Domain Analysis & Requirements

Domain Analysis

To understand the Problem one needs to understand the Domain

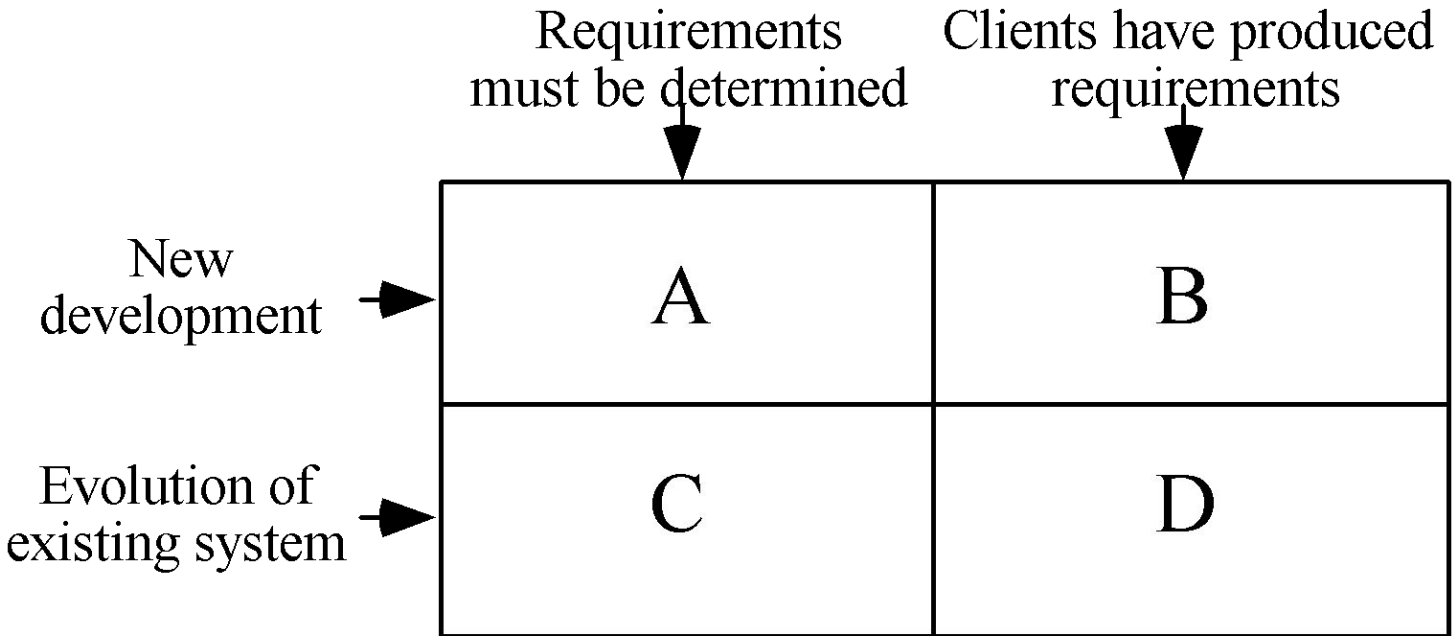
domain

domain expert

domain analysis document – general knowledge, customers, environment, current tasks and procedures, competing software ...

Where are we to begin with?

Which type is the project?



Problem Definition and Scope

Express problem as:

A *difficulty* the users or customers are facing, OR

An *opportunity* that will result in some benefit such as improved productivity or sales.

A good problem statement is short

Define early, with scope for revision

Narrowing Scope

Narrow the *scope* by defining a more precise problem

Exclude some of these things if too broad

Determine high-level goals if too narrow

Avoid Scope Creep - Kitchen Sink Syndrome

Happens when a project's requirements, goals, or vision changes beyond what was originally agreed upon.

How to have an "original agreement"?

Avoid “Gold Plating”

This happens

- when someone works on a product or task beyond the point of diminishing returns
- when you keep tweaking / ‘improving’ something even after finishing it because you think you might be able to offer some value-addition

Requirement

A statement about

- what the proposed system will do
- that all stakeholders agree
- must be made true in order for the customer's problem to be adequately solved

Short and concise piece of information (sentence, diagram)

Says something about the system (its tasks)

All the stakeholders have agreed that it is valid

(read, understand, review, negotiate, agree, APPROVE)

It helps solve the customer's problem

A collection of requirements is a *requirements document*.

Types of Requirements

Functional

Describe *what* the system should do

Non-Functional

C*onstraints* that must be adhered to during development

Functional

1 What *inputs* the system should accept

2 What *outputs* the system should produce

3 What data the system should *store* that other systems might use

4 What *computations* the system should perform

5 The *timing and synchronization* of the above

Functional

That is, addresses both humans and systems

- Everything that a user of the system needs to know regarding its functionality
- Everything that is relevant to any other system which may interface with this system

EVENTS NITC

Functional requirements for the above problem

About 25

For each one state which of 1, 2, 3, 4, 5 it addresses

EVENTS v1 / v2

Of these 25 requirements identify each to one of the following.

Suitably prioritise the following requirements by marking them as Mandatory(M), Nice to Have(N), Superfluous(S).

Domain Analysis

Feasibility Study

Project Type: A, B, C or D

Problem Definition

Scope

Requirement [Review]

Attendance - in Offline classes

What is the problem?

What is your solution?
Or even before,
What is your approach to solving?

Attendance: Which one do we go for?

