20MCA107 - Advanced Software Engineering

Course Outcomes(Cos)

- Understanding of Software Engineering Principles.
- Knowledgeable about Software Industry best practices.
- Knowledgeable about advanced approaches in software design and development.
- Hands on knowledge in select tools and frameworks to work at all stages in Software development life cycle.

Why Learn Software Engineering ???

- Software Engineering applies the knowledge and theoretical understanding gained through computer science to building high-quality software products.
- 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. The typical design and development process involves these steps:
  1. Assess user needs to determine the specifications of the software.
  2. Create flowcharts and diagrams to draft the software program.
  3. Write the algorithms, or detailed instructions, that direct the computer hardware.
  4. Code these instructions in the appropriate programming language.
  5. Test the program and fix any bugs.

Continues..

- “Software Engineering is the branch of engineering that deals with the design, development, implementation and maintenance of software”.
- A practitioners of software engineering are called **Software Engineers**.
- *A software engineer applies the principles of software engineering in designing, development, maintenance and testing of software.*

Career Opportunities

- Software Developer.
- Test Engineer.
- Project Manager.
- Software Architect.
- Software Quality Manager.
- Doctoral Student/Scientist.

Advantage of being a software.

- In software we can make changes to any part of a program even after it is completely written.
- We can refine the program during development to better meet user needs.
- We can add new features to take advantage of opportunities discovered during implementation, and make modifications to meet evolving business needs.

Why is Software Engineering Important?

- Software engineering includes techniques for avoiding the many pitfalls that otherwise may lead to failure.
- It ensures the final application is effective, usable, and maintainable.
- It helps you meet milestones on schedule and produce a finished project on time and within budget.
- Software engineering gives you the flexibility to make changes to meet unexpected demands.

Goal of software engineering

- In essence
- The primary goals of software engineering are:
  - To improve the quality of the software products.
  - To increase the productivity
  - To give job satisfaction to the software engineers.

Syllabus

- **Module 1 [10 hrs]**
- 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. Class work: Prepare Software Specification Document for a moderately complex process flow system (*e.g.* Broadband fault booking and resolution system covering technical, operational and commercial aspects, covering organizational and subscriber use cases).
- **Module 2 [10 hrs]**
- Programming Style Guides and Coding Standards; Literate programming and Software documentation; Documentation generators, Javadoc, phpDocumentor.
- Version control systems basic concepts; Concept of Distributed version control system and Git; Setting up Git; Core operations in Git version control system using command line interface (CLI): Clone a repository; View history; Modifying files; Branching; Push changes, Clone operation, add, commit, log, diff commands, conflict resolution. Pushing changes to the master; Using Git in IDEs and UI based tools.
- Software Quality: Understanding and ensuring requirements specification quality, design quality, quality in software development, conformance quality.

Continues..

- **Module 3 [10 hrs]**
- OOP Concepts; Design Patterns: Basic concepts of Design patterns, How to select a design pattern, Creational patterns, Structural patterns, Behavioural patterns. Concept of Anti-patterns.
- Unit testing and Unit Testing frameworks, The xUnit Architecture, Writing Unit Tests using at least one of Junit (for Java), unittest (for Python) or phpdbg (PHP). Writing tests with Assertions, defining and using Custom Assertions, single condition tests, testing for expected errors, Abstract test.
- **Module 4 [10 hrs]**
- Concepts of Agile Development methodology; Scrum Framework.
- Software testing principles, Program inspections, Program walkthroughs, Program reviews; Blackbox testing: Equivalence class testing, Boundary value testing, Decision table testing, Pairwise testing, State transition testing, Use-case testing; White box testing: control flow testing, Data flow testing.
- Testing automation: Defect life cycle; Regression testing, Testing automation; Testing non-functional requirements.
- **Module 5[10 hrs]**
- Software Configuration Management: Using version control, Managing dependencies, Managing software configuration, Managing build and deployment environments.
- Continuous Integration: Prerequisites for continuous integration, Essential practices.
- Continuous Delivery: Principles of Software delivery, Introduction and concepts.
- Build and deployment automation, Learn to use Ansible for configuration management.
- Test automation (as part of continuous integration), Learn to set up test automation cases using Robot Framework.

Introduction

- What is a software?
- Who does it?
- Why is it important?
- What are steps?
- Software is:
  - Instructions (computer programs) that when executed provide desired features, function, and performance;
  - Data structures that enable the programs to adequately manipulate information, and
  - Descriptive information in both hard copy and virtual forms that describes the operation and use of the programs.

What is a Software?

- Software is not just the programs

  but also it includes

  all associated documentation

  configuration data

  These are needed to make the a quality software.
- A software system usually consists of a number of separate programs :
  - Configuration files – used to set up the program.
  - System documentation – which describes the structure of the system.
  - User documentation – which explains how to use the system.

Continues..

- 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:
  - 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.

Continues..

- Software has characteristics :-
  - *Software is developed or engineered; it is not manufactured in the classical sense.*
  - *Software doesn’t “wear out.”*
  - *Although the industry is moving toward component-based construction, most software continues to be custom built.*

What is Software Engineering?

- It is “a systematic approach to the analysis, design, assessment, implementation, test, maintenance and reengineering of software, that is, the application of engineering to software”.
- Some other definitions
  - IEEE definition: Software Engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.
  - It is an organized, analytical approach to the design, development, use, and maintenance of software.
  - 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.

Continues..

- The profession of software engineering consists of :-
  - Conceiving
  - Communicating
  - Specifying
  - Designing
  - Building
  - Testing
  - Maintaining
- Software engineering activities also include preparation of:
  - Documentation
  - Tool
  - Etc

Continues..

- One word for SE is – Modelling(otherwise Translation).
- Two word for SE is – Modelling and Optimization.
- Software product concept -> requirements specification -> design -> code -> automatically translated by compilers and assemblers, which produce machine executable code.
- The optimization -> economical translation.
- “Economical” means in terms of efficiency, clarity, and other desirable properties.

History

- Software development and software engineering began in the late 1950s.
- First became a profession as a result of a NATO sponsored conference on software engineering in 1968.
- Software Engineering emerged as a profession from late 1980’s and early 1990’s.

Role of Software Engineers.

- The production of software is a problem-solving activity that is accomplished by modeling.
- It’s a human activity that is biased by previous experience, and is subject to human error. The primary objective is to eliminate errors.
- Develop a code which has the following characterstics.
  - Built to be tested.
  - Designed for reuse.
  - Ready for inevitable change.

Time usage in software development.

- Approximately 10% time is used for writing codes.
- Remaining 90% time is used for following activities:-
  - Eliciting requirements.
  - Analyzing requirements.
  - Writing software requirements documents.
  - Building and analyzing prototypes.
  - Developing software designs.
  - Writing software design documents.
  - Researching software engineering techniques or obtaining informationabout the application domain.
  - Developing test strategies and test cases.
  - Testing the software and recording the results.
  - Isolating problems and solving them.
  - Learning to use or installing and configuring new software and hardware tools.
  - Writing documentation such as users manuals.
  - Attending meetings with colleagues, customers, and supervisors.
  - Archiving software or readying it for distribution.
  - Etc.

Continues..

- The activities in the previous slide may or may not be there.
- It may be in the sequential order.
- Some may recur.
- So the software engineers may specialise in small subset of these activities.

Software Engineers Knowledge Areas.

- Professionalism engineering economics
- Software requirements
- Software design
- Software construction and implementation
- Software testing
- Software maintenance
- Software configuration management
- Software engineering management
- Software engineering process
- Software engineering tools and methods
- Software quality

Characteristics of Software

- Software can be characterized by any number of qualities.
- 2 qualities are External and Internal qualities.
- External quality
  - Visible to the user –usability and reliability
- Internal qualities are those that may not be necessarily visible to the user, but help the developers to achieve improvement in external qualities.
  - Example : Good Requirement, design documentation…
- Distinction cannot be made between internal and external quality, because they are closely tied.

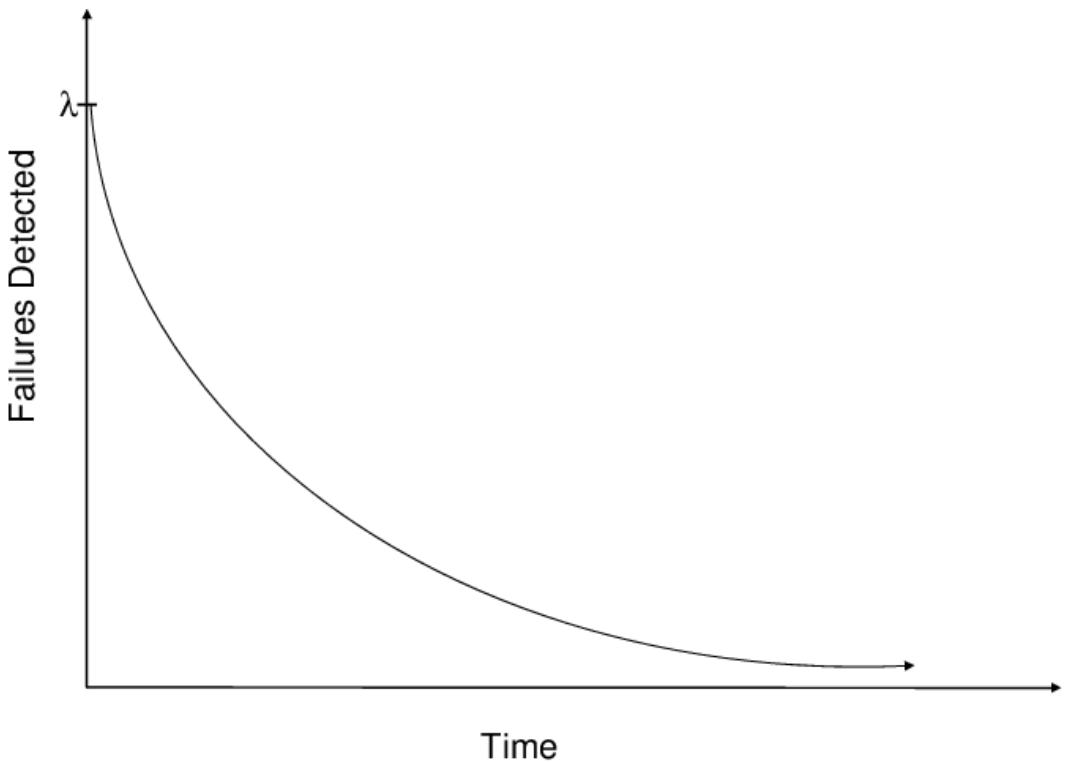
Continues..

- Reliability
  - Can the user “depend on” the software?
  - Other loose characterizations of a reliable software system include:
    - The system “stands the test of time.”
    - There is an absence of known catastrophic errors (those that disableor destroy the system).
    - The system recovers “gracefully” from errors.
    - The software is robust(strong and healthy).
  - Other informal views of reliability may be accuracy, real-time performance etc., depends on what type of system is that.

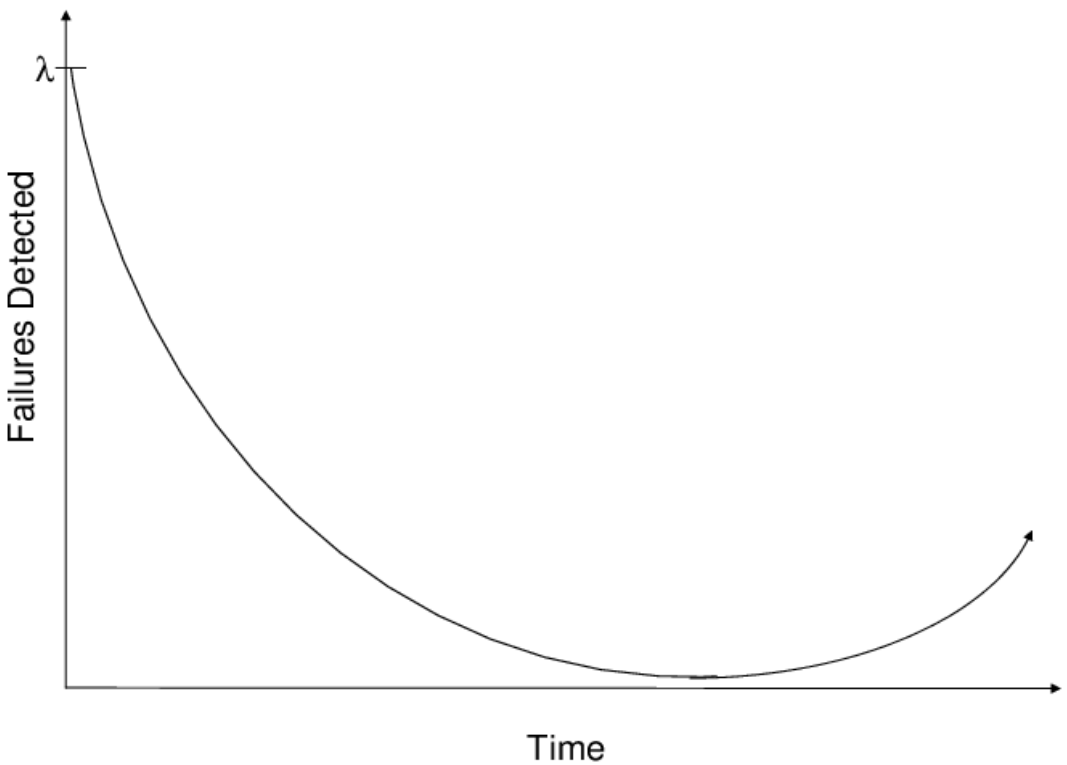
Continues..

