# Module I - **Software Engineering Models**

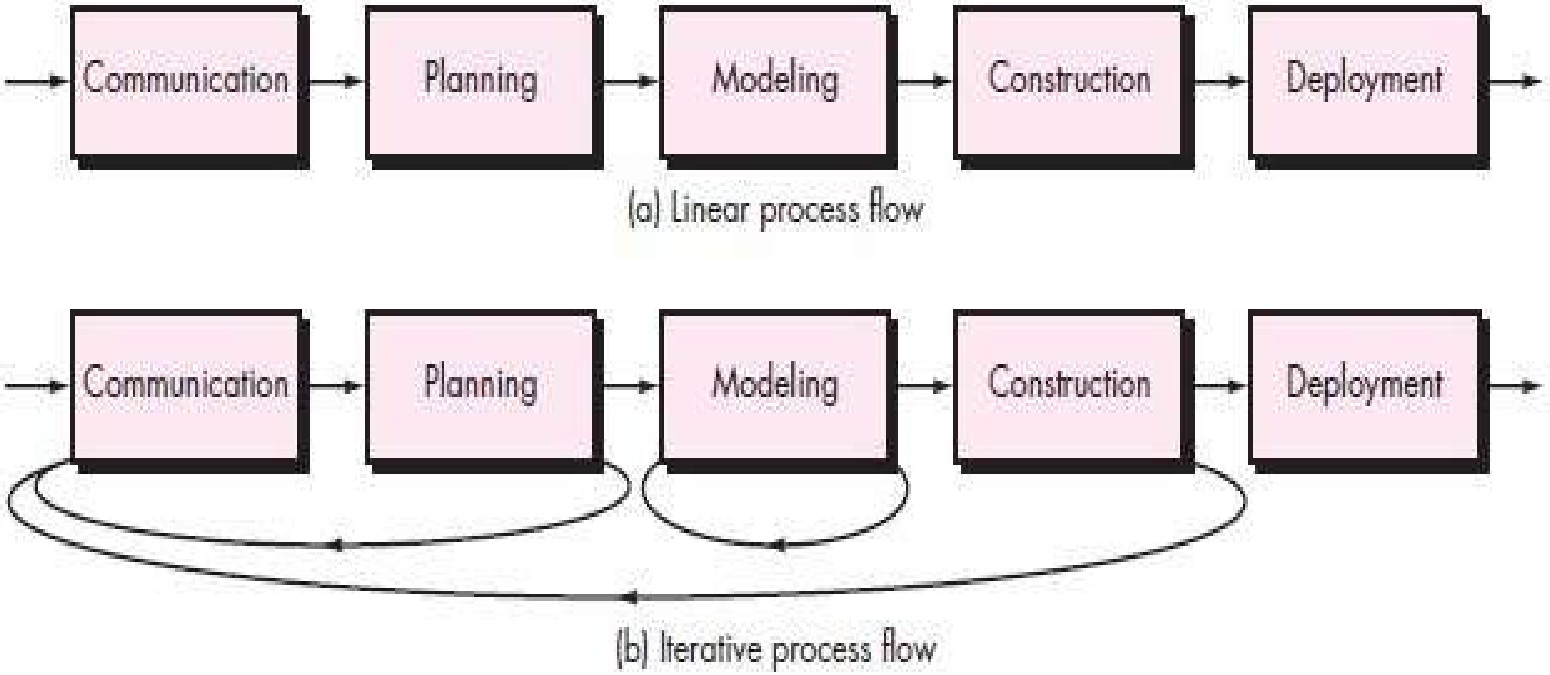
- Predictive software engineering models
- Model Approaches
- Prerequisites
- Predictive and Adaptive
  - waterfall
  - waterfall with feedback
  - Sashimi
  - Incremental waterfall
  - V model
- Prototyping and prototyping models.

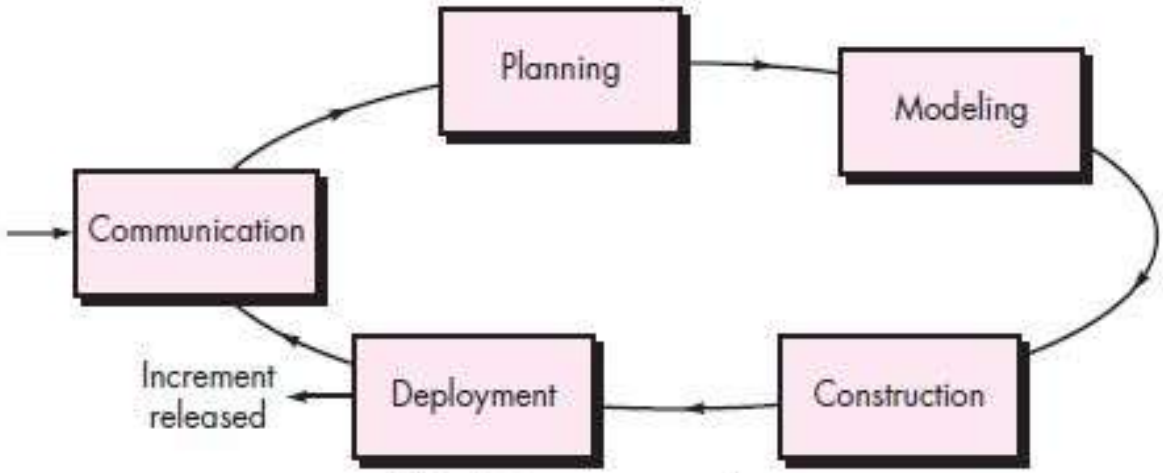
# Introduction – Process Model

- 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.