- Call / Sign
- ID card
- PIN
- ID Card + PIN
- Fingerprint (Biometric)
- A couple of pictures

Non Functional Requirements

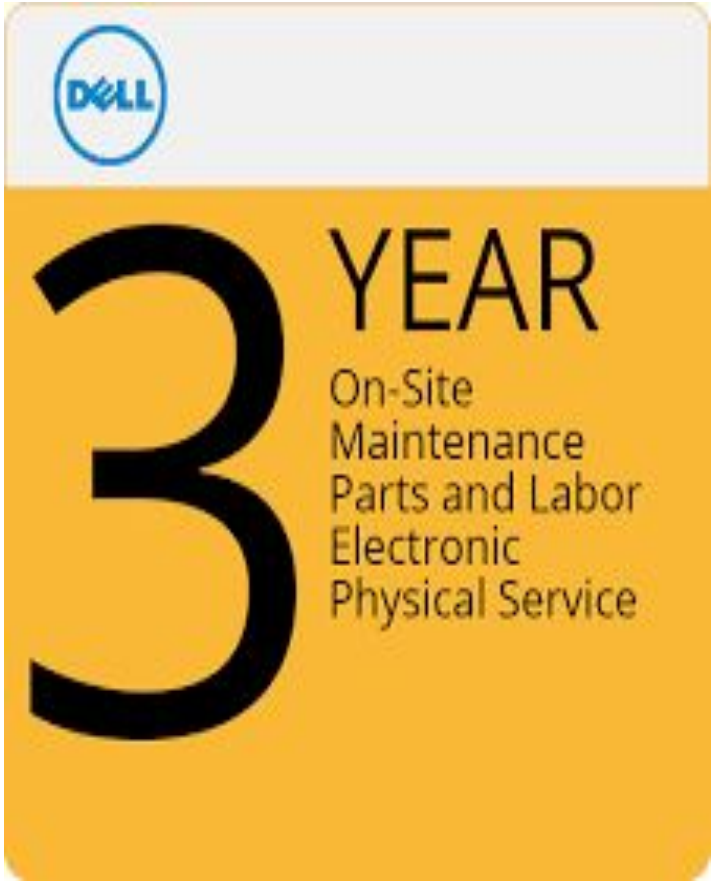
Why is it important?

Non Functional Requirements

Non Functional Requirements

- Constraints
- Verifiable, generally Measurable
- Constraints on
  - Design to achieve level of quality required
  - Environment and Technology
  - Project plan and development

Laptop Purchase


DELL
3
YEAR
On-Site
Maintenance
Parts and Labor
Electronic
Physical Service


HP Care Pack

योगक्षेमं वहाम्यहम्

Constraints on Design

To achieve usability, efficiency, reliability, maintainability and reusability

Response time* (result in xx milli seconds)

Throughput (transactions per minute)

Resource usage (RAM size and CPU %)

Reliability (MTBF)

Availability (downtime) – HDD - *how to measure* - hot swap

Recovery from failure (MTTR and limit on loss of data*)

Future works for maintainability and enhancement

Reusability (Percentage)

Constraints on Environment Technology

- Platform
  - Hardware
  - OS
- Technology
  - Language
  - DB

Constraints on Project Plan and Process

- Development Methodology
  - For quality
  - No details in the document
- Cost and Time
  - Budget
  - What to expect, and when
  - Other catchy clauses
  - Not in requirements, but in “Contract”

Techniques for gathering and analysing requirements

Gathering information

Observation

Interview

Brainstorming

Rapid Prototyping

Observation

Shadowing users to see what they are doing

Talk while they work

Videotaping

Points of conflicts

The extras, bargains

Useful in complex systems

Interviewing

Spread over time / Preparations

Specific details – boundary conditions - Limits

Vision – plans for future

Alternative ideas

Minimum acceptable level

Diagrams / Pictures / Scenes of real life

Empathy

Brainstorming

Moderated

Representatives of stakeholders (how to choose?)

Five – Fifteen

Trigger question

Ideas round the table – Flip chart

Voting to prioritise

Joint Application Design, Six Thinking Hats

Prototyping

Rapidly implemented

Limited functionalities

For Feedback, Early testing

Paper prototyping

UI using rapid prototyping languages

Throwaway vs. Incrememental

Our Approach

- Screens on Paper
- Inputs - Workflows - Reports
- Use Case Diagrams

Use Cases

How the user will use the system?

A *use case* is a typical sequence of actions that a user performs in order to complete a given task

Use case analysis models the system from the point of view of how users interact with this system when trying to achieve their objectives.

A *use case model* consists of

a set of use cases

an optional description or diagram indicating how they are related

Use Case

In general, a use case should cover the *full sequence of steps* from the beginning of a task until the end.

A use case should describe the *user's interaction* with the system and not the computations the system performs.

A use case should be written so as to be as *independent* as possible from any particular user interface design.

A use case should only include actions in which the actor interacts with the computer.

Scenarios

A scenario is an *instance* of a use case

- It expresses a *specific occurrence* of the use case
  - — a specific actor ...
  - — at a specific time ...
  - — with specific data.