- *Measure of Software reliability*
  - The probability that the software will operate as expected over a specified time interval.
  - Let *S* be a software system and *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* ;
    - *r(t) = P(T > t)*

Bathtub curves

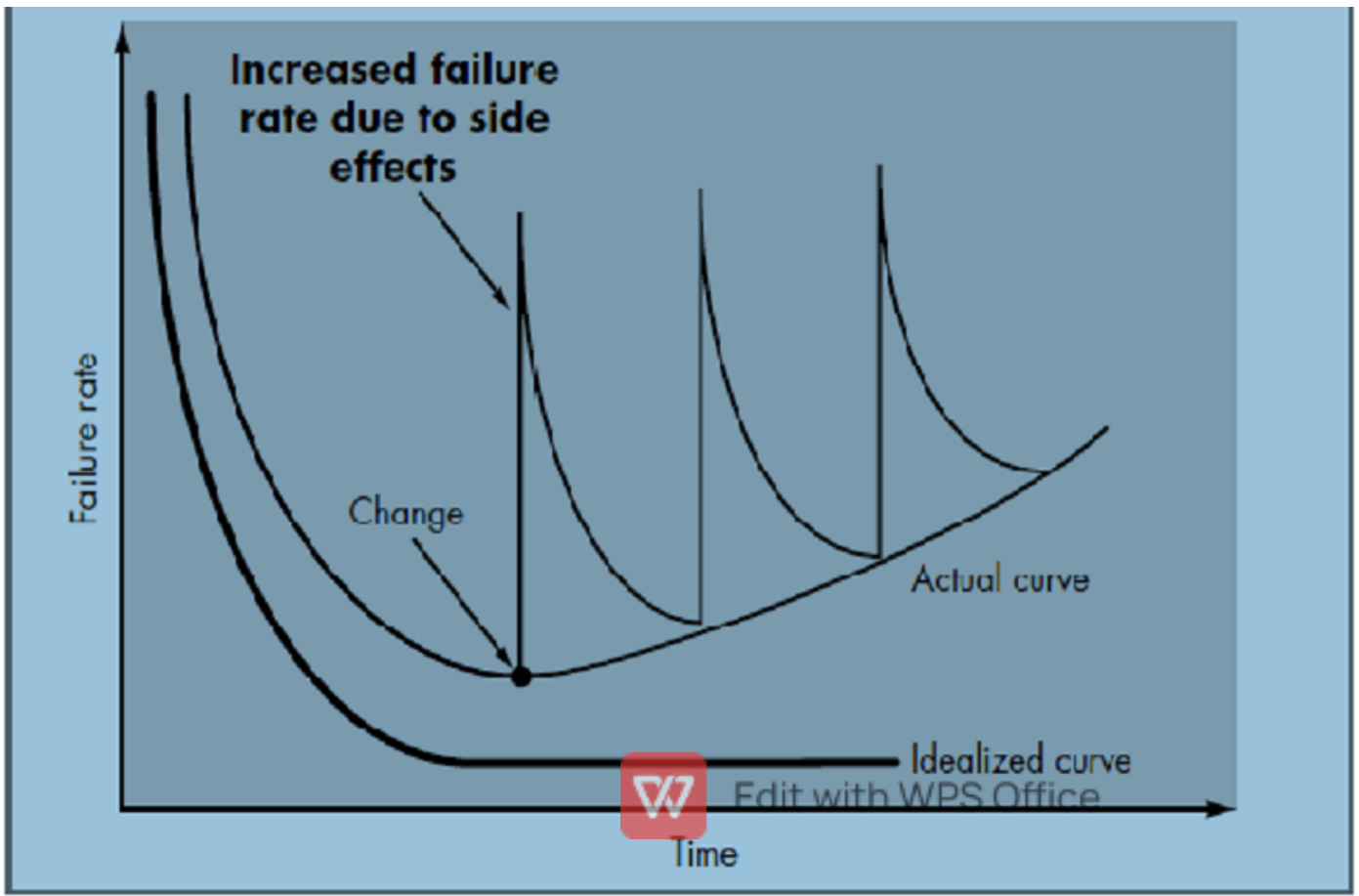


Failure function for physical components that wear out, electronics, and even biological systems.



Software failure function

Continues..


Increased failure rate due to side effects
Failure rate
Change
Actual curve
Idealized curve
Time
Edit with WPS Office

Continues..

- It is clear that a large number of errors will be found in a particular software product early, followed by a gradual decline as defects are discovered and corrected, just as in the exponential model of failure.
- Other characteristics are:
  - Correctness
    - Closely related to reliability.
    - Minor deviation from the requirements.
    - Correctness can be measured in terms of the number of failures detected over time.
  - Performance
    - It is the measure of some required behaviour with respect to some relative time constraint.
    - Example :- real time system.
    - Measuring performance is based on mathematical or algorithmic complexity.
  - Usability
    - Measure of how easy the software is for humans to use.
    - Ease of use or user friendliness.
    - Measure of usability is informal feedback from users(surveys, reports, etc.)

Continues..

- Interoperability
  - It is ability of the software system to coexist and cooperate with other systems.
  - Example :- Embedded systems.
  - Measured in terms of compliance with open system standards.
- Open System
  - It is an extensible collection of independently written applications that cooperate to function as an integrated system.
  - Related to interoperability.
  - It is different from open source code.
  - It allows the addition of new functionality by independent organizations through the use of interfaces whose characteristics are published.
- Maintainability Evolvability and Repairability
  - Software Engineer should anticipate changes.
  - A software system in which changes are relatively easy to make has a high level of maintainability.
  - Maintainability has got 2 properties - Evolvability and Repairablity.
  - 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.

Continues..

- Portability
  - Software is portable if it can run easily in different environments.
  - Environment refers to the hardware on which the system resides, the operating system, or other software in which the system is expected to interact.
  - Example - Java programming language.
  - It can be 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.
- Traceability
  - It is the relationships between requirements, their sources, and the system design.
  - It can be obtained by providing links between all documentation and the software code.
  - There should be links:
    - from requirements to stakeholders who proposed these requirements
    - between dependent requirements
    - from the requirements to the design
    - from the design to the relevant code segments
    - from requirements to the test plan
    - from the test plan to test cases.

Continues..

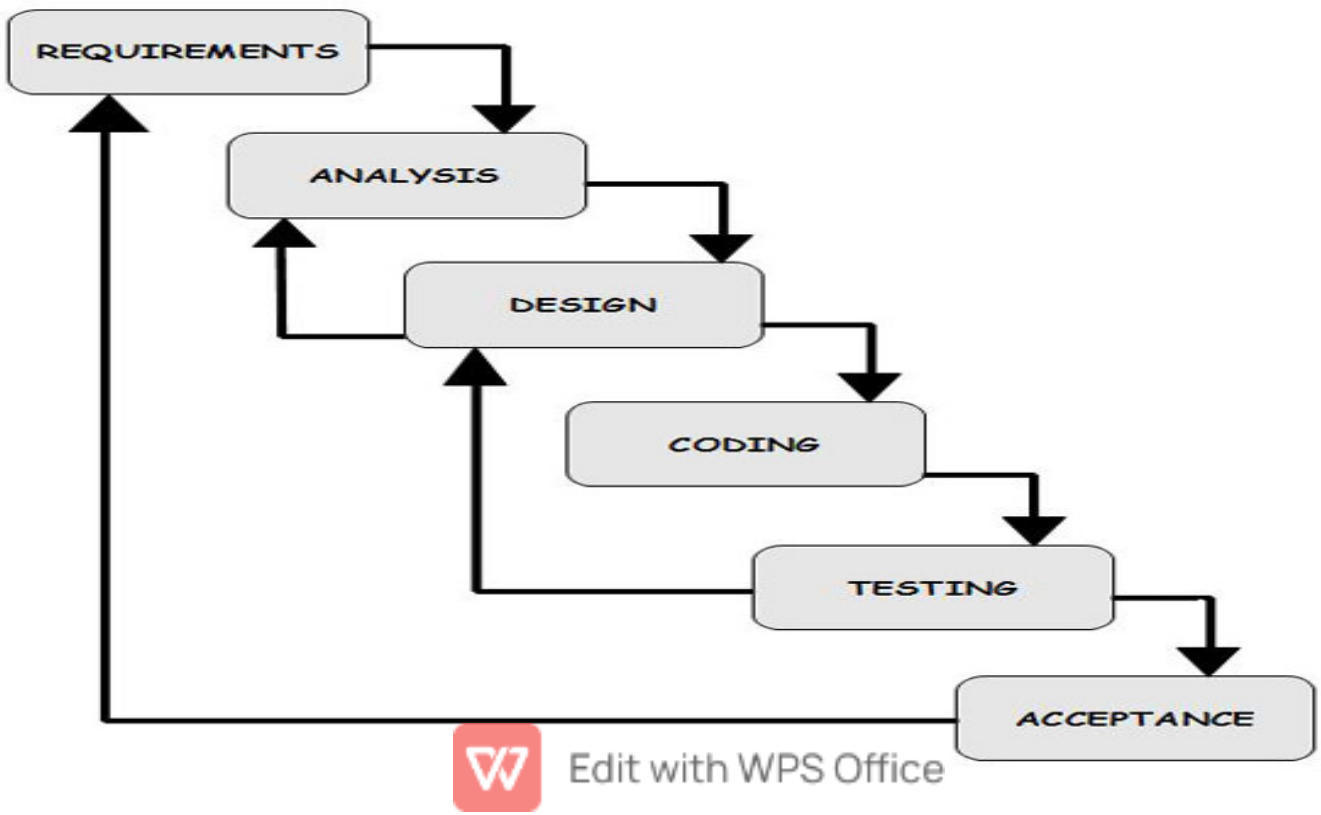
- Other Software Qualities

Negative Code Quality	Positive Code Quality
Fragility - Changes cause the system to break.	Robustness - Strong
Immobility - Code is hard to reuse.	Reusability
Needless complexity	Simplicity
Needless repetition	Parsimony
Opacity - Code is not clear.	Clarity
Rigidity - Design is hard to change.	Flexibility
Viscosity - Easier to do the wrong thing.	Fluidity

Life Cycle of a Software System

- Steps to be followed for developing a Quality Software.
- Different steps involved in software development:
  - Requirements Gathering
  - Design
    - High-level Design
    - Low-level Design
  - Development
  - Testing
  - Deployment
  - Maintenance

Phases in Software Engineering



Requirement Gathering

- A project cannot 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 should be done?
  - When and how long it should be continued?
  - Most important - What the project’s goals?
- They give the project direction.
- At the beginning of the project, you gather requirements from the customers to figure out what you need to build.

Continues..

- **Requirements are the features that your application must provide.**
- We need to find out what the customers want and what the customers need.
- Defining the user's needs is a time-consuming process.
- Once the customers' wants and needs are clearly specified, then requirements document have to be prepared.
- 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 the requirements document to see whether the project is heading in the right direction.
- At the end of the project, it is verified with requirements and find whether the finished application actually does what it's supposed to do.

Continues..

- Properties of good requirement:
  - Clear
    - Clear, concise, and easy to understand.
    - Cannot be vague or ill-defined.
  - Unambiguous
  - Consistent
    - 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.
  - Prioritized.
    - Should include every feature but don't have the time or budget, many thing have to be avoided. So the requirements are to be prioritized.
  - Verifiable
    - If it is not verifiable, how do you know whether you've met it? Being verifiable means the requirements must be limited and precisely defined.

Continues..

- MOSCOW Method
- This method is used for prioritizing application features.
- MOSCOW stand for the following:
  - 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. They can be pushed back into release 2.
  - W—Won't.
    - These are completely optional features that the customers have agreed I will not be included in the current release. They may be included in a future release if time permits.

REQUIREMENT CATEGORIES

Audience Oriented Requirements

- Focus on different audiences and the different point 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

- Layout project's high level goals.
  - User Requirements
  - Functional Requirements
  - Non Functional Requirements
  - Implementation Requirements

Continues..

User Requirements

- Describe how project will be used by end-users.

Functional Requirements

- Detailed statements of the project's desired capabilities.
- Example: The reports that the application produces, interfaces to other applications .

Non Functional Requirements

- Statements about the quality of the application's behavior.
- Applications performance , reliability and security characteristics.

Implementation Requirements

- Temporary features that are needed to transition to using the new system but that will be later discarded.
- The task described in implementation requirement don't always involve programming.
- Examples such as hiring new staff, buying new hardware, preparing training materials, Training users to use the system .

Continues..