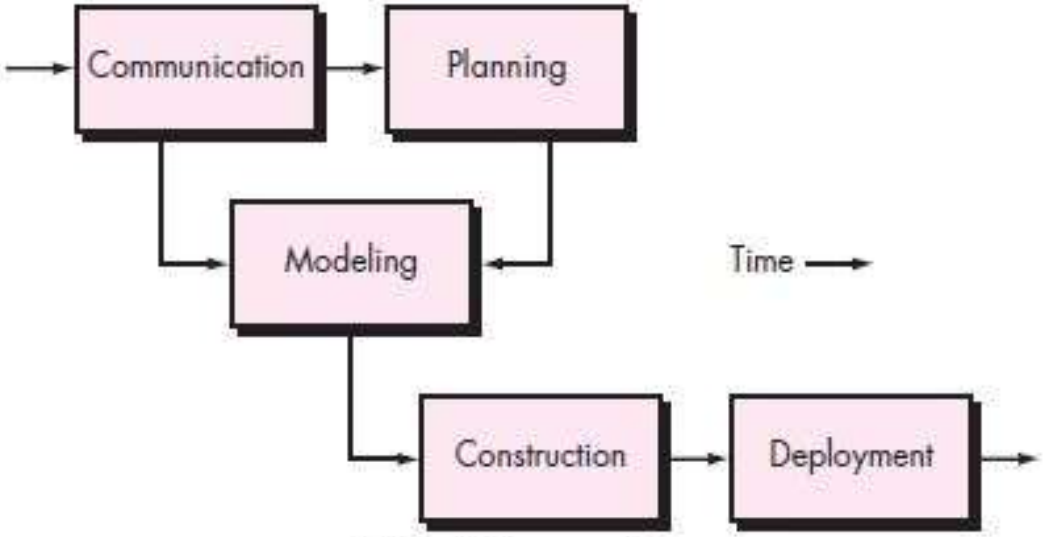
- ***Process flow*** — describes how the framework activities and the actions and tasks that occur within each framework activity are organized with respect to sequence and time

**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

- **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.