How to describe a single use case

A. Name: Give a short, descriptive name to the use case.

B. Actors: List the actors who can perform this use case.

C. Goals: Explain what the actor or actors are trying to achieve.

D. Preconditions: State of the system before the use case.

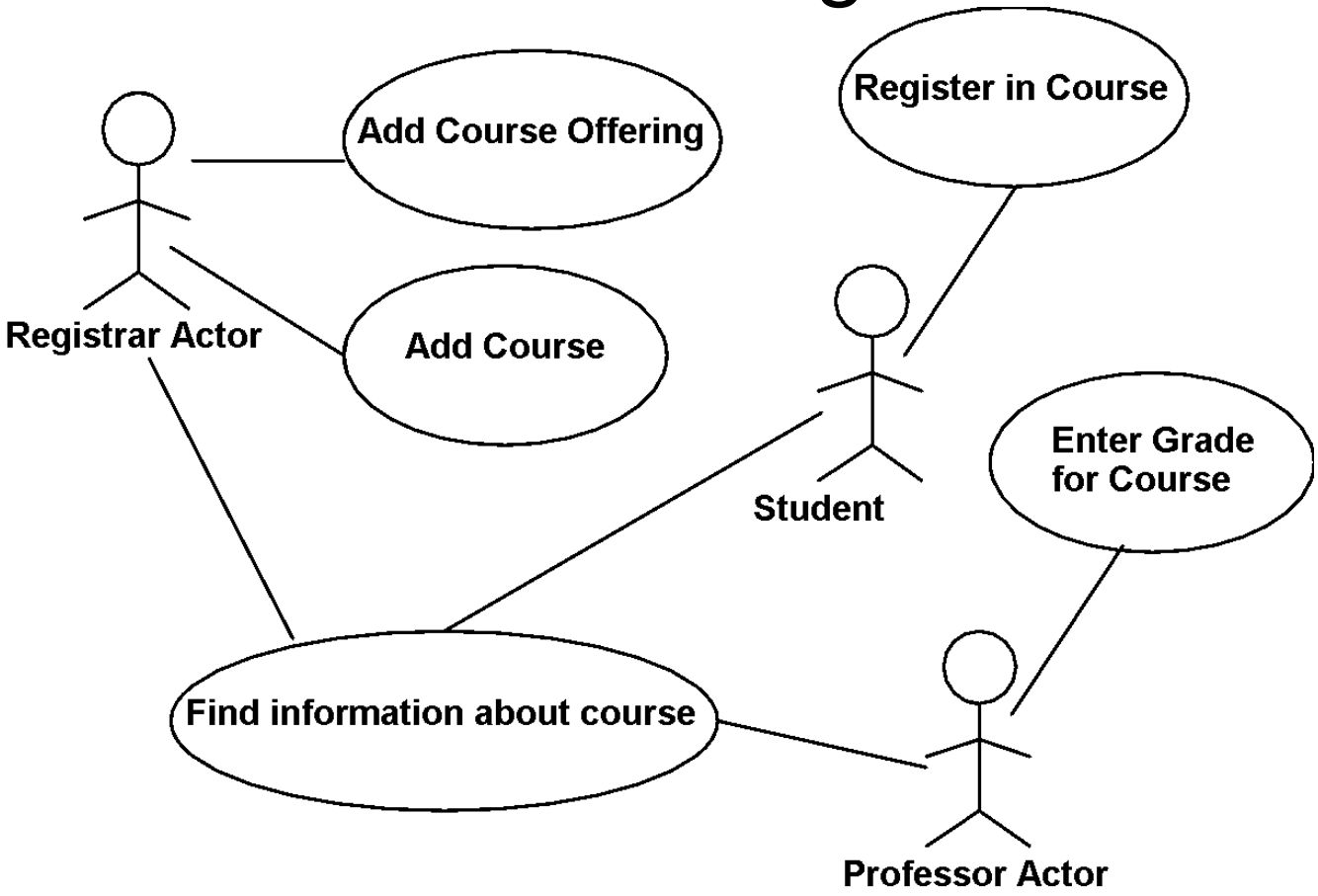
E. Description: Give a short informal description.

F. Related use cases.

G. Steps: Describe each step using a 2-column format.

H. Postconditions: State of the system in following completion.

Use case diagrams



Extensions

- Used to make *optional* interactions explicit or to handle *exceptional* cases.
- By creating separate use case extensions, the description of the basic use case remains simple.
- A use case extension must list all the steps from the beginning of the use case to the end.
  - Including the handling of the unusual situation.

Generalizations

- Much like superclasses in a class diagram.
- A generalized use case represents *several similar* use cases.
- One or more specializations provides details of the similar use cases.

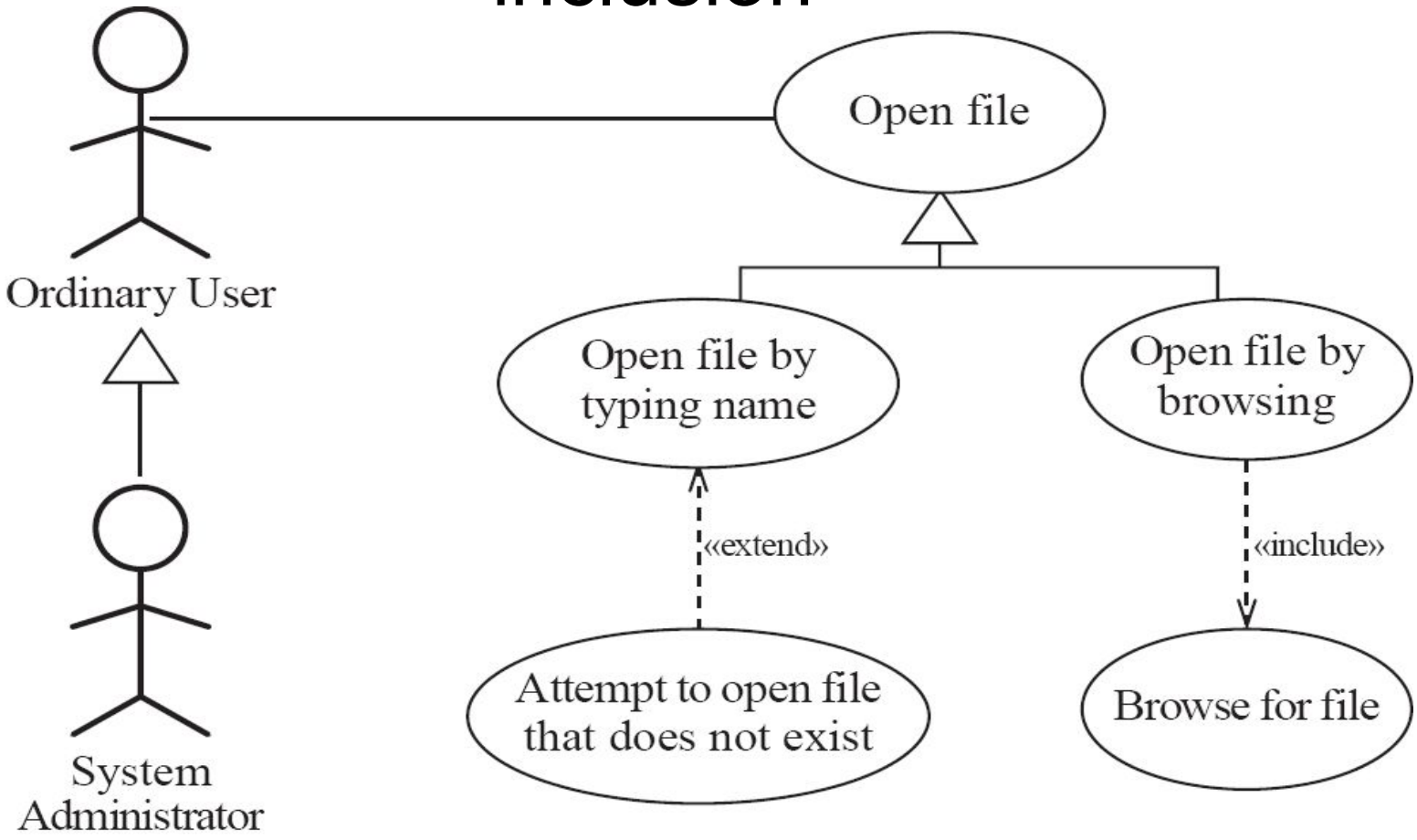
Inclusions

- Allow one to express *commonality* between several different use cases.