- FURPS
  - It is another word for system's requirement categories. Full form of FURPS:
  - **Functionality** —What the application should do. These requirements describe the system's general features including what it does, interfaces with other systems, security etc.
  - **Usability** —What the program should look like. These requirements describe self-oriented features such as the application's general appearance, ease of use, navigation methods, and responsiveness.
  - **Reliability** —How reliable the system should be. These requirements indicate such things as when the system should be available ,how often it can fail, and how accurate the system is.
  - **Performance** —How efficient the system should be. These requirements describe such things as the application's speed, memory usage, disk usage, and database capacity.
  - **Supportability** —How easy it is to support the application. These requirements include such things as how easy it will be to maintain the application, how easy it is to test the code, and how flexible the application is.

Contines..

- Listen to Customers (and Users)
  - Learn as much as you can about the problem users are trying to address and any ideas they may have about how the application might solve that problem.
  - Initially, focus as much as possible on the problem, not on the customers' suggested solutions, so you can keep the requirements flexible.
  - If the customers insist on a particular feature that you think is unimportant, or if they request something that just seems strange, ask them why they want it.

Use the Five Ws (and One H)

- Who
  - Ask who will be using the software and get to know as much as you can about those people.
  - Find out if the users and the customers are the same and learn as much about the users as you can.
- What
  - Figure out what the customers need the application to do.
  - Focus on the goals as much as possible rather than the customers' ideas about how the solution should work.
- When
  - Find out when the application is needed. If the application will be rolled out in phases, find out which features are needed when.
- 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. Find out if there is a real reason to believe a new application will help.
- How
  - If the users are used to doing something a certain way, you may reduce training time.

Study Users

- Interviewing customers (and users) can get you a lot of information, but often customers (and users) won’t tell you everything they do or need to do.
- Users consider information unimportant for them but that may be important to the development team.
- Pay attention to how they do things.
- Figure out where they spend most of their time.
- Look for the tasks that go smoothly and those that don’t.

Refining Requirements

- After study user, you should have a good understanding about the users' current operations and needs.
- You need to distill the goals (what the customers need to do) into approaches (how the application will do it).
- Three approaches for converting goals into requirements.
  - Copy Existing Systems
  - Clairvoyance
  - Brainstorm

Continues..

- Advantages
  - It doesn’t take an enormous amount of software engineering experience to dig through an existing application and write down what it does.
  - This approach also makes it more likely that the requirements can actually be satisfied, at least to the extent the current system works.
  - This approach provides an unambiguous example of what you need to do.
- Disadvantages
  - You probably wouldn’t be building a new version of an existing system unless you planned to make some changes.
  - Users are often reluctant(dislike) to give up even the tiniest features in an existing program.

Clairvoyance

- Foreseeing.
- This technique is particularly effective if the project team has previously built a similar system.
- In this case, the team already knows more or less what the application needs to do, which things will be easy and which will be hard, how much time everything requires.

Continues..

- Advantages
  - Having an experienced project lead greatly increases the chances that the requirements will include everything you need to make the project succeed.
  - It also greatly increases the chances that the team will anticipate problems and handle them easily as development continues.
  - Documenters who have written user manuals for similar applications will find writing manuals for the new project easier.
  - Project managers with similar experience will know what tasks are likely to be difficult.
  - Even customers with previous software engineering experience will be better at creating good requirements.
- Disadvantages
  - This technique will not lead you to new innovative solutions that might be better than the old ones

Brainstorm

- Both the previous techniques share common advantage.
- They are unlikely to lead you to new innovative solutions that might be better than the old ones.
- If revolutionary solutions are needed, you need to be more creative.
- Brainstorming helps to create more creative solutions.
- The basic approach that most people think of as brainstorming is called the *Osborn method developed by* Alex Faickney Osborn.

Continues..

- The base of the Osborn method is:
  - To gather as many ideas as possible, not worrying about their quality or practicality.
  - After you assemble a large list of possible ideas, you examine them more closely to see which deserve further work.
  - Try to get a diverse group of participants which include customers, users, user interface designers, system architects, team leads, programmers, trainers, and anyone else who has an interest in the project.
  - Get as many different viewpoints as you can.
  - To keep the ideas flowing, don't judge or critique any of the ideas. If you criticize someone's ideas, that person may shut down and stop contributing.
  - Write down every idea no matter how impractical it may seem.

Continues..

- Osborn’s method uses the following four rules:
  - Focus on quantity.
    - Do everything you can to keep the ideas flowing. The more ideas you collect, the greater your chances of finding a really creative and revolutionary solution.
  - Withhold criticism.
    - Criticism can make people stop contributing. Early criticism can also eliminate seemingly bad ideas that lead to better ideas.
  - Encourage unusual ideas .
    - You can always “tone down a wild idea” but you may need to think way outside of the box to find really creative solutions.
  - Combine and improve ideas.
    - Form new ideas by combining other ideas or using one idea to modify another.

Continues..

- Other than Osborn's method there are several other brainstorming techniques. The following list describes some of those techniques.
- Popcorn
  - People just speak out as ideas occur to them. This works fairly well with small groups of people who are comfortable with each other.
- Subgroups
  - Break the group into smaller subgroups (possibly in the same room) and have each group brainstorm. When the subgroups are finished, have the larger group discuss their best ideas. This works well if the main group is very large.
- Sticky notes
  - Also called the nominal group technique (NGT). Participants write down their ideas on sticky notes, index cards, etc. The ideas are collected, read to the group, and the group votes on each idea. The best ideas are developed further, possibly with other rounds of brainstorming.
- Idea passing
  - Participants sit in a circle. Each person writes down an idea and passes it to the next person. The participants add thoughts to the ideas they receive and pass them on to the next person.
- Circulation list
  - This is similar to idea passing except the ideas are passed via e-mail, envelope, or some other method outside of a single meeting.
- Rule breaking
  - List the rules that govern the way you achieve a task or goal. Then everyone tries to think of ways to break or circumvent those rules while still achieving the goal.
- Individual
  - Participants perform their own solitary brainstorming sessions. They can write (or speak) their trains of thought, use word association, draw mind maps (diagrams relating thoughts and ideas).

Continues..

- The following list describes some tips that can make brainstorming more productive.
  - Work in a comfortable room where everyone can feel at ease.
  - Provide food and drinks.
  - Start by recapping the users' current processes and the problems you are trying to solve.
  - Use a clock to keep sessions short and lively..
  - Allow the group's attention to wander a bit, but keep the discussion more or less on topic.
  - Few jokes can keep people relaxed and help ideas flow.
  - If you get stuck, restate the problem.
  - Allow silent periods so that people have time to think about the problem and their ideas.
  - Reverse the problem. For example, instead of trying to think of ways to build better blogging software, think of ways to build worse blogging software.
  - Write ideas in slightly ambiguous ways and let people give their interpretations.
  - At the end, summarize the best ideas and give everyone copies so that they can think about them later.

RECORDING REQUIREMENTS

- After gathering the requirements, the actual requirements have to be recorded.
- It can be done by many ways. They are :
  - UML(Unified Modeling Language)
  - User Stories
  - Use Cases
  - Prototypes

UML

- Used to specify how parts of the system should work.
- It uses several kinds of diagrams to represent different pieces of the system.
- Example:
  - Classes – class diagrams
  - Sequence – sequence diagram
  - Etc.
- Major drawback of UML is that it is complicated.

User Stories

- It is exactly what you might think: a short story explaining how the system will let the user do something.
- User stories should come with acceptance testing procedures that you can use at the end of development to decide whether the application satisfied the story.
- It is less technical.
- Advantages
  - They are easy to write, easy to understand, and can cover just about any situation you can imagine.
  - Your customers, developers, managers, and other team members already know how to understand stories without any new training.
- Disadvantages
  - It can be easily written, that are confusing, ambiguous, inconsistent with other stories, and unverifiable.

Use Cases

- It is a description of a series of interactions between actors. The actors can be users or parts of the application.
- Template of an use case
  - Title —the title includes an action (examines) and the main actor (user).
  - Main success scenario —A numbered sequence of steps describing the most normal variations of the scenario.
  - Extensions —Sequences of steps describing other variations of the scenario. This may include cases such as when the user enters invalid data or the application can’t handle a request.

Prototypes

- It is a mockup of some or all of the application.
- The idea is to give the customers a feel for what the finished application will look like and how it will behave than you can get from text descriptions such as user stories and use cases.
- A simple user interface prototype might display forms that contain labels, text boxes, and buttons showing what the finished application will look like.
- It might use less efficient algorithms.
- Load data from a text file instead of a database, or display random messages instead of getting them from another system. It might even use hard-coded fake data.
- In nonfunctional prototype, the buttons, menus, and other controls on the forms wouldn't actually do anything.
- A functional prototype (or working prototype) looks and acts as much like the finished application.

Continues..

- Two different kinds of prototype are :
  - Throwaway Prototype :
    - After you've fine-tuned the prototype so that it represents the customers' requirements , can be left alone.
    - It can be referred further if there's a question about what the application should look like or how it should work.
    - The software has to start from scratch when building the application. Such prototypes are called a *throwaway prototype*.
  - Evolutionary Prototype
    - You can start replacing the prototype code and fake data with production-quality code and real data.
    - Over time, you can evolve the prototype into increasingly functional versions until eventually it becomes the finished application. This type prototypes are called an *evolutionary prototype*.

Requirement Specification

- Otherwise called SRS or software Requirement Specification.
- How to write requirements, depends on the project.
- If a simple software is going to be developed, a simple description may be enough but for a standard software the legal procedures has to be considered.
- Several templates for requirement specifications are available.

VALIDATION AND VERIFICATION

- Requirement validation is the process of making sure that the requirements say the right things.
- Someone, often the customers or users, need to work through all the requirements and make sure that

  they: (1) Describe things the application should do.

  (2) Describe everything the application should do.
- Requirement verification is the process of checking that the finished application actually satisfies the requirements.

High Level Design

- Provides a view of the system at an abstract level.
- It shows how the major pieces of the finished application will fit together and interact with each other.
- It is the overall system design covering the system architecture.
- It describes the relation between various modules and function of the system.
- It will not focus on the details of how the pieces of the application will work.
- Software development as a process that chops up the system into smaller and smaller pieces until the pieces are small enough to implement.
- High-level design is the first step in the chopping up process.
- Individual pieces might be small enough that they can be handled by individual developers instead of teams.

Continues..

- Some of the most common items you might want to specify in the high-level design.
  - Security
  - Hardware
  - User Interface
  - Architecture
  - Reports
  - Other Outputs
  - Database
  - Configuration Data
  - Data flows and States
  - Training

Security

The high-level design should sketch out all the application's security needs.

- Operating system security—This includes the type of login procedures, password expiration policies, and password standards.
- Application security—A separate application username and password. Application security also means providing the right level of access to different users.
- Data security— Protecting data from destructive forces and unwanted actions of unauthorized users.
- Network security—Network security consists of policies and practices adopted to prevent and monitor unauthorized access of computer network and network accessible resources.
- Physical security—Many software engineers overlook physical security. Physical security prevents and discourages attackers by installing alarms, cameras etc.

Hardware

- You can build systems to run on mainframes, desktops, laptops, tablets and phones.

Additional hardware that you need to specify might include the following:

- Printers.
- Network components (cables, modems, gateways, and routers).
- Servers (database servers, web servers, and application servers).
- Specialized instruments (scales, microscopes, programmable signs, and GPS units).
- Audio and video hardware (webcams, headsets, and VOIP).

User Interface

- Indicates the main methods for navigating through the application.
- In addition to the application's basic navigational style, the high- level user interface design can describe special features such as clickable maps, important tables, or methods for specifying system settings (such as sliders, scrollbars, or text boxes).
- Also address general appearance issues such as color schemes, company logo placement and form skins.

Internal Interfaces

- Specifies the internal interactions or specify how the pieces will interact.
- The teams assigned to the pieces can work separately without needing constant coordination.
- Specifies these internal interactions clearly and unambiguously so that the teams can work as independently as possible.

External Interfaces

- Many applications must interact with external systems.
- External interface are easier to specify than internal interface.

Architecture

- An application’s architecture describes how its pieces fit together at a high level.

Monolithic

- In this type a single program does everything.
- It displays the user interface, accesses data, processes customer orders, prints invoices, launches missiles, and does whatever else the application needs to do.

Disadvantages

- The pieces of the system are tied closely together, so it doesn’t gives any flexibility.
- If you get any of the details wrong, the tight coupling between the pieces of the system makes fixing them later difficult.

Advantages

- No need for complicated communication across networks.
- You don’t need to worry about the network going down; and you don’t need to worry about network security.

Continues..

- Client/Server
  - A client/server architecture separates pieces of the system that need to use a particular function (clients) from parts of the system that provide those functions(servers).
  - For example, many applications rely on a database to hold information about customers, products, orders and employees. The application needs to display that information in some sort of user interface. One way to do that would be to integrate the database directly into the application.
  - One problem with this design is that multiple users cannot use the same data. You can fix that problem by moving to a two-tier architecture where a client (the user interface) is separated from the server.

Continues..

Two-tier architecture

- The clients and server communicate through some network such as a local area network (LAN), wide area network (WAN), or the Internet.
- The two-tier architecture support multiple clients with the same server, but it ties clients and servers relatively closely together.

Three-tier architecture

- Three-tier helps to increase the separation between the clients and server by introducing a middle-tier.
- The middle tier is separated from the clients and the server by networks.
- The database runs on one computer, the middle tier runs on a second computer, and the instances of the client uses another computer.
- It provides an interface that can map data between the format provided by the server and the format needed by the client.
- If you need to change the way the server stores data, you need to update only the middle tier so that it translates the new format into the version expected by the client.
- If the client's data needs change, you can modify the middle tier.