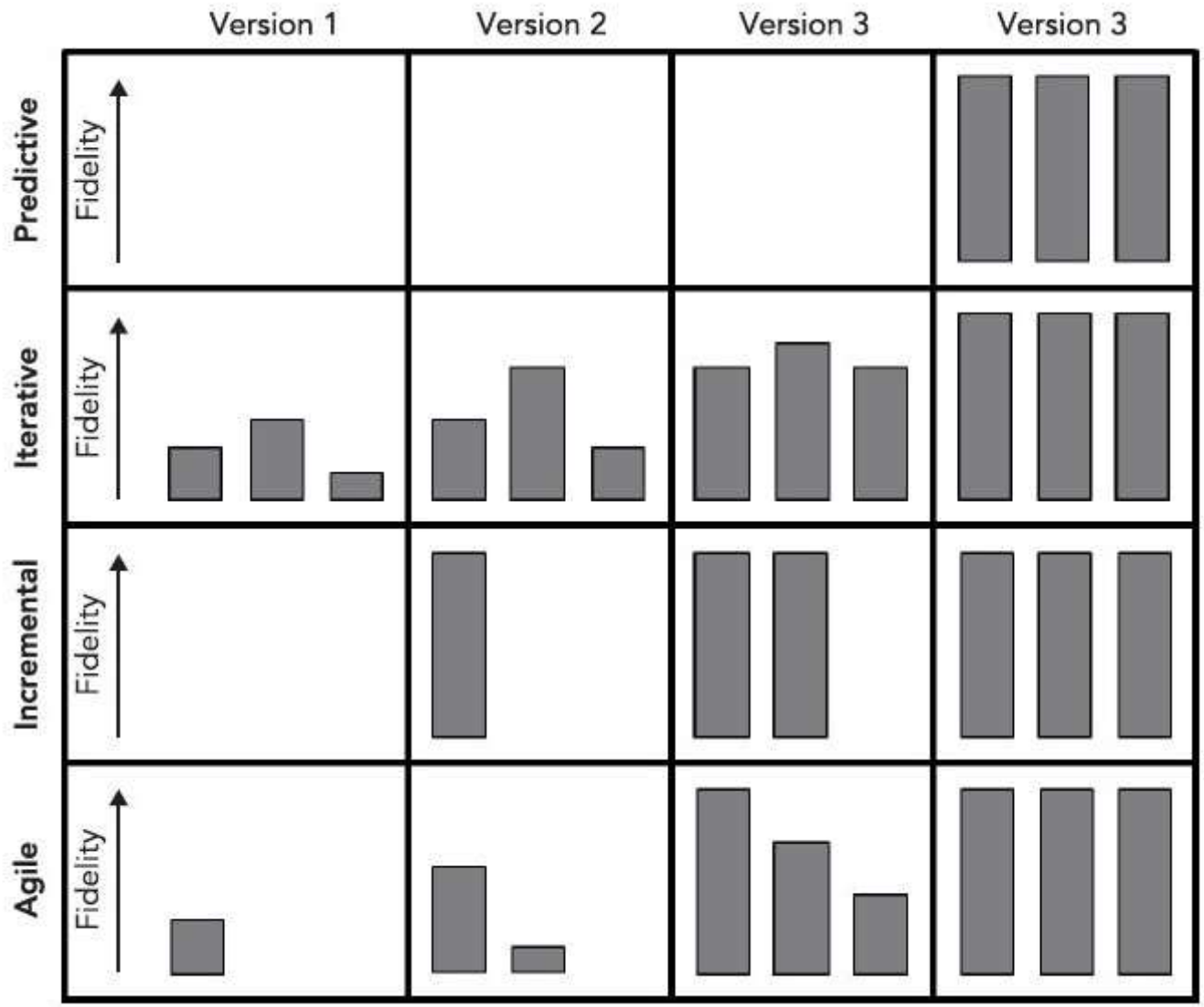
**Increment:**

- 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