- Are included in other use cases
  - — Even very different use cases can share sequence of actions.
  - — Enable you to avoid repeating details in multiple use cases.

- Represent the performing of a *lower-level task* with a lower-level goal.

Example of generalization, extension and inclusion



Example description of a use case

Use case: Open file

Related use cases:
Generalization of:
- Open file by typing name
- Open file by browsing

Steps:

Actor actions	**System responses**
1. Choose 'Open…' command	2. File open dialog appears
3. Specify filename	
4. Confirm selection	5. Dialog disappears

Example (continued)

Use case: Open file by typing name

Related use cases:
Specialization of: Open file

Steps:

Actor actions	**System responses**
1. Choose 'Open…' command	2. File open dialog appears
3a. Select text field	
3b. Type file name	
4. Click 'Open'	5. Dialog disappears

Example (continued)

Use case: Open file by browsing

Related use cases:
Specialization of: Open file
Includes: Browse for file

Steps:

Actor actions	System responses
1. Choose 'Open…' command	2. File open dialog appears
3. Browse for file	
4. Confirm selection	5. Dialog disappears

Example (continued)

Use case: Attempt to open file that does not exist

Related use cases:
Extension of: Open file by typing name

Actor actions	**System responses**
1. Choose 'Open…' command	2. File open dialog appears
3a. Select text field	
3b. Type file name	
4. Click 'Open'	5. System indicates that file does not exist
6. Correct the file name	
7. Click 'Open'	8 Dialog disappears

Example (continued)

Use case: Browse for file (inclusion)

Steps:

Actor actions	**System responses**
1. If the desired file is not displayed, select a directory	2. Contents of directory is displayed
3. Repeat step 1 until the desired file is displayed	
4. Select a file	

The modeling processes: Choosing use cases on which to focus

- Often one use case (or a very small number) can be identified as *central* to the system
  - The entire system can be built around this particular use case
- There are other reasons for focusing on particular use cases:
  - Some use cases will represent a high *risk* because for some reason their implementation is problematic
  - Some use cases will have high political or commercial value

The benefits of basing software development on use cases

- They can help to define the *scope* of the system
- They are often used to *plan* the development process
- They are used to both develop and validate the requirements
- **They can form the basis for the definition of testcases**
- They can be used to structure user manuals

Use cases must not be seen as a panacea

- The use cases themselves must be validated
  — Using the requirements validation methods.

- There are some aspects of software that are not covered by use case analysis.

- Innovative solutions may not be considered.

Examples
from the Lab

A sample usecase for template purpose

Use Case #15 (View My Books)

Author – Aman Singh

Purpose – User can see how many books they have added.

Requirements Traceability – F15

Priority – High. User can check their book availability.

Preconditions – User must have to be login before see him/her added books.

Post conditions – All books will showed which added by him/her.

Actors – User.

Extends – U8.

Flow of Events

1. Basic Flow – User open the Application and login. Click on view my book. Get all details.

2. Alternative Flow – None.

Includes – None.

SRS Document - Two Extremes

An informal outline of the requirements using a few paragraphs or simple diagrams - requirements *definition*

A long list of specifications that contain thousands of pages of intricate detail - requirements *specification*

Errors happen at extremes

Factors deciding SRS type and level

- The size of the system
- The need to interface to other systems
- The readership
- The stage in requirements gathering
- The level of experience with the domain and the technology
- The cost that would be incurred if the requirements were faulty

Important

Prototype

Feedback

Document and Review

Rebuild

Why waste time and money on invisible steps?

Importance of Process, Skills, and Engineering

SRS Review

Each Requirement should

Have **benefits that outweigh the costs** of development (CBA)

Be **important** for the solution of the current problem(80-20)

Be expressed using a **clear and consistent notation**

Be **unambiguous**

Be **logically consistent**

Lead to a system of **sufficient quality**

Be **realistic** with available resources

Be **verifiable******

Be uniquely **identifiable** (Section Number)

Does not over-constrain the design of the system (Avoid How)

SRS Review Points

Sufficiently complete with functional and non-functional requirements and well organized

Provide rationale

Agreed to by all the stakeholders – Contract

Change Management in Requirements

Requirements change because:

Business process changes

Technology changes

The problem becomes better understood

The never ending process:

Continued Interaction

CBA

Small vs. Large

Forcing unexpected changes into a partially built system will probably result in a poor design and late delivery

Some changes are enhancements in disguise

Avoid making the system *bigger*, only make it *better*

Software also becomes old and die (mercy killing?)

Difficulties, Risks in Requirement Analysis

Lack of understanding of the domain or the real problem

Do domain analysis and prototyping

Requirements change rapidly

Perform incremental development, build flexibility into the design, do regular reviews

Attempting to do too much

Document the problem boundaries at an early stage, carefully estimate the time

It may be hard to reconcile conflicting sets of requirements

*Brainstorming, JAD (***facilitator, end users, developers, observers, mediators and experts)** *sessions, competing prototypes*

It is hard to state requirements precisely

Break requirements down into simple sentences and review them carefully, look for potential ambiguity, make early prototypes