- Multi-tier architecture (N -tier architecture) - uses more than three tiers.

Continues..

Component-Based Software Engineering(CBSE)

- A system with a collection of loosely coupled components that provide services for each other.

Service-Oriented Architecture

- A service-oriented architecture (SOA) is similar to a component-based architecture except the pieces are implemented as services.
- A service is a self-contained program that runs on its own and provides some kind of service for its clients.

Continues..

Data-centric or database-centric architectures

- Storing data in a relational database system.
- Using tables instead of hard-wired code to control the application.
- Using stored procedures inside the database to perform calculations and implement business logic.

Event-Driven

- In an event-driven architecture (EDA), various parts of the system respond to events as they occur.

Rule-Based

- A rule-based architecture uses a collection of rules to decide what to do next.
- These systems are sometimes called expert systems or knowledge-based systems .
- Eg: Troubleshooting system

Continues..

Distributed

- In a distributed architecture, different parts of the application run on different processors and may run at the same time.
- The processors could be on different computers scattered across the network, or they could be different cores on a single computer.
- Service-oriented and multi-tier architectures are often distributed, with different parts of the system running on different computers.

Mix and Match

- An application doesn’t need to stick with a single architecture. Different pieces of the application might use different design approaches.

Outputs

- Report
  - Not only deliverables, there are many more reports.
  - Software project can use some kinds of reports.
  - Business applications might include reports that deal with customers (who's buying, who has unpaid bills, where customers live), products (inventory, pricing, what's selling well), and users (which employees are selling a lot, employee work schedules).
- Other Outputs
  - In addition to normal reports, you should consider other kinds of outputs that the application might create.
  - The application could generate printouts (of reports and other things), web pages, data files, image files, audio, video, output to special devices, e-mail, or text messages.

Database

- You need to specify whether the application will store data in text files, XML files, a full-fledged relational database, or something more exotic such as a temporal database or object store.
- Three common database-specific issues that you should address during high level design:
  - Audit trails.
  - User access.
  - Database maintenance.
- Audit trails
  - An *audit trail* keeps track of each user who modifies (and in some applications / views) a specific record.
  - Auditing can be as simple as creating a history table that records a user's name, a link to the record that was modified and the date when the change occurred.
  - Simply log.

Continues..

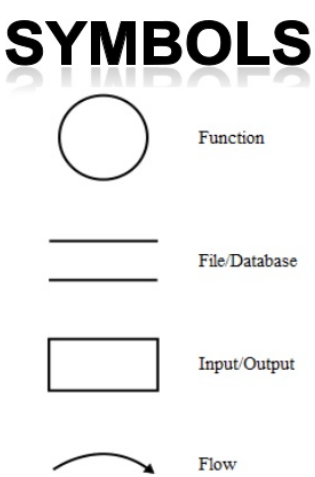
- User Access
  - Many applications also need to provide different levels of access to different kinds of data.
  - One way to handle user access is to build a table listing the users and the privileges they should be given.
  - The program can then disable or remove the buttons and menu items that a particular user shouldn't be allowed to use.
- Database Maintenance
  - Over time database gets disorganized and full of random junk, so the database has to be organized.
  - Move the older data into a *data warehouse*, a secondary database that holds older data for analysis.
  - Removing old data from a database can help keep it responsive, but a lot of changes to the data can make the database's indexes inefficient and that affect the performance. So the solution is periodically run database tuning software to restore best performance.
  - So during design a database backup and recovery scheme should be introduced.

Configuration Data

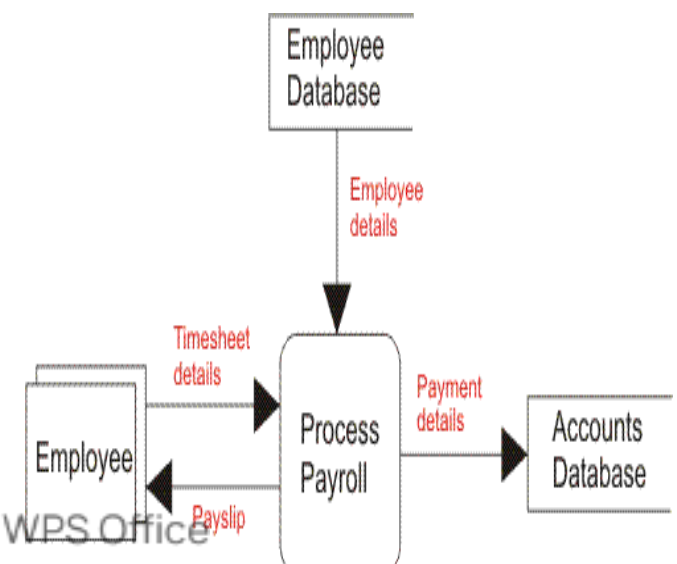
- The workload can be reduced, if you provide configuration screens so that users can fine-tune the application without making you write new code.
- Make sure that only the right users can modify the parameters.
- In many applications, only managers should change these values.

Data flows and States

- In order to describe the system and the way processes interact with the data, many data flows and state transition diagrams are used.
- Example : DFD



Example



Training

- Deciding whether users want to attend courses taught by instructors, read printed manuals, watch instructional videos, or browse documentation online.

UML

- Unified Modeling Language
- 13 diagram types divided into three categories.
- Diagram
  - Structure Diagram
    - **Class Diagram**
    - Composite Structure Diagram
    - **Component Diagram**
    - **Deployment Diagram**
    - Object Diagram
    - **Package Diagram**
    - Profile Diagram
  - Behavior Diagram
    - **Activity Diagram**
    - **Use Case Diagram**
    - State Machine Diagram
    - Interaction Diagram
    - **Sequence Diagram**
      - Communication Diagram
      - Interaction Overview Diagram
      - Timing Diagram

Low level Design

- 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.
- It gives more specific guidance for how the parts of the system will work and how they will work together.
- It refines the definitions of the database, the major classes and the internal and external interfaces.
- High-level design focuses on “what”. Low-level design focus on “how”.
- The boarder between high level and low level design cannot be clearly defined.
- Low-level design can be considered as high-level design for micro-managers.

Continues..

- While doing low level design, the following things has to be considered.
  - Object Oriented Design
  - Database Design

OO Design

- Major type of classes will be identified during high-level design.
- Then refine that design to identify the specific classes that program will need.
- The new classes should include definitions of the properties, methods, and events they will provide for the application to use.
- An instance of a class is an object with the class's type.
- Classes define three main items: properties, methods, and events.
- A property is something that helps define an object.
- A method is a piece of code that makes a d n object do something.
- An event is something that occurs to tell the program that something interesting has happened.
- An object-oriented design pattern is an arrangement of classes that performs some common and useful task.

Continues..

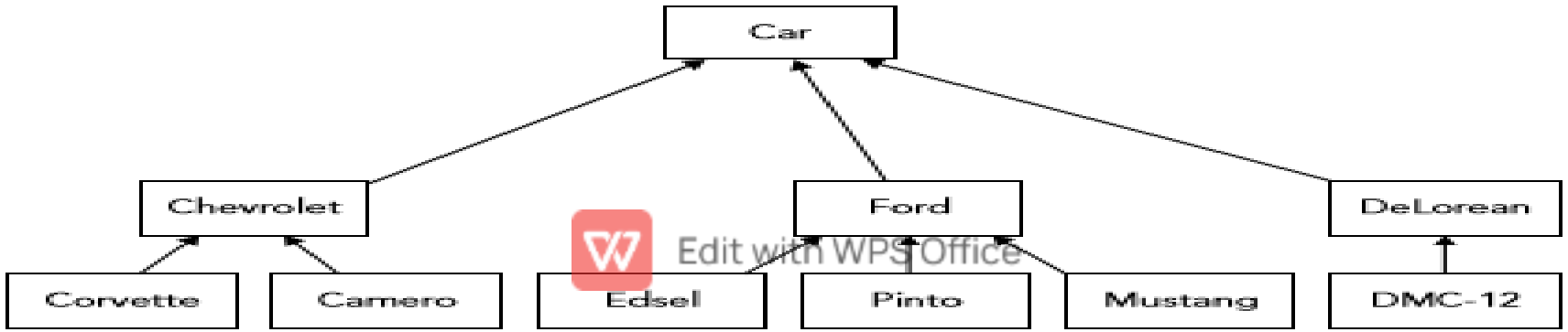
- Identifying classes
  - For example, suppose the task is to write an application called Free Wheeler Automatic Driver (FAD) that automatically drives cars.
  - So there should be two classes Car and Destination.
  - The Car class is going to be fully loaded
  - Properties - CurrentSpeed, CurrentDirection and FuelLevel
  - Methods - Accelerate, Decelerate, ActivateTurnSignal etc.
  - Events - DriverPressedStart, FuelLevelLow and CollisionImminent.

Continues..

- Building Inheritance Hierarchies
  - After defining the main class, you need to get the characteristics of other classes.
  - For example, Free Wheeler is going to need a Car class to represent the vehicle it's driving, but different vehicles have different characteristics.
  - You can capture the differences between related classes by deriving a child class from a parent class.
  - Child classes automatically inherit the properties, methods, and events defined by the parent class.
  - Types of inheritance : single inheritance, multiple inheritance, multi-level inheritance and derived inheritance.
  - The capability to treat objects as if they were actually from a different class is called *polymorphism* .

Refinement

- Refinement is the process of breaking a parent class into multiple subclasses to capture some difference between objects in the class.
- Overrefinement , which happens when you refine a class hierarchy unnecessarily, making too many classes that make programming more complicated and confusing.
- Example, this can be further refined.



Continues..

- When the hierarchy grows following problems may arise.
  - The classes are capturing data that isn’t relevant to the application.
  - The differences between cars could easily be represented by properties instead of by different classes.

Generalizations

- Refinement starts with a single class and creates child classes to represent differences between objects.
- Generalization does the opposite: It starts with several classes and creates a parent for them to represent common features.
- Example:
  - You define a Customer class and an Employee class. They share some properties such as Name , Address, and ZodiacSign , so you generalize them by making a Person class to hold the common properties.

Object Composition

- Inheritance is one way you can reuse code.
- A child class inherits all of the code defined by its parent class, so you don’t need to write it again.
- Another way to reuse code is object composition , a technique that uses existing classes to build more complex classes.

Database Design

- There are many different kinds of databases that you can use to build an application.
- For example, specialized kinds of databases store hierarchical data, documents, graphs and networks, key/value pairs, and objects.
- But the most popular kind of databases are relational databases.
- Advantages of relational database:
  - Simple
  - Easy to use
  - Provide a good set of tools for searching
  - Combining data from different tables, sorting results, and otherwise rearranging data.

Continues..

- A relational database stores related data in tables.
- Each table holds records/tuples that contain pieces of data that are related.
- The pieces of data in each record are called fields.
- Each field has a name and a data type.
- Example for a database.
- Example for a relational database.
- The table containing the foreign key is often called the child table.
- The table that contains the uniquely identified record is often called the parent table .

Continues..

- While designing a relational database the following problems may arise:
  - Duplicate data can waste space and make updating values slow.
  - You may be unable to delete one piece of data without also deleting another unrelated piece of data.
  - An otherwise unnecessary piece of data may need to exist so that you can represent some other data.
  - The database may not allow multiple values when you need them.

Development

- To many programmers, development is the heart of software engineering.
- Without development, there is no application.
- As the programmers write the code, they test it to make sure it doesn’t contain any bugs.

USE THE RIGHT TOOLS

- For the convenience of the programmer, the developer has to purchase the right tools.
- Hardware
  - For developing the right application, programmer should have adequate hardware.
  - That means, programmers need fast computers with adequate memory and disk space.
  - To write bugfree code, a programmer must stay focused on a method's design until it has been completely written.
- There are two drawbacks to buying the programmers everything they need.
  - May go overboard and buy all sorts of fun toys that they don't actually need.
  - By giving developers everything they may sometimes forget that their users may not have such nice equipment.

Continues..

- Network
  - Even if hackers makes problems, the developers should have to give access to the external network(Internet).
- Development Environment
  - An **integrated development environment** (**IDE**) is a software application that provides comprehensive facilities to computer programmers for software development.
  - An IDE normally consists of a source code editor, build automation tools and a debugger.
  - Example :
    - Visual Studio - Visual C#, Visual Basic, Visual C++, JavaScript, and F#
    - Besides above debuggers, code profilers, class visualization tools, auto-completion when typing code, context-sensitive help, team integration tools, and more.

Continues..

- Source Code Control
  - Source code control is important for program code where changing a single character can reduce a working program worthless.
  - A good source code management system enables you to go back through past versions of the software and see exactly what changes were made and when.
  - Source code control programs also prevent multiple programmers to work on same code.
- Profilers
  - Profilers let you determine what parts of the program use the most time, memory, files, or other resources.
  - It saves huge amount of time when you’re trying to tune an application’s performance.
  - A small part of a program’s code may determines its overall performance, so there is no need to study every line’s performance.

Continues..