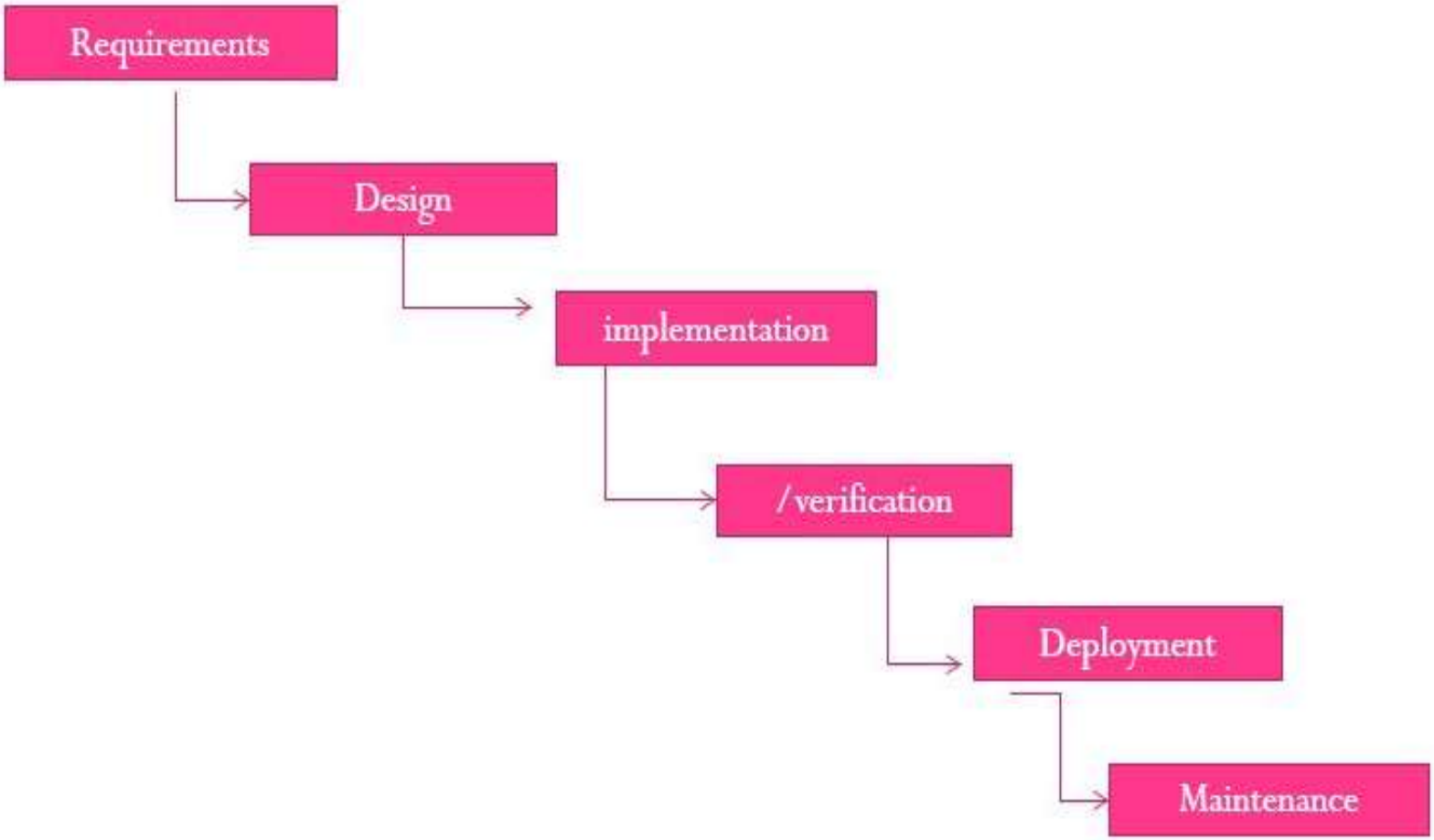
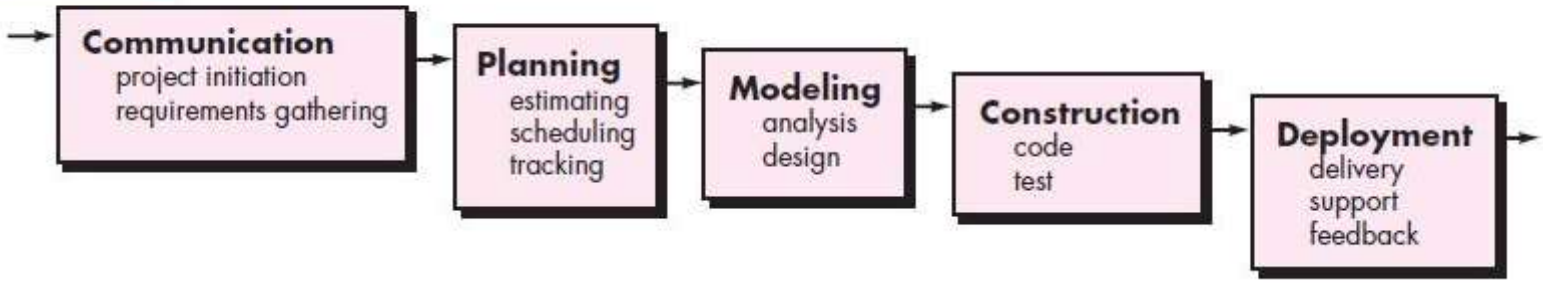