SAMPLE SRS - Borrow & Lend

A few UC diagrams and Use Cases

READ the ANTI-SRS article

Object-Oriented

Software Engineering

Practical software development

using

UML

&

Java

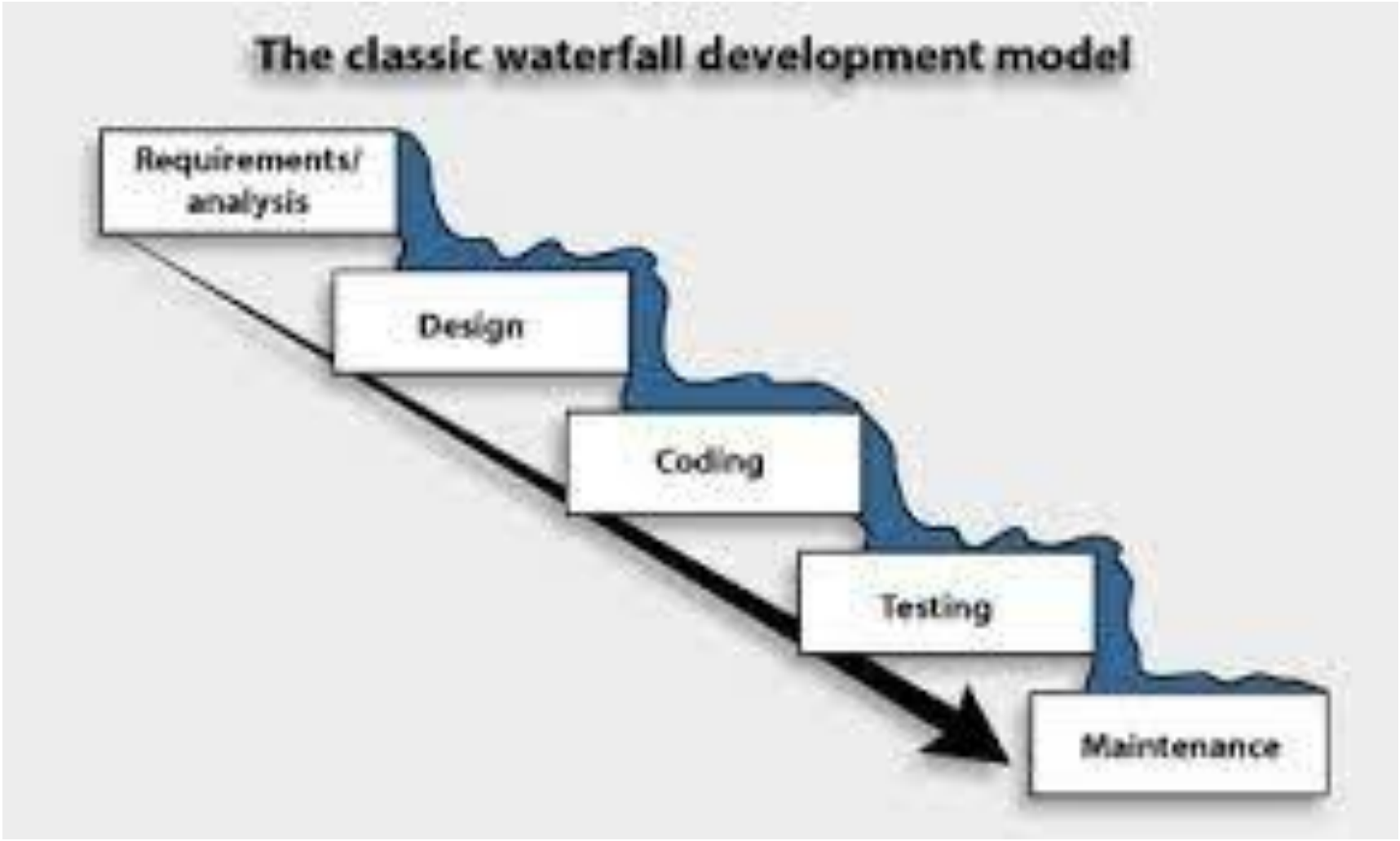