- Static Analysis Tools
  - This tool study code without executing it.
  - It focus on the code's style.
  - It can measure how interconnected different pieces of code are or how complex a piece of code is.
  - Example
    - they can measure how interconnected different pieces of code are or how complex a piece of code is.
- Testing Tools
  - Automated tools for testing.
  - It makes testing a whole program lot faster, easier, and more reliable.
- Source Code Formatters
  - They are development environments which can format code in a better way.
  - Formatting makes code easier to read and understand.
  - Reduces the number of bugs in the code and makes finding and fixing bugs easier.
  - Example
    - Some environments automatically indent source code to show how code is nested in if-then statements and loops.

Continues..

- Refactoring Tools
  - A tool used for rearranging code to make it easier to understand, more maintainable, or generally better.
  - Some refactoring tools let you do things like easily define new classes or methods.
  - They can be more useful if you’re managing existing code.
- Training
  - Makes programmers more effective and keeps them happy.
  - Improve performance and help you retain your staff.

Continues..

- SELECTING ALGORITHMS
  - For solving hard programming problems a good algorithm have to be selected.
  - Qualities of an efficient algorithm
    - Effective
    - Efficient
    - Predictable
    - Simple
    - Prepackaged
- Top Down Design
  - Also called stepwise refinement.
  - Start with a high-level statement of a problem, and you break the problem down into more detailed pieces.
  - Examine the pieces and break that pieces which are too big into smaller pieces.

Continues..

- PROGRAMMING TIPS AND TRICKS
  - Be Alert.
  - Write for People, Not the Computer.
  - Comment First.
  - Write Self-Documenting Code.
  - Keep It Small.
  - Stay Focused.
  - Avoid Side Effects: unexpected result.
  - Validate Results.
  - Practice Offensive Programming.
  - Use Exceptions.
  - Write Exception Handlers First.
  - Don’t Repeat Code.
  - Defer Optimization: First make it work. Then make it faster if necessary.

Testing

- Definition by James Bach
  - Testing is the process of comparing the invisible to the ambiguous, so as to avoid the unthinkable happening to the anonymous.
- All nontrivial programs contain bugs.
- The industry average number of bugs per thousand lines of code (kilo lines of code or KLOC) is typically estimated at about 15 to 50.
- Testing Goals
  - The code may not work in every possible situation.
  - Testing is study a piece of code to see whether it meets the requirements and whether it works correctly under all circumstances.

REASONS BUGS NEVER DIE

- A bug is a flaw in a program that causes it to produce a g n incorrect result or to behave unexpectedly.
- Removing some bugs is just more trouble than it's worth.
- The following are some reasons that the bugs couldn't be removed.
  - Diminishing Returns
    - Finding a error in the starting is relatively easier than moving towards the end. And it may cost more than the software.
  - Deadlines
  - Consequences
  - It's too soon
    - Next version is too soon.
  - Usefulness
    - If a user is used with a bug, they may not be interest in correcting that bug.
  - Obsolesce
    - It may be good to spend time to correct bug on a software which is going to be obsolete.
  - It is not a bug.
    - The user couldn't understand, what the program is supposed to do.
  - It never ends.
  - It is better than nothing.
  - Fixing bugs is dangerous.

Which Bugs to Fix?

- Depending on the following factors bugs have to be fixed :
  - Severity —How painful is the bug for the users? How much work, time, money, or other resources are lost?
  - Work-arounds —Are there work-arounds?
  - Frequency —How often does the bug occur?
  - Difficulty —How hard would it be to fix the bug? (Of course, this is just a guess.)
  - Riskiness —How risky would it be to fix the bug? If the bug is in particularly complex code, fixing it may introduce new bugs

LEVELS OF TESTING

- Bugs are easiest to fix if you catch them as soon as possible.
- In order to catch bugs as soon as possible, you can use several levels of testing.
- Unit Testing
  - It verifies the correctness of a specific piece of code.
  - After finishing a piece of code, it has to be tested.
  - Test it as thoroughly as possible because it will get harder to fix later.
  - Usually unit tests apply to methods.
- Integration Testing
  - It verifies that the new method works and plays well with others.
  - It checks that existing code calls the new method correctly, and that the new method can call other methods correctly.
  - Regression testing can be done at this phase.
  - Regression test is testing the program's entire functionality to see if anything changed when you added new code to the project.
  - Performing regression testing on a large project can take a lot of time.

Continues..

- Automated Testing
  - As developer has so many works, he couldn’t do all the test.
  - So on behalf of developer the testing can be conducted by an automated tool.
  - Automated testing tools let you define tests and the results are produced.
  - After testing, the testing tool can compare the results it got with expected results.
  - Some testing tools can run load tests that simulate a lot of users all running simultaneously to measure performance.
  - A good testing tool should let you schedule tests so that you can run regression testing every night. Correction can be done in the morning.
- Component Interface Testing
  - It studies the interactions between components.
  - It is similar to regression testing in the sense that both examine the application as a whole, but component interface testing focuses on component interactions.

Continues..

- System testing
  - It is an end-to-end run-through of the whole system.
  - It tests every part of the system to discover each and every bugs as possible.
- Acceptance Testing
  - It is to determine whether the finished application met the customers' requirements.
  - The user or customer checks whether the finished application satisfies the requirements gathered during the requirement gathering.
  - The requirements may be changed.

Other Testing Categories

- Testing that differ in their scope, focus, or point of view.
- The following list summarizes other categories of testing:
  - **Accessibility test** —Tests the application for accessibility by those with visual, hearing, or other impairments.
  - **Alpha test** —First round testing by selected customers or independent testers.
  - **Beta test**—Second round testing after alpha test. Some thing like trial version.
  - **Compatibility test** —Focuses on compatibility with different environments such as computers running older operating system versions.
  - **Destructive test** —Makes the application fail so that you can study its behavior when the worst happens.
  - **Functional test** —Deals with features the application provides. These are generally listed in the requirements.
  - **Installation test** —Makes sure you can successfully install the system on a fresh computer.
  - **Internationalization test** —Tests the application on computers localized for different parts of the world.
  - **Nonfunctional test** —Studies application characteristics that aren't related to specific functions the users will perform.
  - **Performance test** —Studies the application's performance under various conditions such as normal usage, heavy user load, limited resources (such as disk space), and time of day.
  - **Security test** —Studies the application's security.
  - **Usability test** —Determines whether the user interface is intuitive and easy to use.

TESTING TECHNIQUES

- The following describe some approaches for designing tests to find bugs(this may be a combination of the above discussed test types).
  - Exhaustive Testing
    - Proves that a method works correctly under all circumstances.
  - Black-Box Testing
    - You know what it is supposed to do, but you have no idea how it works.
    - You then throw all sorts of inputs at the method to see what it does.
  - White-Box Testing
    - In this case, you know how the method does its work.
    - You then use your extra knowledge to design tests to try to make the method crash and burn.
  - Gray-Box Testing
    - It is a combination of white-box and black-box testing.
    - In this case you have some knowledge about the method.

TESTING HABITS

- Test and Debug When Alert.
- Test Your Own Code.
- Have Someone Else Test Your Code.
- Fix Your Own Bugs.
- Think Before You Change.
- Don’t Believe in Magic.
- See What Changed.
- Fix Bugs, Not Symptoms.
- Test Your Tests.

HOW TO FIX A BUG?

- Ask yourself how you could prevent a similar bug in the future.
- Ask yourself if a similar bug could be hiding somewhere else.
- Look for bugs hidden behind this one.
- Examine the code’s method and look for other possibly unrelated bugs.
- Make sure your fix doesn’t introduce a new bug

Estimating number of bugs.

- Exact number of bugs couldn’t be estimated.
- An approximate number can be found by two methods.
  - Tracking bugs found
    - Estimate bugs by tracking the number of bugs found over time.
    - When testing starts, the number will be high.
    - When testers uncovers the bugs, it levels.
    - This continues up to no bugs found.
  - Seeding
    - Simply scatter some bugs throughout the application.
    - Run your tests and see how many of the artificial bugs you find.
    - Example
      - Insert 40 bugs in to code and tests finds 34 of them.
      - That is 85-percent of bugs have been found.
      - When you tested 135 real bugs are found, then there may have originally been approximately $135 \div 0.85 \approx 159$ bugs.
      - That means there are about $159 - 135 = 24$ bugs remaining.

Deployment

- After the development and testing phases, deployment has to be done.
- Is also called installation, implementation and release.
- Deployment is the process of putting the finished application in the users' hands.
- Scope
  - Scope includes the size of the application.
  - It includes the amount of data involved, the number of external systems that are affected, and the quantity of code and so on…

Continues..

- The Plan
  - To start deployment planning, list the steps that you hope to follow.
  - Describe each step in detail as it is supposed to work.
- Cutover
  - Cutover is the process of moving users to the new application.
  - There are several ways to manage cutover:
    - Just post the new version on the Internet and let users grab it.
    - E-mail a new version to users.
    - Just install the new system on users’ computers.

Cutover

- Initially, the users may be unable to do their jobs.
- To help them several techniques can be used:
  - Staged deployment
    - we build a staging area, a fully functional environment where we can practice deployment until we have worked out all the features.
  - Gradual cutover
    - The new application will be installed for some users while other users continue working with their existing system.
    - First one user to the new application and thoroughly test it.
    - If every thing goes well, the second user will be moved to the new system.
    - This will be continued until everyone is running the new application.
  - Incremental deployment
    - The new system's features will be released to the users gradually.
    - First one tool will be installed.
    - After the users are used to the new tool, next tool will be installed.
  - Parallel testing.
    - Depending on the complexity of the new system, both the existing system and the application will run in parallel.

Deployment Tasks

- During deployment the following thing might need to deal with:
  - **Physical environment**.
  - **Hardware**.
  - **Documentation.**
    - It might include training materials, user manuals, help guides, and cheat sheets listing common commands.
  - **Training**.
  - **Database.**
    - There may be a need to install database software on one or more central database servers and on the users' computers.
  - **Other people's software.**
    - **It includes systems that** interact with your application (purchasing systems, web services, file management tools, cloud services, and printing and scanning tools).
  - **Your software**.
    - It includes the application itself, plus any extra tools you've created. It also includes monitoring and testing tools that let you make sure the application is working correctly.

Deployment Mistakes

- The basic steps for successful deployment are
  - Make the plan.
  - Anticipate mistakes.
  - Work through the plan overcoming obstacles as they arise.
- The following are some of the mistakes that may arise:
  - Assume everything will work.
  - Have no rollback plan .
  - Allow insufficient time.
  - Don’t know when to surrender.
  - Skip staging.
  - Install lots of updates all at once.
  - Use an unstable environment.
  - Set an early point of no return.

Maintenance

- It is the modification of a software product after delivery to correct faults, to improve performance or other attributes.
- Maintenance uses 75 percent of a project's total cost.
- Generally maintenance tasks are grouped into the following four categories:
  - Perfective
  - Adaptive
  - Corrective
  - Preventive
- The total 75 percent of the cost is distributed among tasks as below:
  - Perfective —50 percent
  - Adaptive —25 percent
  - Corrective —20 percent
  - Preventive —5 percent

Maintenance Task

- Perfective
  - Improving existing features and adding new ones.
  - The users may still want changes, adjustments, and improvements.
- Adaptive
  - Modifying the application to meet changes in the application's environment.
  - If the users' hardware, OS, database, other tools (such as spreadsheets or reporting tools), network security, or other pieces of the users' environment changed, then it could break our application.
  - So this has to be corrected.
- Corrective
  - Fixing bugs.
  - Study the code so that we are sure we understand it, fix the bug, test, and release a new version of the application with the bug fixed.
- Preventive
  - Restructuring the code to make it more maintainable.
  - Restructuring the code to make it easier to debug and maintain in the future.

Project Planning

- Software project management begins with a set of activities that are collectively called project planning.
- Estimating the work to be done, the resources that will be required and the time that will elapse from start to finish.
- Objective of project planning.
  - Is to provide a framework that enables the manager to make reasonable estimates of resources, cost, and schedule.
  - It attempts to define best case and worst case scenario so that project outcomes can be bounded.
  - Project plan must be adapted and updated as project proceeds because of uncertainty.

Tasks in project planning

1. Establish project scope.
2. Determine feasibility.
3. Analyse risks.
4. Define required resources.
   a. Determine required human resources.
   b. Define reusable software resources.
   c. Identify environmental resources.
5. Estimate cost and effort
   a. Decompose the problem.
   b. Develop two or more estimates using size, function, process tasks, or use cases.
   c. Reconcile the estimates.
6. Develop a project schedule.
   a. Establish a meaningful task set.
   b. Define a task network.
   c. Use scheduling tools to develop a time-line chart.
   d. Define schedule tracking mechanism.

Project Scope

- Project scope describes
  - The functions and features that are to be delivered to end users.
  - The data that are input and output.
  - The content that is presented to users as a consequence of using the software.
  - The performance, constraints, interfaces, and reliability that bound the system.
- Project scope is defined using one of 2 techniques:
  1. A narrative description of software scope is developed after communication with all stakeholders.
  2. A set of use cases is developed by end users.
- Performance considerations encompass processing and response time requirements.
- Constraints identify limits placed on the software by external hardware, available memory, or other existing systems.