**FIGURE 2.3** The waterfall model



- **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)

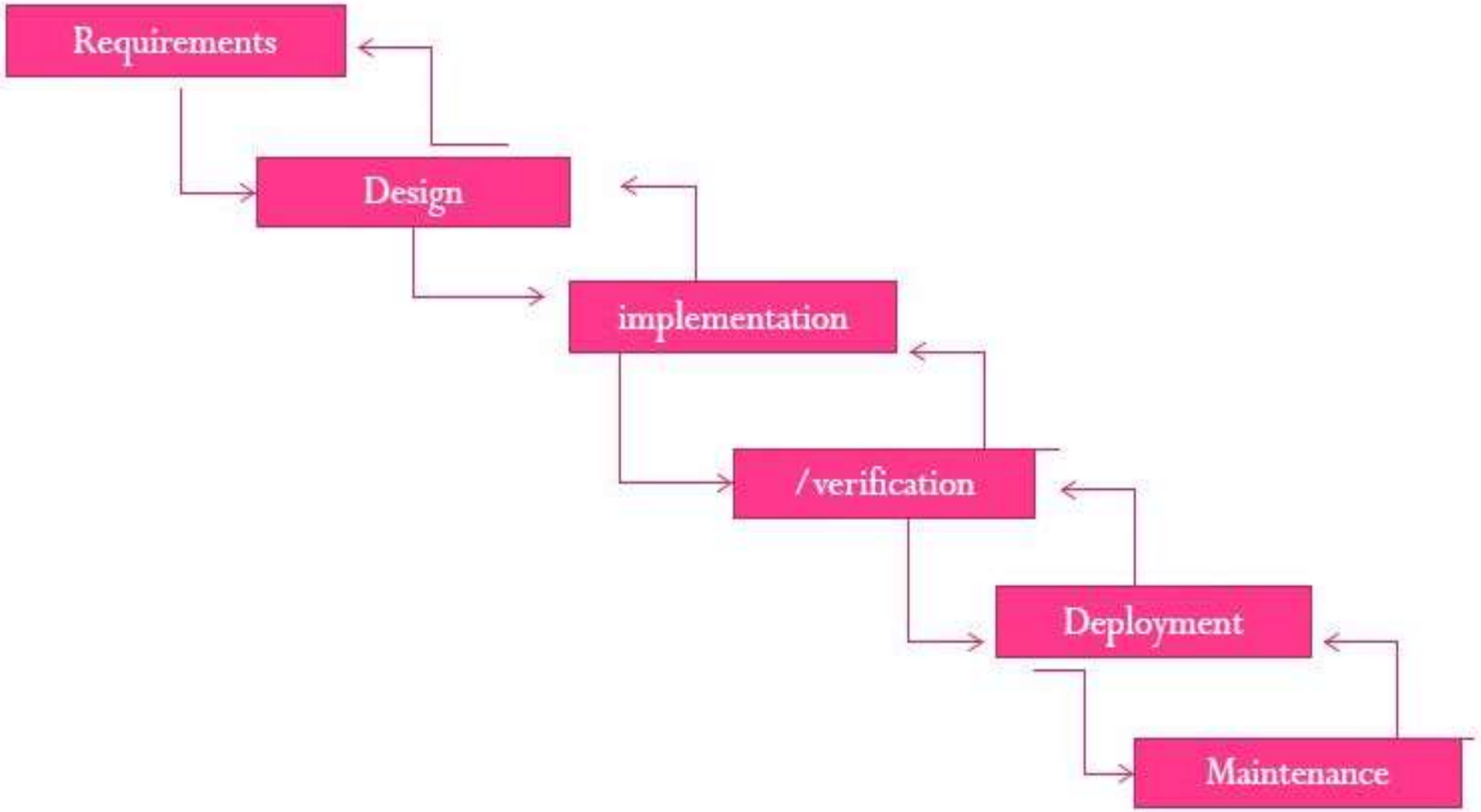
# When to use the Waterfall Model

- 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



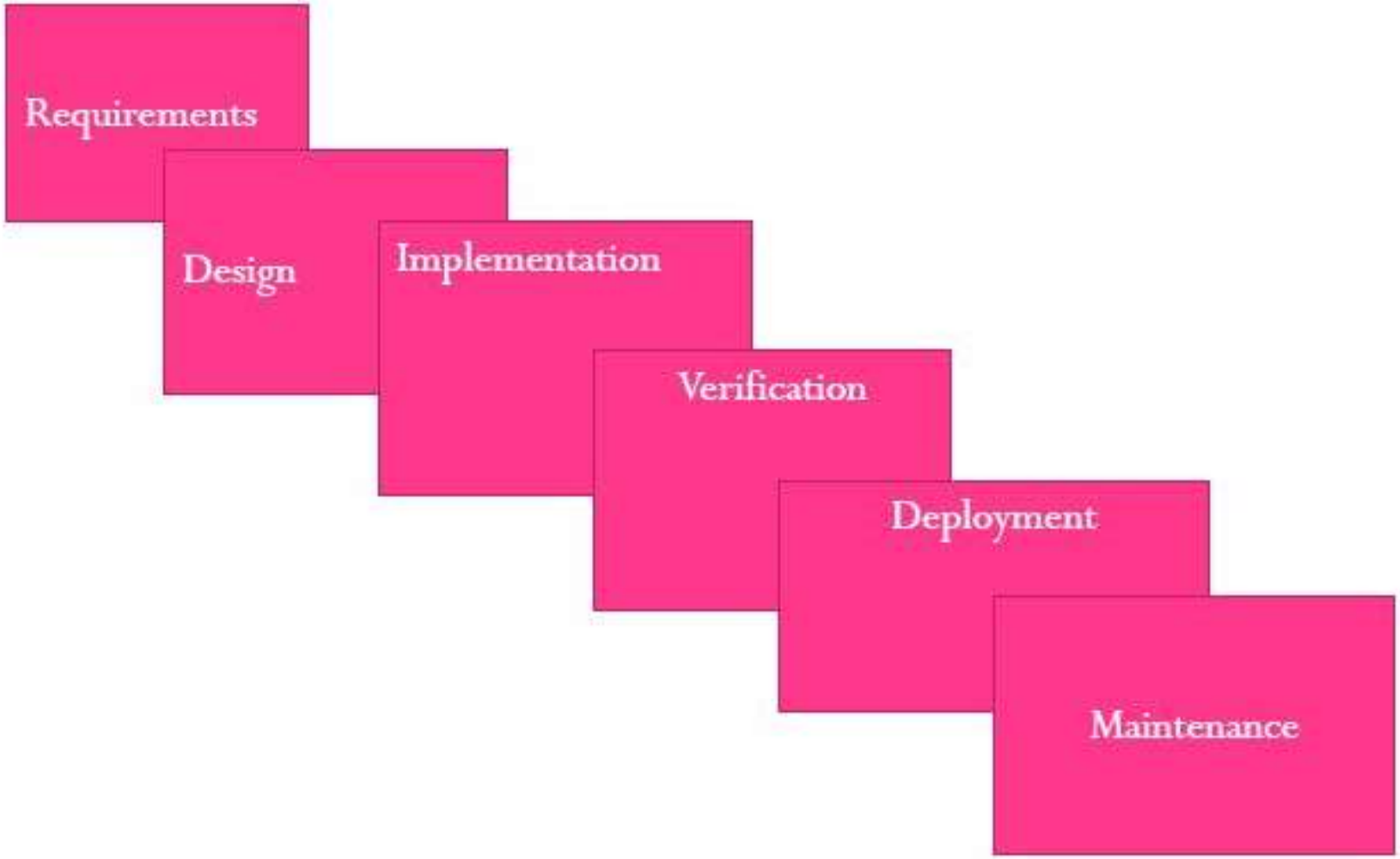
- Enables the developer to move backward from one step to the previous step.
- If we are working on design and discover that there was a problem in the requirements ,we can briefly go back to the requirements and fix them.
- In short, the waterfall model with feedback enables us to move backward from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