Different Process Models

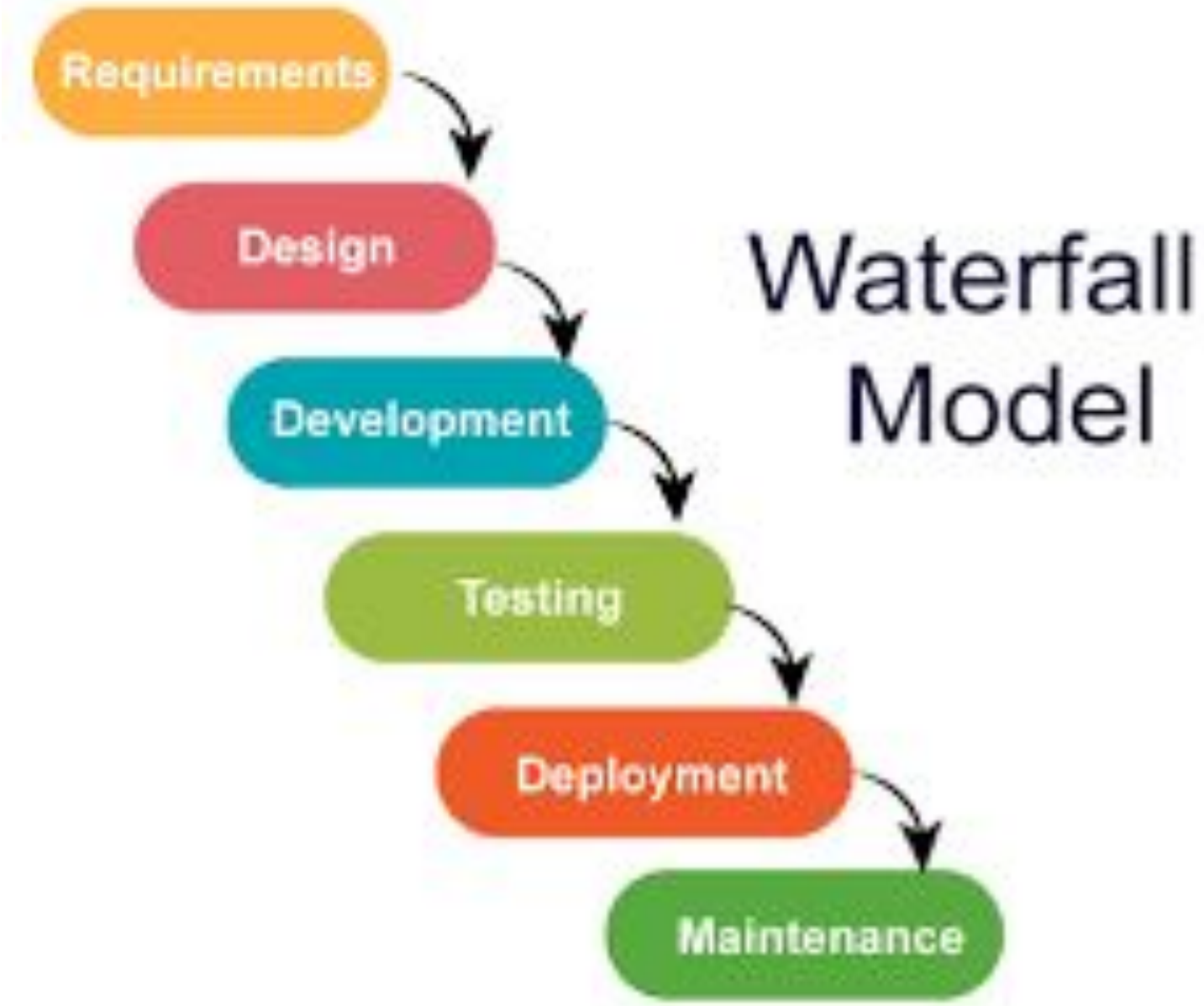
specification-oriented,

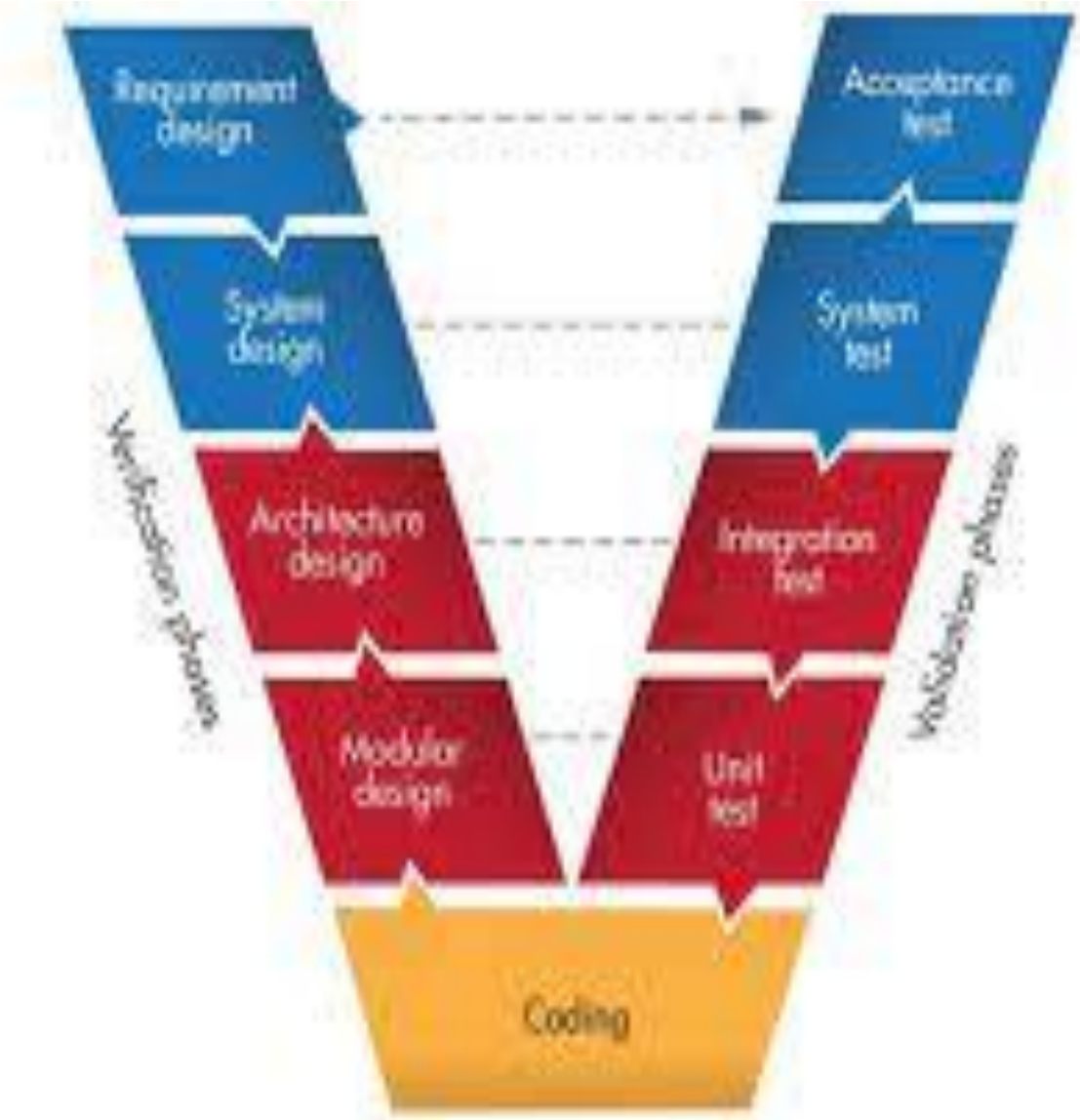
prototype-oriented,

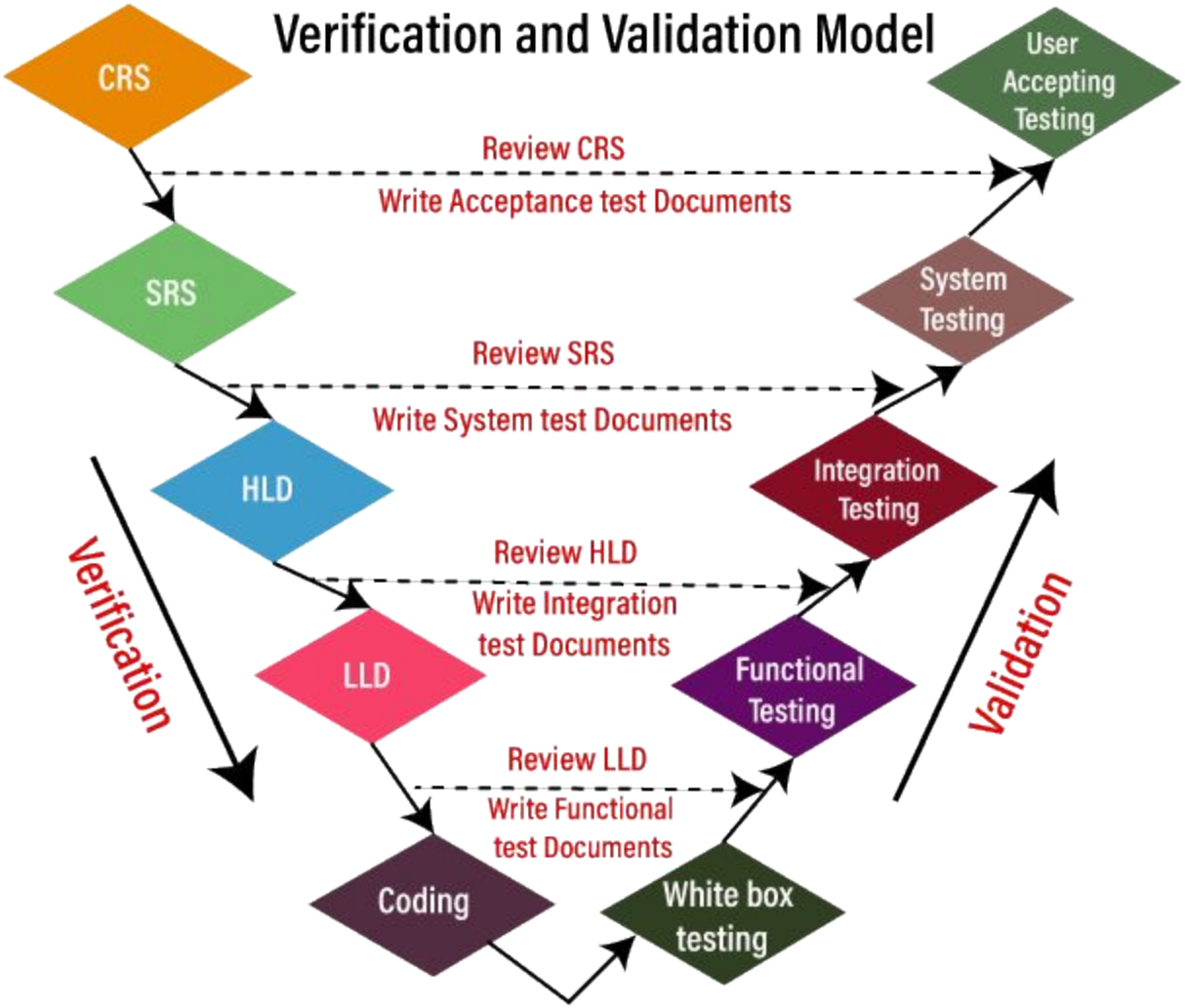