Planning Phase-Resources

- The second planning task is estimation of the resources required to accomplish the software development .
- Resources are the people, reusable software components, and the development environment (hardware and software tools).
- Each resource is specified with four characteristics:
  - Description of the resource.
  - A statement of availability.
  - Time when the resource will be required.
  - Duration of time that the resource will be applied.

Human Resources

- The planner begins by evaluating software scope and selecting the skills required to complete development.
- For relatively small projects (a few person-months), a single individual may perform all software engineering tasks, consulting with specialists as required.
- For larger projects, the software team may be geographically dispersed across a number of different locations.
- The number of people required for a software project can be determined only after an estimate of development effort (e.g., person-months) is made.

Reusable Software Resources

- There are 5 types of components:
- Off-the-shelf components.
  - Existing software that can be acquired from a third party or from a past project.
- COTS (commercial off-the-shelf) components
  - They are purchased from a third party, are ready for use on the current project, and have been fully validated.
- Full-experience components.
  - Existing specifications, designs, code or test data developed for past projects that are similar to the software to be built for the current project.
  - Members of the current software team have had full experience in the application area represented by these components.
  - Therefore, modifications required for full-experience components will be relatively low risk.

Continues..

- Partial-experience components.
  - Existing specifications, designs, code, or test data developed for past projects that are related to the software to be built for the current project but will require substantial modification.
  - Members of the current software team have only limited experience in the application area represented by these components.
  - Therefore, modifications required for partial-experience components have a fair degree of risk
- New components.
  - Software components must be built by the software team specifically for the needs of the current project.

Environmental Resources

- The environment that supports a software project, often called the *software engineering environment* (SEE), incorporates hardware and software.
- Hardware provides a platform that supports the tools (software) required to produce the work products that are an outcome of good software engineering practice.

Software Project Estimation

- Software cost and effort cannot be estimated exactly.
- Too many variables—human, technical, environmental, political may affect the cost and effort estimation.
  - To achieve reliable cost and effort estimates, a numberof options arise: Delay estimation until late in the project (obviously, we can achieve 100 percent accurate estimates after the project is complete!).
  - Base estimates on similar projects that have already been completed.
  - Use relatively simple decomposition techniques to generate project cost and effort estimates.
  - Use one or more empirical models for software cost and effort estimation.

Continues..

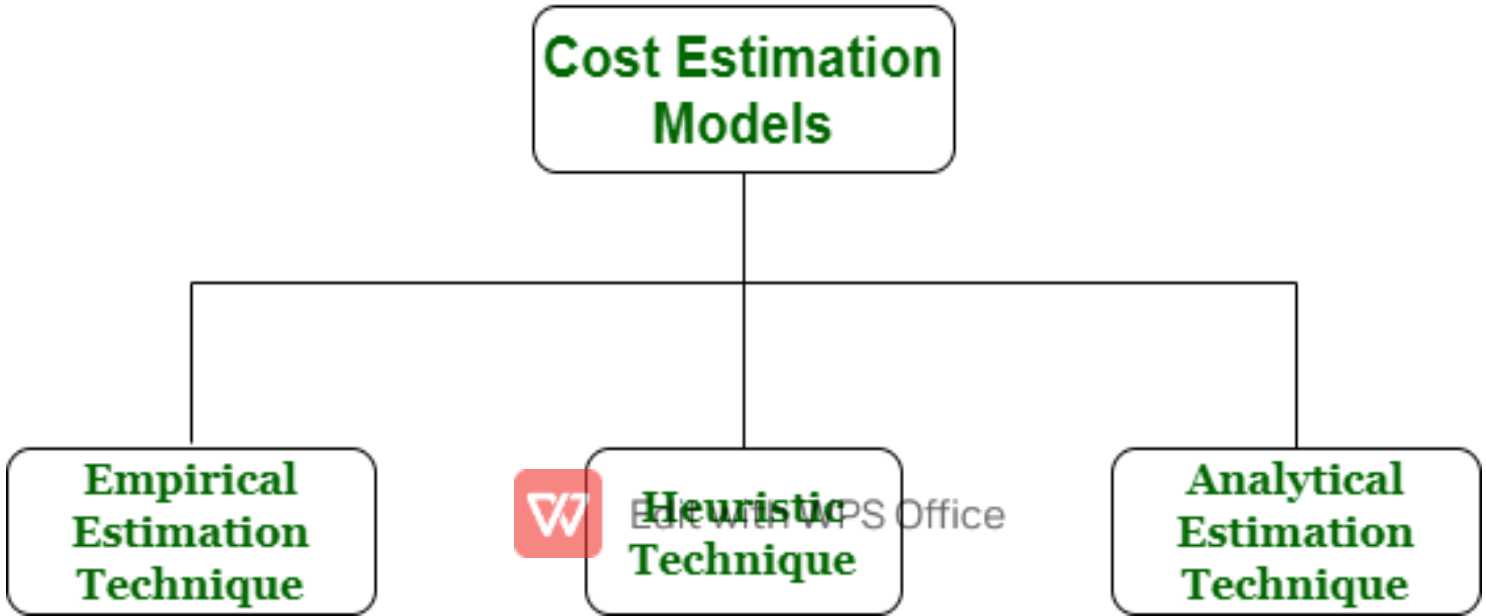
- Option #1 is not practical, but results in good numbers.
- Option #2 can work reasonably well, but it also relies on other project influences being roughly equivalent.
- Options #3 and #4 can be done in tandem to cross check each other.

Contnues..

- Decomposition techniques
  - These take a "divide and conquer" approach.
  - Cost and effort estimation are performed in a stepwise fashion by breaking down a project into major functions and related software engineering activities.
- Empirical estimation models
  - Offer a potentially valuable estimation approach if the historical data used to seed the estimate is good.
- A model is based on experience (historical data) and takes the form
  - $d = f(vi)$
    - where
      - d is one of a number of estimated values (e.g., effort, cost, project duration)
      - vi are selected independent parameters (e.g., estimated LOC or FP).

Continues..

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



Decomposition Techniques

- Before an estimate can be made and decomposition techniques applied, the planner must
  - Understand the scope of the software to be built.
  - Generate an estimate of the software's size.
- Then one of two approaches are used
  - Problem-based estimation
    - Based on either source lines of code or function point estimates.
  - Process-based estimation
    - Based on the effort required to accomplish each task.

Continues..

- Approaches to Software Sizing
- Function point sizing
  - Develop estimates of the information domain characteristics (Product Metrics for Software).
- Standard component sizing
  - Estimate the number of occurrences of each standard component.
  - Use historical project data to determine the delivered LOC size per standard component.
- Change sizing
  - Used when changes are being made to existing software.
  - Estimate the number and type of modifications that must be accomplished.
  - Types of modifications include reuse, adding code, changing code, and deleting code.
  - An effort ratio is then used to estimate each type of change and the size of the change.

Continues..

- Problem-Based Estimation
  1) Start with a bounded statement of scope.
  2) Decompose the software into problem functions that can each be estimated individually.
  3) Compute an LOC or FP value for each function.
  4) Derive cost or effort estimates by applying the LOC or FP values to your baseline productivity metrics (e.g., LOC/person-month or FP/person-month).
  5) Combine function estimates to produce an overall estimate for the entire project.
- In general, the LOC/pm and FP/pm metrics should be computed by project domain
  - Important factors are team size, application area, and complexity
- LOC and FP estimation differ in the level of detail required for decomposition with each value
  - For LOC, decomposition of functions is essential and should go into considerable detail (the more detail, the more accurate the estimate)
  - For FP, decomposition occurs for the five information domain characteristics and the 14 adjustment factors
    - External inputs, external outputs, external inquiries, internal logical files, external interface files

Continues..

- For both approaches, the planner uses lessons learned to estimate an optimistic, most likely, and pessimistic size value for each function or count (for each information domain value).
- Then the expected size value S is computed as follows:

$$S = (S_{opt} + 4S_m + S_{pess})/6$$

- Historical LOC or FP data is then compared to S in order to cross-check it.

Process-Based Estimation

1) Identify the set of functions that the software needs to perform as obtained from the project scope.
2) Identify the series of framework activities that need to be performed for each function.
3) Estimate the effort (in person months) that will be required to accomplish each software process activity for each function.
4) Apply average labor rates (i.e., cost/unit effort) to the effort estimated for each process activity.
5) Compute the total cost and effort for each function and each framework activity.
6) Compare the resulting values to those obtained by way of the LOC and FP estimates
   - If both sets of estimates agree, then your numbers are highly reliable.
   - Otherwise, conduct further investigation and analysis concerning the function and activity breakdown.

- This is the most commonly used of the two estimation techniques (problem and process).

Reconciling Estimates

- The results gathered from the various estimation techniques must be reconciled to produce a single estimate of effort, project duration, and cost.
- If widely divergent estimates occur, investigate the following causes:-
  - The scope of the project is not adequately understood or has been misinterpreted by the planner.
  - Productivity data used for problem-based estimation techniques is inappropriate for the application, obsolete (i.e., outdated for the current organization), or has been misapplied.
- The planner must determine the cause of divergence and then reconcile the estimates.

Empirical Estimation Models

- Estimation models for computer software use empirically derived formulas to predict effort as a function of LOC or FP
- Resultant values computed for LOC or FP are entered into an estimation model
- The empirical data for these models are derived from a limited sample of projects
  - Consequently, the models should be calibrated to reflect local software development conditions

Empirical Estimation

- **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.

Continues..

- 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.

Continues..

- **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 analysing. 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.

COCOMO 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.

Continues..

- 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)

Continues..

- **Levels of COCOMO**
  - **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.

Continues..

- Write the formulas for finding COCOMO.
- Problems

Software Engineering Models

- Introduction
  - For every project one has to follow all the six steps :- requirements, design, development testing, deployment, and maintenance.
  - Depending on the character, size etc., the order, overlap, etc. of the steps may change.

Model Approaches

- From their experience software engineers have developed a lot of different development models.
- They argue that their models are the best models.
- For them, the model which they developed are best.
- There are so many different models because
  - Developers may just be trying to come up with the coolest names and acronyms. Examples are Scrum, Sashimi, RAD, etc.
  - Huge number of people have spent an enormous amount of time on software engineering. Some people noticed that there were problems with the methods they were using.

PREDICTIVE AND ADAPTIVE MODELS

- PREDICTIVE MODEL
  - Predict in advance what needs to be done and then do it.
  - It uses the requirements to design the system, and use the design as a blueprint to write the code and then test the code.

ADAPTIVE MODEL

- Enables you to change the project's goals if necessary during development.
- Lets you periodically reevaluate and decide whether you need to change direction.
- Just gives you chances to fine-tune the project if necessary.

Success and Failure Indicators

- The following list describes some indicators that mean a predictive project may be successful.
  - User involvement.
  - Clear vision.
  - Limited size.
  - Experienced team.
  - Realistic.
  - Established technology.
- The following list describes a few other things that might indicate a predictive project won’t succeed:
  - Incomplete requirements .
  - Unclear requirements .
  - Changing requirements.
  - No resources.

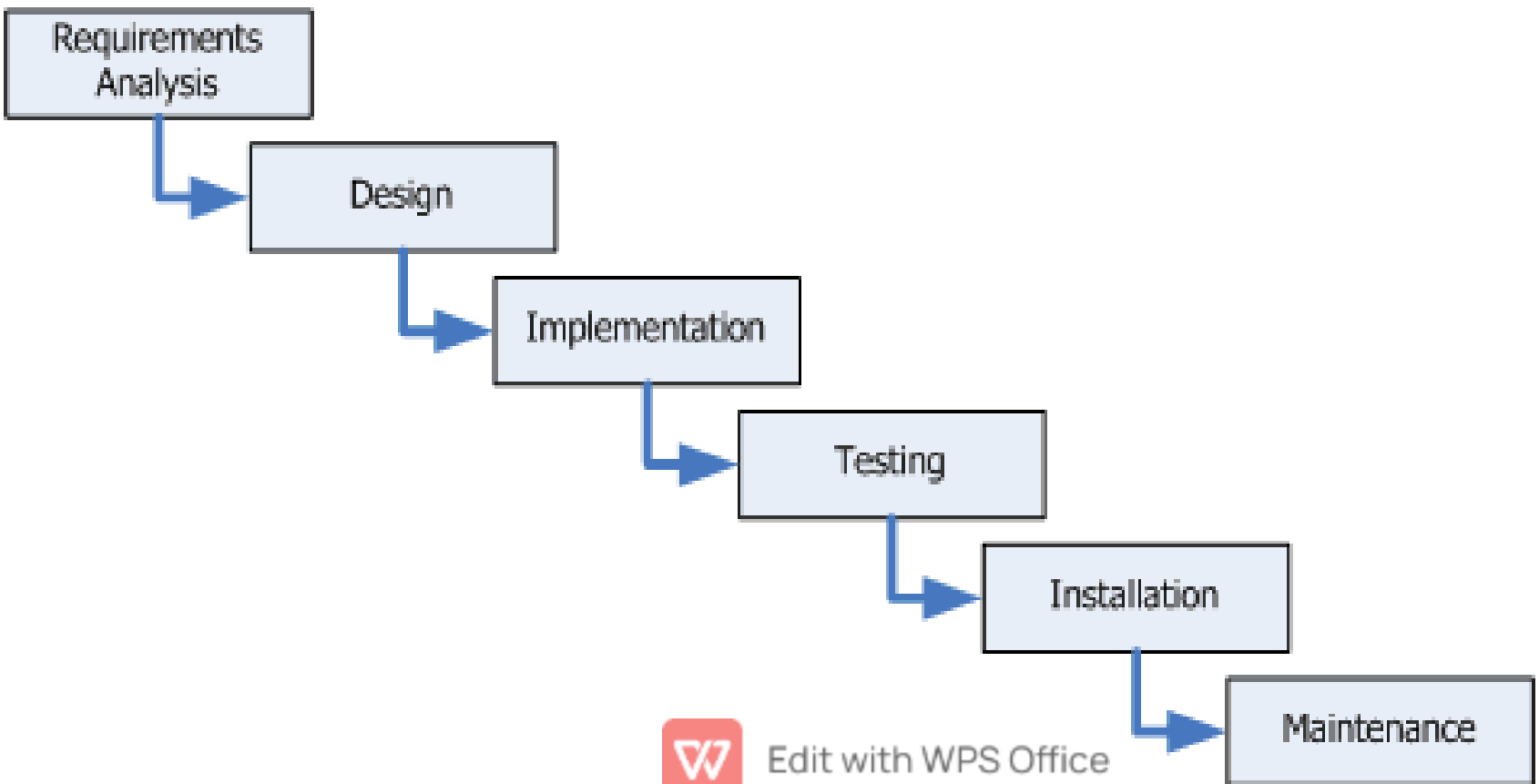
Advantages and Disadvantages

- Advantages
  - Predictability.
  - Stability.
  - Cost-savings .
  - Detailed design.
  - Less refactoring.
  - Fix bugs early.
  - Better documentation.
  - Easy maintenance.
- Disadvantages
  - Inflexible.
  - Later initial release.
  - Big Design Up Front (BDUF).