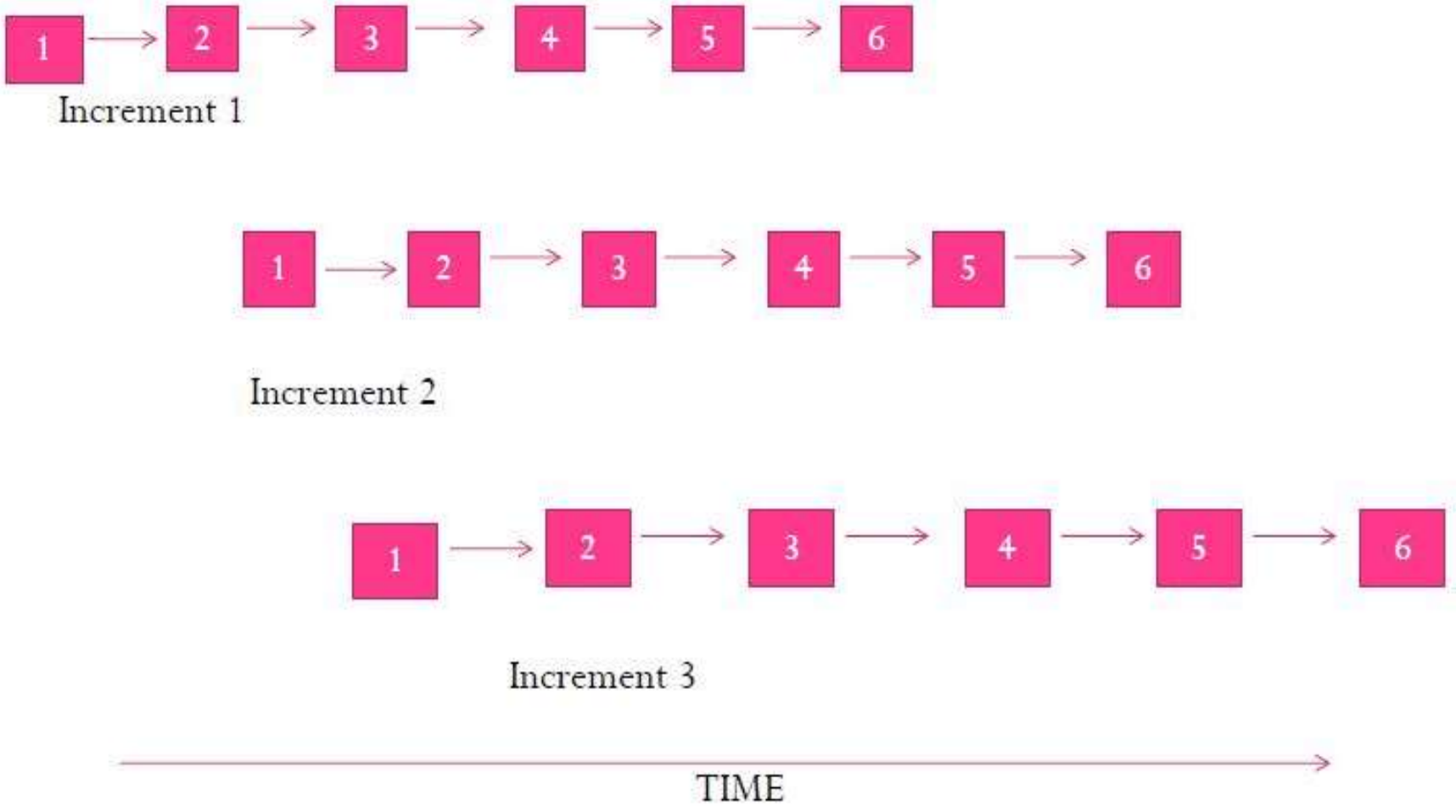
**Disadvantage:**

- 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.