simulation-oriented,

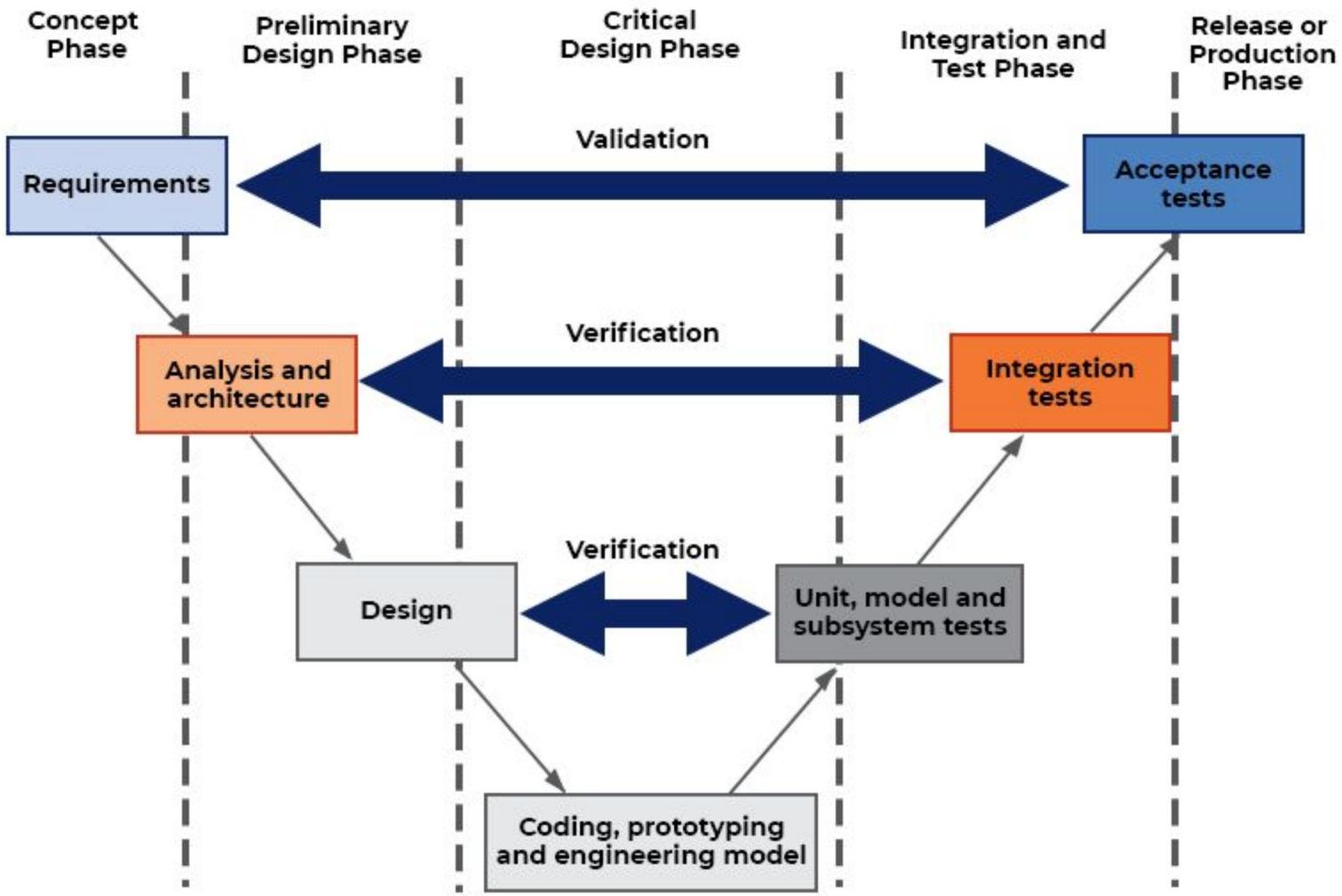
automatic transformation-oriented,


The classic waterfall development model
Requirements/ analysis
Design
Coding
Testing
Maintenance

