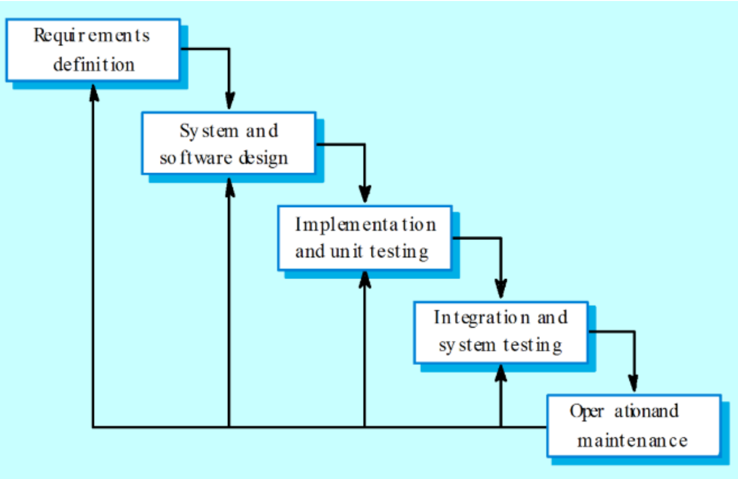
# ***SOFTWARE DESIGN WITH UML***
## ***CIA 1***
## ***UNIT 1***
## ***SOFTWARE DEVELOPMENT PROCESS***

- A **software process model** is an abstract representation of a process. It presents a description of a process. A process is a collection of activities, actions, and tasks that are performed when some work product is to be created
- **Software development life cycle**
  - Communication - understand the stakeholders' objectives for the project and gather requirements
  - Planning - software project plan defines the software engineering work by describing the technical tasks to be conducted
  - Modeling - creating models to better understand software requirements and the design that will achieve requirements
  - Construction - combines code generation and the testing that is required to uncover errors in the code
  - Deployment - the software is delivered to the customers who evaluate the delivered product and provides feedback based on the evaluation
- **Umbrella activities**
  - Software project tracking and control
  - Risk management
  - Technical reviews
  - Software quality assurance
  - Measurement - process, project and product measures
  - Software configuration management
  - Reusability management
  - Work product preparation and production
- **Project** - Temporary venture that exists to produce a defined outcome
  - A project has scope, fixed timeline, project plan, budget, timescale, deliverables, tasks, and resources
  - Not been done before within the organization
- **Process** - It involves a series of related tasks that teams must carry out to achieve a result
  - Regularly repeated
- **Key components of project management**
  - Time
  - Cost
  - Scope
  - Quality
- Software project management focuses on
  - People
  - Product
  - Process
  - Project

- Generic software process models
  - Waterfall model - separate and distinct phases of specification and development
  - Evolutionary development - specification, development and validation are interleaved
  - Component-based software engineering - the system is assembled from existing components

***WATERFALL MODEL:***

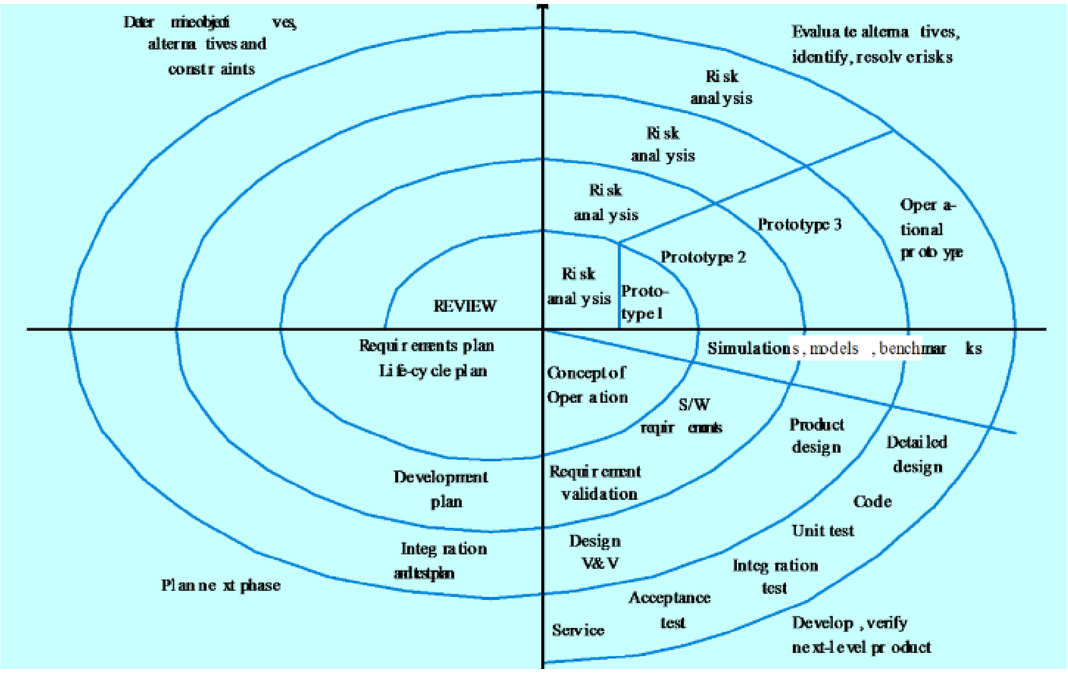
- Phases
  - Requirements analysis and definition
  - System and software design
  - Implementation and unit testing
  - Integration and system testing
  - Deployment and maintenance
- Advantages
  - It is simple to understand and use
  - It functions well for smaller tasks and projects with well-defined requirements
  - It is a dependable and predictable technique for developing software
  - It offers a clear picture of the end product's appearance and functionality
  - It is a sequential, linear strategy that makes it simpler to estimate the time and resources needed for every project phase
- Disadvantages
  - It does not enable end-user feedback
  - A new phase starts only after the prior phase has been completed. But phase should overlap to enhance efficiency and decrease costs
  - It is unsuitable for complicated projects because its linear and sequential nature complicates handling numerous dependencies and interrelated components
  - Testing is usually done toward the end of the development process. Defects cannot be found until late in the development process, which may be costly and time consuming to resolve
  - Inflexibility makes it difficult to respond to changing customer requirements
- When to use
  - Simple or small projects
  - Requirements are known, clear, and fixed
  - Well-documented process
  - Inexperienced team
  - There is no immediate feedback