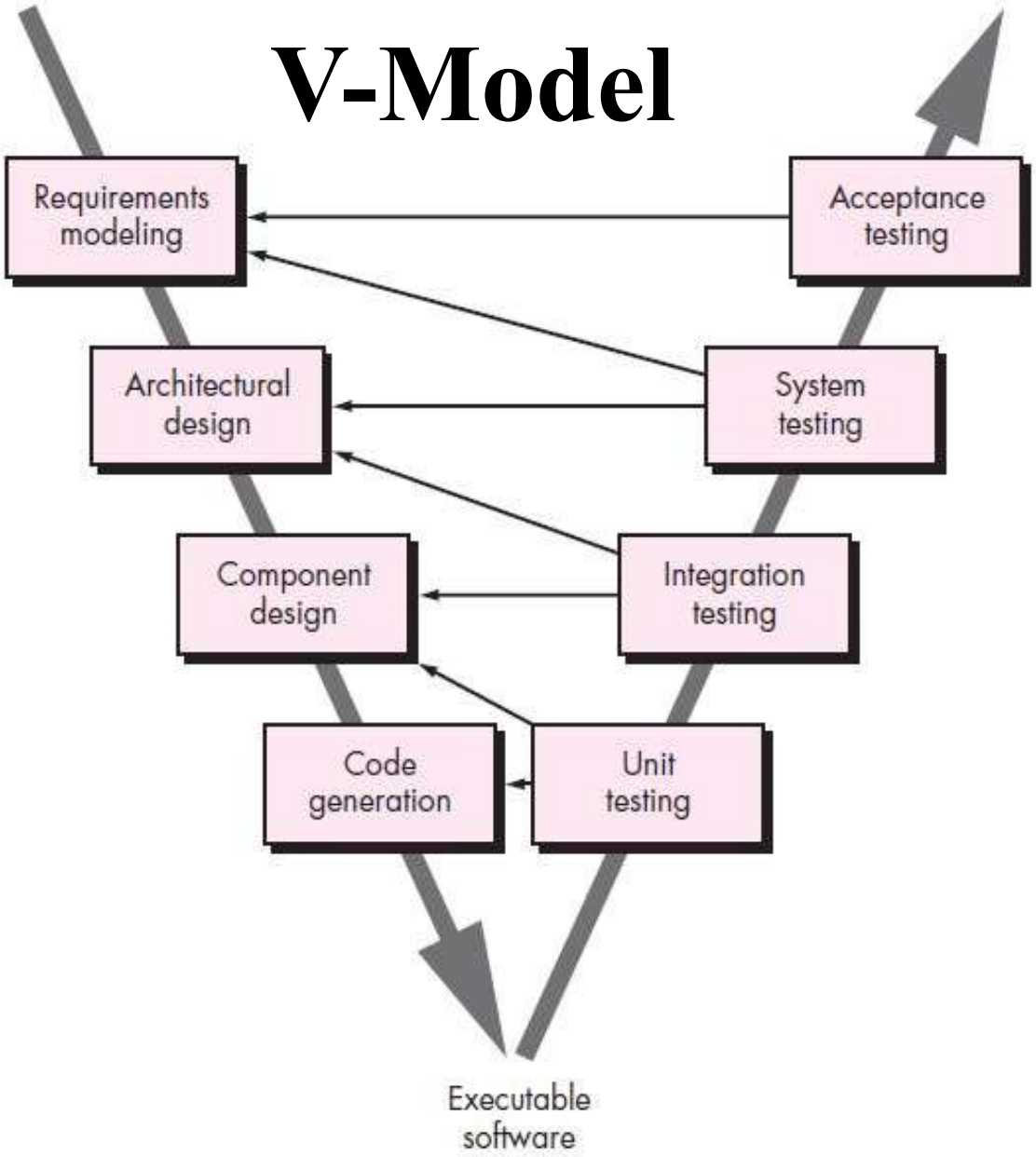
# Incremental waterfall



- 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.**

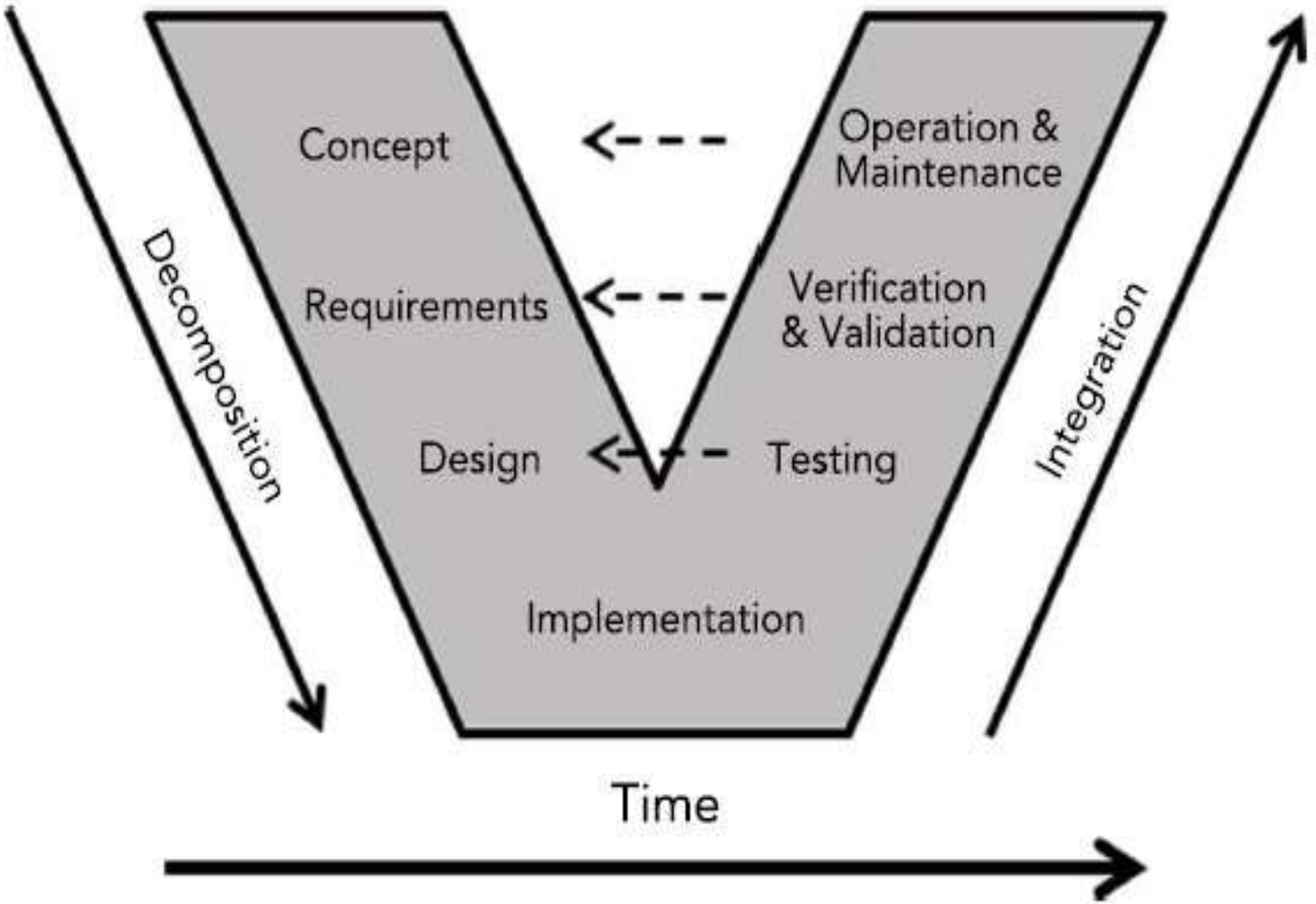
# V-Model


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.

