout the software life cycle.
- **Requirements engineering** is the process of eliciting, documenting, analyzing, validating, and managing requirements.
- Traceability is also an important factor in conducting an impact analysis to determine the effort needed to make such changes.

# Characteristics of good requirements

**Clear**

- Good requirements are clear, concise, and easy to understand.
- That means they can’t be pumped full of management-speak, and confusing jargon.
- It is okay to use technical terms and abbreviations if they are defined some where or they are common knowledge in the project's domain.
- In short, requirements cannot be vague or ill-defined.
- Each requirement must state in concrete

**Unambiguous**

- A requirement must be unambiguous.
- If the requirement is worded so that we can't tell what it requires, then we can't build a system to satisfy it.
- Although this may seem like an obvious feature of any good requirement, it's sometimes harder to guarantee than we might think.
- As we write requirements, do our best to make them unambiguous.
- Read them carefully to make sure we can't think of any way to interpret them other than the way we intend.

**Consistent**

- A project's requirements must be consistent with each other.
- That means not only that they cannot contradict each other, but that they also don't provide so many constraints that the problem is unsolvable.
- Each requirement must also be self-consistent.(In other words, it must be possible to achieve.)

**Prioritized**

- When we start working on the project's schedule, it's likely we need to cut a few nice-to-haves from the design.
- We might like to include every feature but don't have the time or budget, so something's got to go.
- At this point, we need to prioritize the requirements.
- If we have assigned costs (usually in terms of time to implement) and priorities to the requirements, then we can defer the high-cost, low-priority requirements until a later release.

**Verifiable**

- Requirements must be verifiable.
- If we can’t verify a requirement, how do we know whether we have met it?
- Being verifiable means the requirements must be limited and precisely defined.

The  characteristics of good requirements. They are as follows:

- **Correct** — The SRS should correctly describe the system behavior.
- **Unambiguous** — An unambiguous SRS is one that is clear and not subject to different interpretations. Using appropriate language can help avoid ambiguity.
- **Complete** — An SRS is complete if it completely describes the desired behavior.
- **Consistent** — One requirement must not contradict another
- **Ranked** — An SRS must be ranked for importance and/or stability. Not every requirement is as critical as another. By ranking the requirements, designers will find guidance in making tradeoff decisions.
- **Verifiable** — Any requirement that cannot be verified is a requirement that cannot be shown to have been met.
- **Modifiable** — The requirements need to be written in such a way so as to be easy to change.
- **Traceable** — The SRS must be traceable because the requirements provide the starting point for the traceability chain.

# REQUIREMENT CATEGORIES

**Audience-Oriented Requirements:**

- These categories focus on different audiences and the different points of view that each audience has.
- They use a somewhat business-oriented perspective to classify requirements according to the people who care the most about them.

**Business Requirements:**

- Business requirements layout the project's high-level goals. They explain what the customer hopes to achieve with the project.

**User Requirements:**

- User requirements (which are also called stake holder requirements), describe how the project will be used by the end users.
- They often include things like sketches of forms, scripts that's how the steps users will perform to accomplish specific tasks, use cases, and prototypes.

**Functional Requirements:**

- Functional requirements are detailed statements of the project's desired capabilities.
- They are similar to the user requirements but they may also include things that the users won't see directly.

## Non-functional Requirements:

- Non-functional requirements are statements about the quality of the application's behavior or constraints on how it produces a desired result.
- They specify things such as the application's performance, reliability, and security characteristics.

## Implementation Requirements:

- Implementation requirements are temporary features that are needed to transition to using the new system but that will be later discarded.
- After we finish testing the system and are ready to use it full time, you need a method to copy any pending invoices from the old database into the new one.

# GATHERING REQUIREMENTS

- **Listen to Customers (and Users):**
  - Start by listening to the customers.
  - Learn as much as we can, about the problem they are trying to address and any ideas.
- **Brain Storming** – is a group activity exercise which helps to develop creative solutions.
- **Use the Five Ws (and One H):**
  - **Who :** ask who will be using the software and get to know as much as we can about those people

- **What :-** Figure out what the customers need the application to do.
- **When :-** Find out when the application is needed.
- **Where :-** Find out where the application will be used. Will it be used on desktop computers in an air-conditioned office? Or will it be used on phones in a noisy subway?
- **Why :-** Ask why the customers need the application. Use the “why” question to help clarify the customers’ needs and see if it is real.
- **How :-**We shouldn’t completely ignore the customers’ ideas. Sometimes, customers have good ideas