WATERFALL MODEL

- Predictive model.
- It assumes that you finish each step completely and thoroughly before you move on to the next step.
- The waterfall model can work reasonably well if all the following assumptions are satisfied:
  - The requirements are precisely known in advance.
  - The requirements include no unresolved high-risk items.
  - The requirements won't change much during development.
  - The team has previous experience with similar projects so that they know what's involved in building the application.
  - There's enough time to do everything sequentially.

Continues..



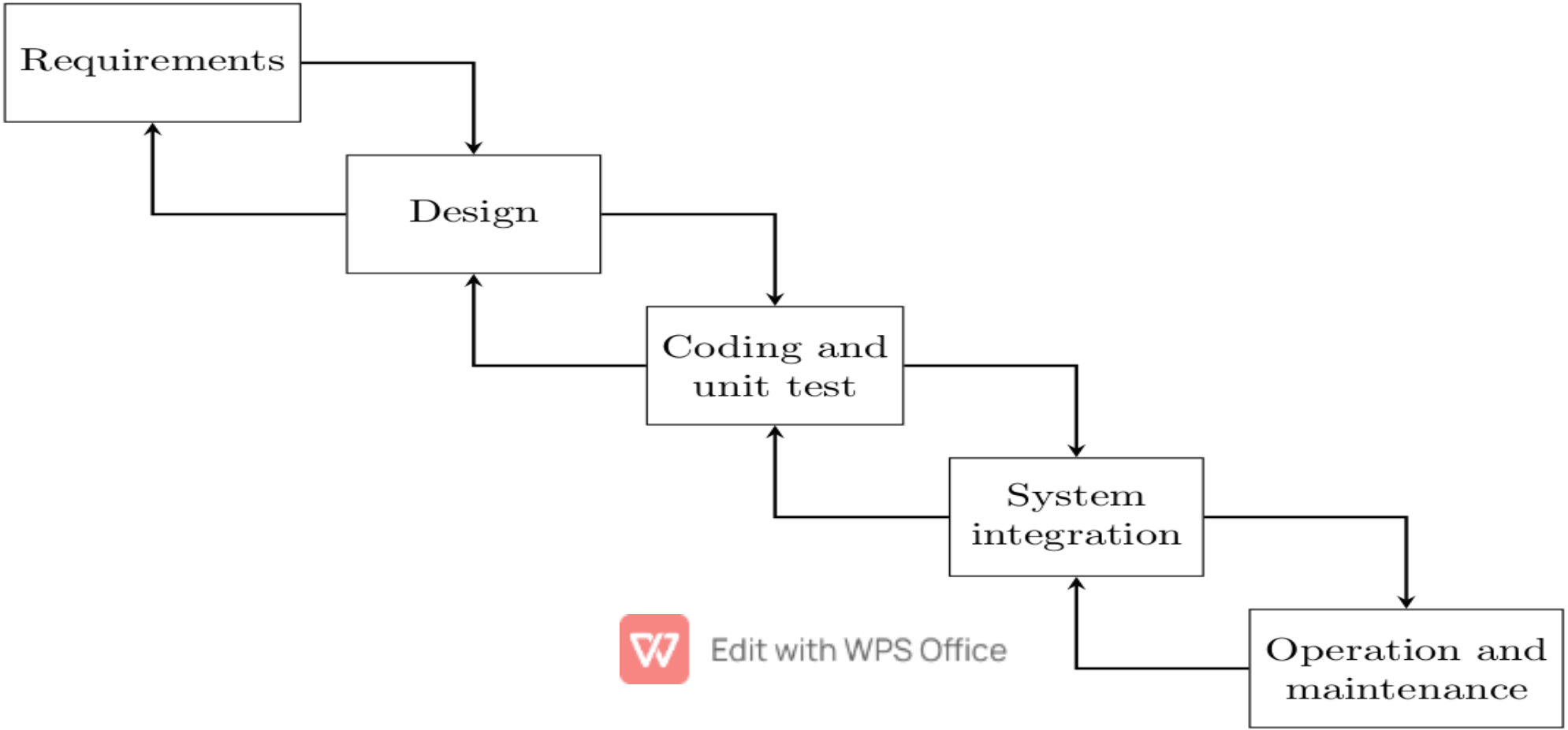
Continues..

- The *waterfall with feedback* variation enables you to move backward from one step to the previous step.
  - If you're working on design and you discover that there was a problem in the requirements, you can briefly go back to the requirements and fix them.
- The farther you have to go back up is little difficult.
  - For example, if you're working on implementation and discover a problem in the requirements, it's hard to skip back up two steps to fix the problem.
- It's meaningless to move back if you find an problem in last steps.
  - For example, if you find a problem during maintenance, then you should probably treat it as a maintenance task instead of moving back into the deployment stage.

SASHIMI

- *Sashimi waterfall or waterfall with overlapping phases.*
- It is similar to the waterfall model except the steps are allowed to overlap.
- Consider a situation, some requirements will be defined while we are still working on others. At that point, some of the team members can start designing the defined features while others continue working on the remaining requirements.
- At a particular time, the design for some parts of the application will be more or less finished, but the design for other parts of the system may not be.
- For that, some developers can start writing code for the designed parts while others continue design tasks.

Continues..



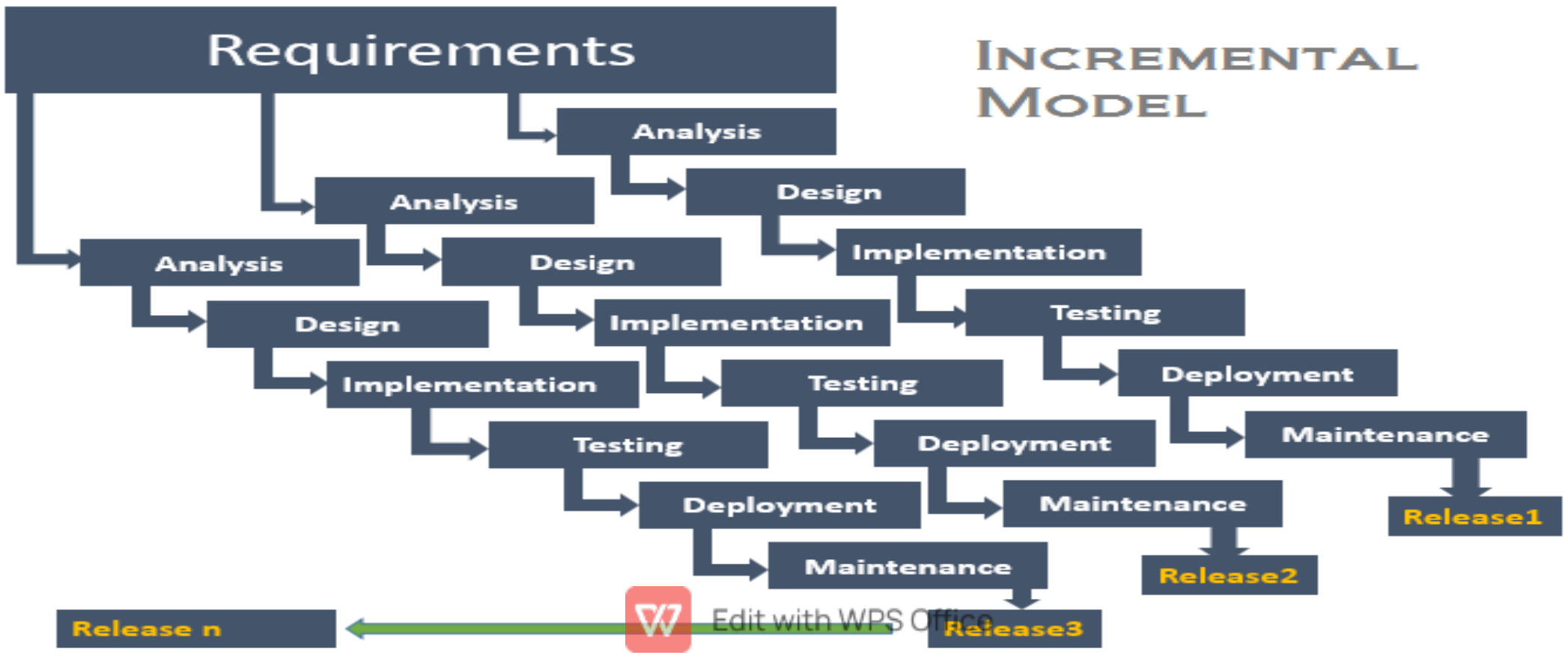
ADVANTAGES OF SASHIMI

- People with different skills can focus on their specialties 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 you discover during design that the requirements are impossible or need alterations, you can make the necessary changes.

INCREMENTAL WATERFALL

- The *incremental waterfall l* model (also called the *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 you’re building the final application incrementally.
- During each increment, you’ll get a better understanding of what the final application should look like.

Continues..


Requirements
INCREMENTAL MODEL
Analysis
Design
Implementation
Testing
Deployment
Maintenance
Release1
Release2
Release3
Release n
Edit with WPS Office

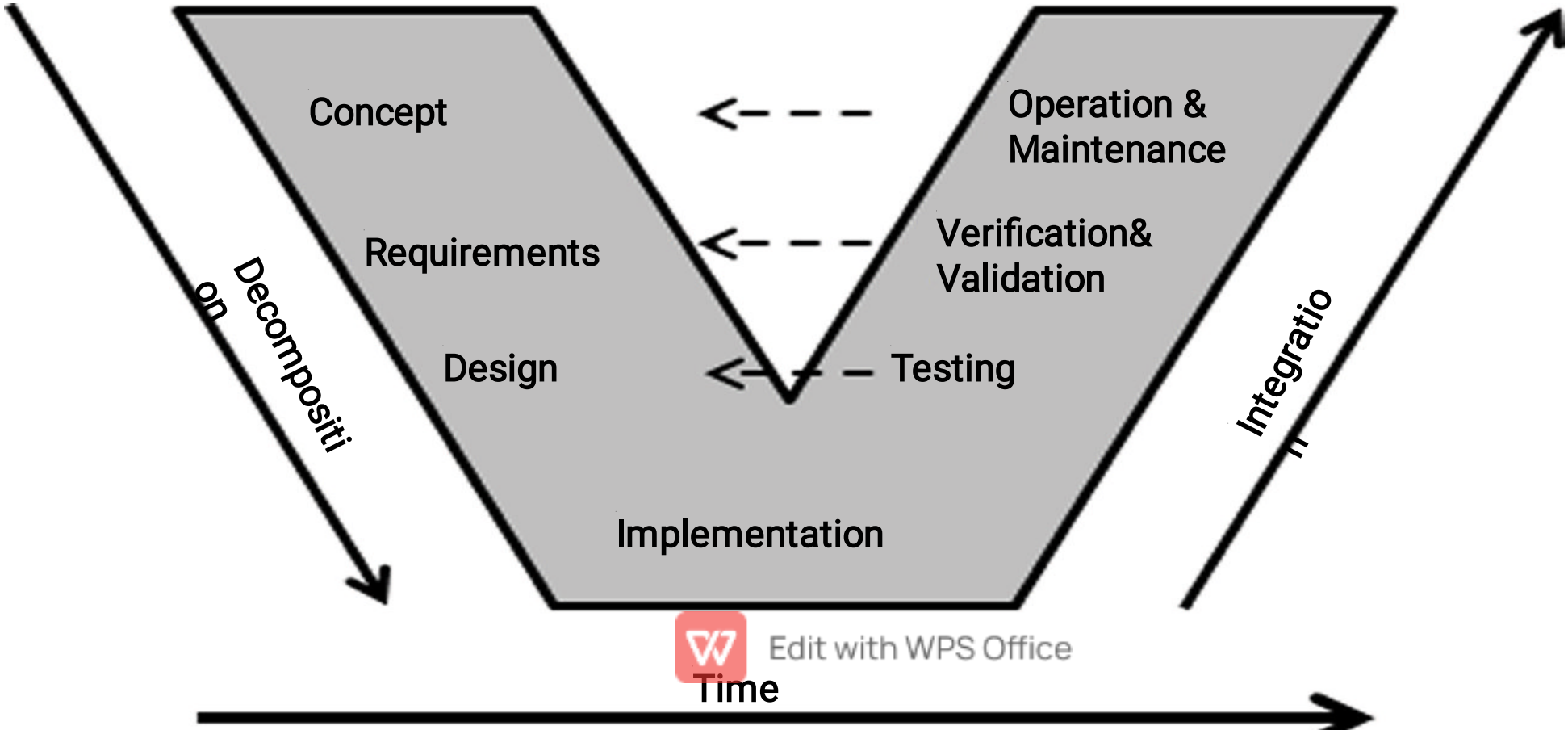
V-MODEL

- *V-model* is a waterfall that's been bent into a V shape.
- The tasks on the left side of the V break the application down from its highest conceptual level into more and more detailed tasks.
- This process of breaking the application down into pieces that you can implement 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 the customers' needs.
- This process of working back up to the conceptual top of the application is called *integration .*

Continues..