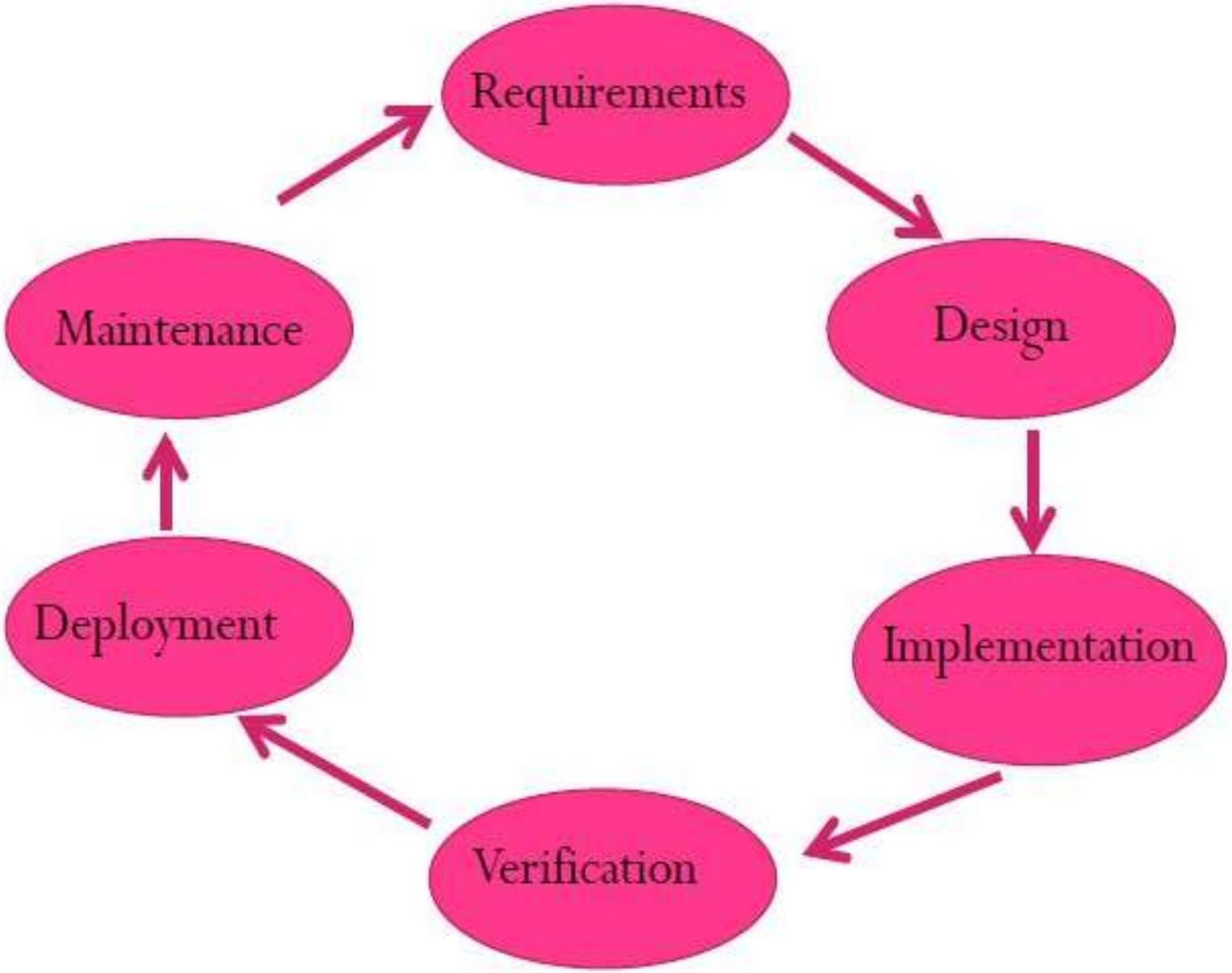
## Never- ending prototype

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

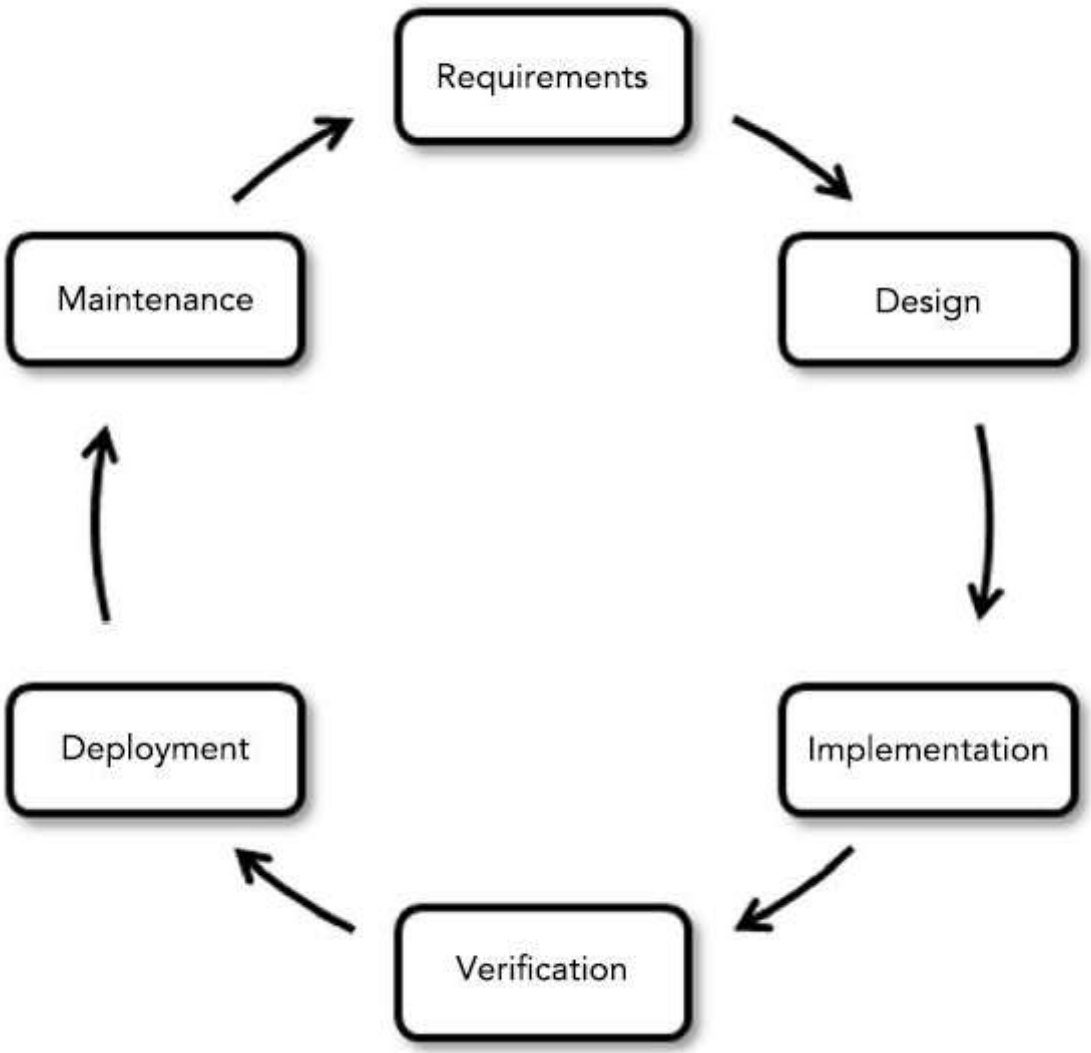
# System Development life cycle (SDLC )

- 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