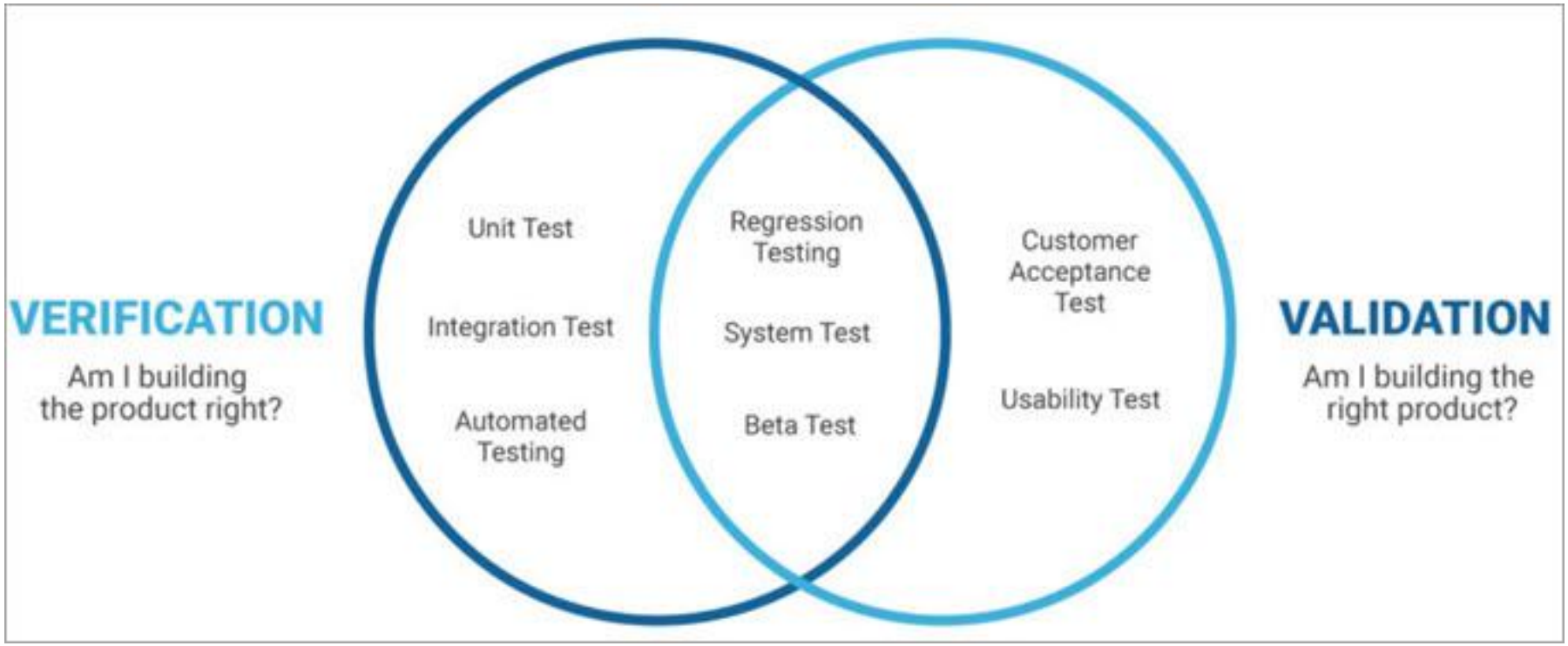

Requirements
Design
Development
Testing
Deployment
Maintenance
Waterfall Model


Requirement design
Acceptance test
System design
System test
Architecture design
Integration test
Modular design
Unit test
Coding
Verification phases
Validation phases


Verification and Validation Model
CRS
User Accepting Testing
Review CRS
Write Acceptance test Documents
SRS
System Testing
Review SRS
Write System test Documents
HLD
Integration Testing
Review HLD
Write Integration test Documents
LLD
Functional Testing
Review LLD
Write Functional test Documents
Coding
White box testing
Verification
Validation


Concept Phase
Preliminary Design Phase
Critical Design Phase
Integration and Test Phase
Release or Production Phase
Validation
Requirements
Acceptance tests
Verification
Analysis and architecture
Integration tests
Verification
Design
Unit, model and subsystem tests
Coding, prototyping and engineering model

Verification	Validation
Are we implementing the system right?	Are we implementing the right system?
Evaluating products of a development phase	Evaluating products at the closing of the development process
The objective is making sure the product is as per the requirements and design specifications	The objective is making sure that the product meets user's requirements
Activities included: reviews, meetings, and inspections	Activities included: black box testing, white box testing, and grey box testing
Verifies that outputs are according to inputs or not	Validates that the users accept the software or not
Items evaluated: plans, requirement specifications, design specifications, code, and test cases	Items evaluated: actual product or software under test
Manual checking of the documents and files	Checking the developed products using the documents and files


VERIFICATION
Am I building the product right?
Unit Test
Integration Test
Automated Testing
Regression Testing
System Test
Beta Test
Customer Acceptance Test
Usability Test
VALIDATION
Am I building the right product?

Software Verification

Software verification provides objective evidence that the design outputs of a particular phase of the software development life cycle meet all of the specified requirements for that phase.

Software verification looks for consistency, completeness, and correctness of the software and its supporting documentation, **as it is being developed,** and provides support for a subsequent conclusion that software is validated.

Software testing is one of many verification activities intended to confirm that software development output meets its input requirements. Other verification activities include various static analyses, code and document inspections, walkthroughs, and reviews.

Software Validation

Software validation is a primarily part of the design validation for a finished product.

Software validation is termed as "**confirmation by examination and provision of objective evidence that software specifications conform to user needs and intended uses, and that all requirements implemented through software can be consistently fulfilled.**"

In practice, software validation activities may occur both during, as well as at the end of the software development life cycle to ensure that all requirements have been fulfilled.

UAT is the fundamental validation activity.