- 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 quite directly to verification and validation. Testing confirms that the design worked.

Continues..



ITERATIVE MODELS - PROTOTYPES

- It is a simplified model that demonstrates some behavior that have to study.
- Two important facts about prototypes are
  - They don’t need to work the same way the final application does.
  - They don’t need to implement all the features of the final application.
- After the customers experiment with the prototype, they can give the feedback to help refine the requirements.

Continues..

- Types of Prototypes
  - Throwaway prototypes
    - We use the prototype to study some aspect of the system and then you throw it away and write code from scratch
  - Evolutionary prototypes
    - The prototype demonstrates some of the application's features. As the project progresses, you refine those features and add new ones until the prototype morphs into the finished application.
- Incremental Prototyping
  - The developer build a collection of prototypes of that separately demonstrate the finished application's features.
  - Then combine the prototypes to build the finished application.

Continues..

- Benefits of Prototype model
  - **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.
  - **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 (particularly vertical prototypes) let the developers quickly explore specific pieces of the application to learn what they involve. Prototypes also let developers test different approaches to see which one is best.

Continues..

- Disadvantages of Prototype Models
  - **Narrowing vision** —People 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
  - **Schedule pressure** —If customers see a prototype that they think is mostly done, they may not understand that you need another year to finish and may pressure you to shorten the schedule
  - **Raised expectations** —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. That can make them follow the methods used by that code (or even reuse the code wholesale) even if a better design exists.
  - **Never-ending prototypes** —Throwaway prototypes are supposed to be built relatively quickly to provide fast feedback.
  - Sometimes, developers spend far too much time refining a prototype to make it look better and include more features that aren't actually necessary.

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.

Continues..

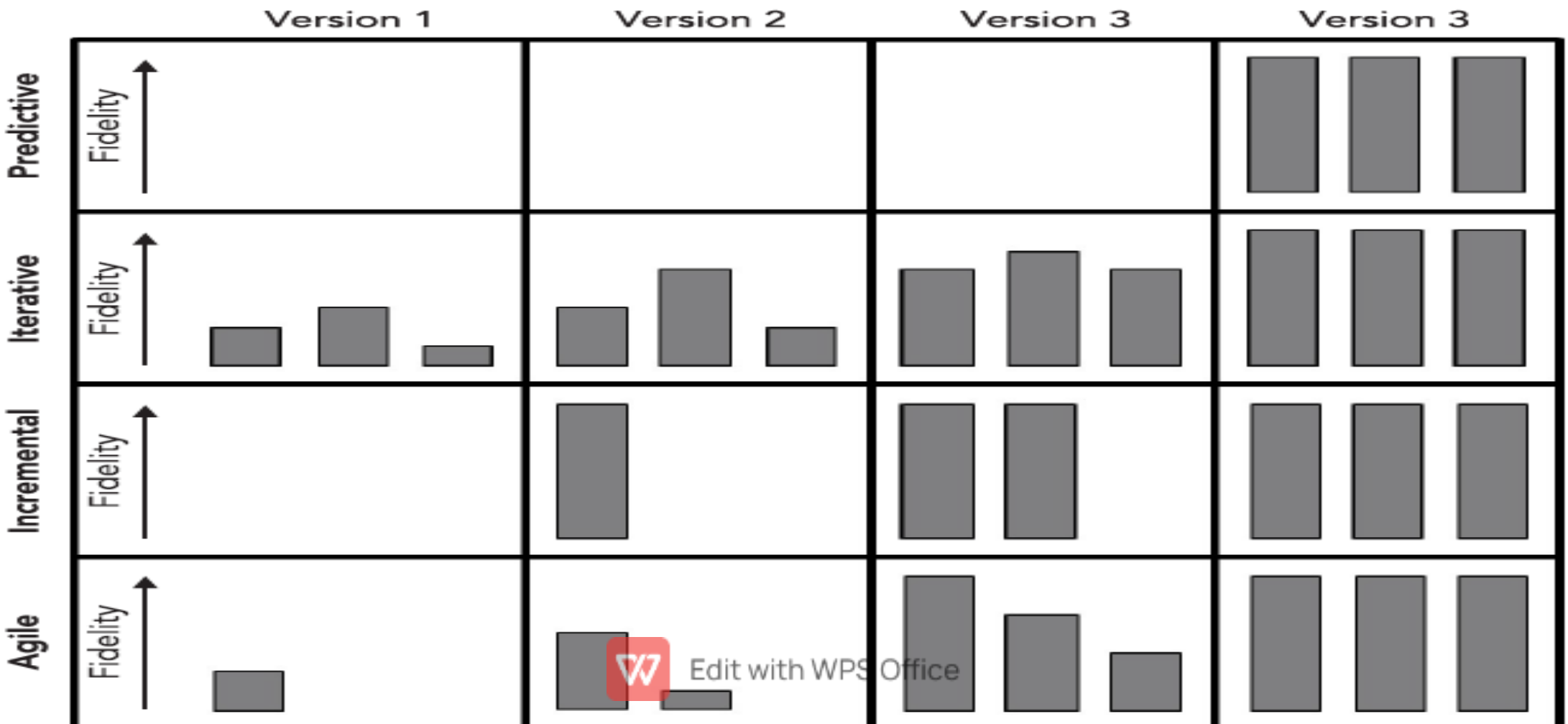
Increment:

- 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.

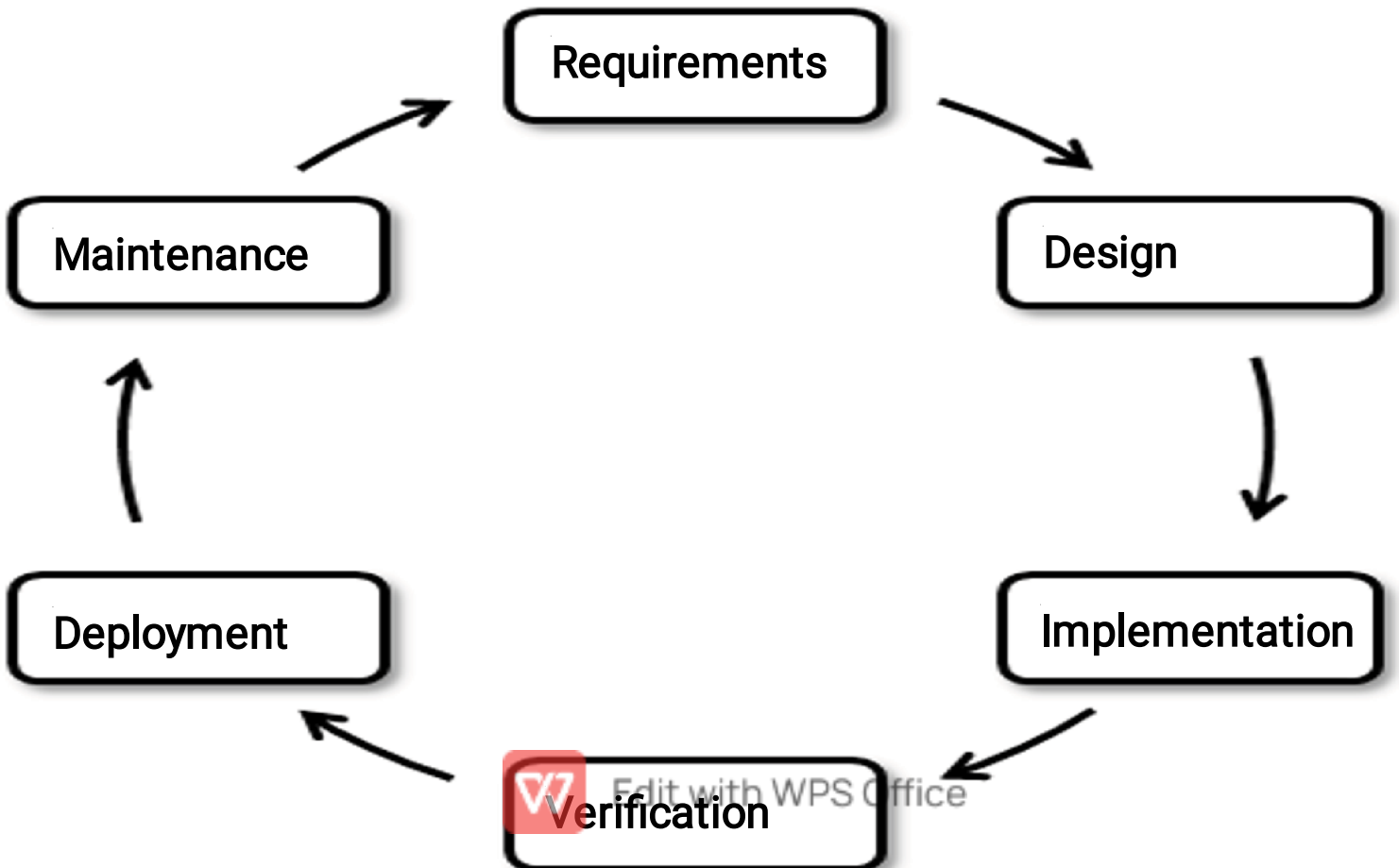
All four of those approaches end with an application that includes all the features at full fidelity. It's the routes they take to get to their final solutions that differ.



SYSTEMS DEVELOPMENT LIFE CYCLE

- SDLC
- Also called The application development life cycle.
- It covers all the tasks that in a software engineering project from start to finish: requirements gathering, design, implementation, verification, deployment and maintenance.
- Waterfall model is one version of SDLC.
- The incremental waterfall model is basically just a series of SDLCs flattened out and possibly with some overlap. So one project starts before the previous one is completely finished.

Continues..



Continues..

The two main ideas of SDLC

- The end of one project can feed directly into the next project in a never-ending circle of life.
- The second new idea is that you can break down the basic steps in a lot more detail if you like.

- Tasks in SDLC are:
  - **Initiation** —An initiator (often a customer, executive champion, or software manager) comes up with the initial idea.
  - **Concept development** —The initiator, usually with help from others who might be interested in the project, explores the concept to see if it's worthwhile and to evaluate possible alternatives.
  - **Preliminary planning** —A project manager (PM) and technical lead are assigned to the project, and they start planning.
    - If it's a big project, the project might be broken into teams and team leads would be assigned.
    - All these leaders make preliminary plans to estimate necessary resources such as staffing, computers, network, development tools etc.

Continues..

- **Requirements analysis** —The team studies the user's needs and creates requirement documents.
  - Those may include text, pictures, use cases, prototypes, and long-winded descriptions of business rules.
  - It may also include UML diagrams showing application structure, user behavior, and anything else that helps the users understand what the team will be building.
- **High-level design** —The team creates high-level designs that specify major subsystems, data flow, database needs, and the rest of the application's high-level structure.
- **Low-level design** —The team creates low-level designs that explain how to build the application's pieces.

Continues..

- **Development** —The team writes the program code and follow good programming practices.
  - They perform unit tests, regression tests, and system tests.
  - They fix the bugs that appear and handle any change requests that are approved by the change committee.
  - The team also prepares user documentation and training materials.
- **Acceptance testing** —The customers finally get a chance to take the application for a test drive in its (almost) final form.
  - After a few bug fixes and perhaps a small change or two, the customers agree that the application satisfies the requirements
- **Deployment** —The team rolls out the application.

Continues..

- **Maintenance** —bug fixes and change requests.
  - 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 or with something completely different.
  - The maintenance team needs to figure out how to upgrade the application to the latest hardware and operating system, and how to dispose of the old hardware.
- **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. During this stage, cleanup crew plans the application's removal and possibly its replacement by something else.