Requirements definition
System and software design
Implementation and unit testing
Integration and system testing
Oper ationand maintenance

***SPIRAL MODEL:***

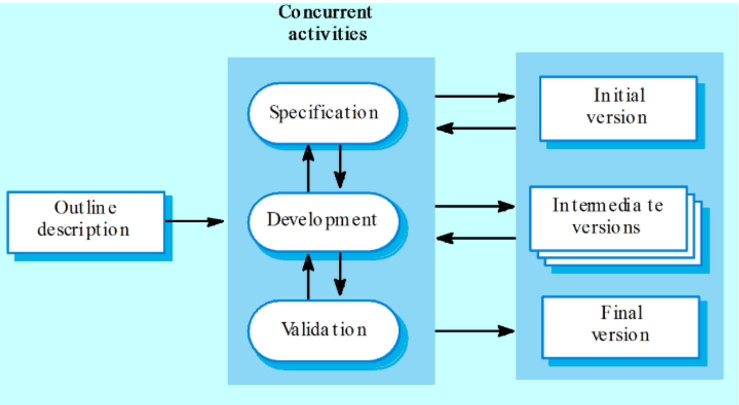
- Evolutionary method
- Each loop represents a phase - loops are chosen depending on what is required
- No fixed phases , loops in spiral are chosen depending on what is required at that time
- Phases
  - Identifying and understanding requirements
  - Performing risk analysis
  - Building the prototype
  - Evaluation of the software's performance

- It is suitable for larger, more complicated projects
- It is very expensive than the waterfall model



## ***EVOLUTIONARY DEVELOPMENT:***

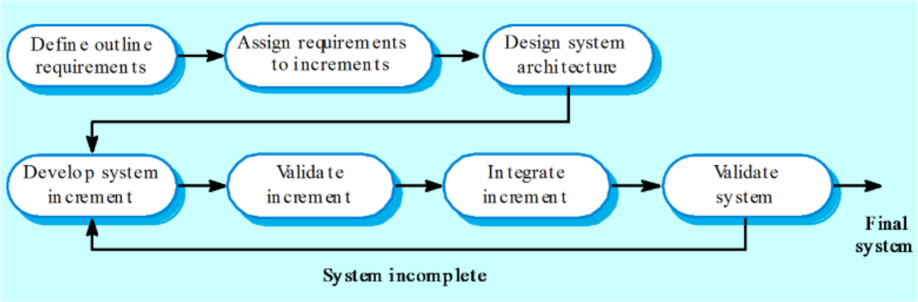
- Exploratory development
  - Objective is to work with customers and to evolve a final system from an initial outline specification
  - Starts with well understood requirements and add new features as proposed by the customer
- Throw-away prototyping
  - Objective is to understand the system requirements
  - Starts with poorly understood requirements to clarify what is really needed
- Problems
  - Lack of process Visibility
  - Systems are poorly structured
  - Special skills like language may be required
- Applicability
  - For small or medium size interactive systems
  - For parts of large systems
  - For short-lifetime systems



## ***INCREMENTAL DEVELOPMENT:***

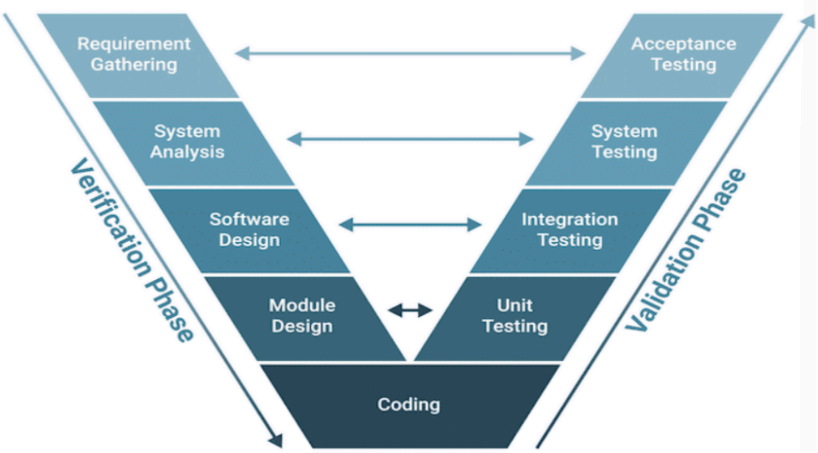
- The development and delivery is broken down into increments with each increment delivering part of the required functionality
- Higher priority requirements are included in early increments

- Advantage:
  - Customer value can be delivered with each increment, so system functionality is available earlier
  - Early increments act as a prototype to help elicit requirements for later increments
  - Lower risk of overall project failure



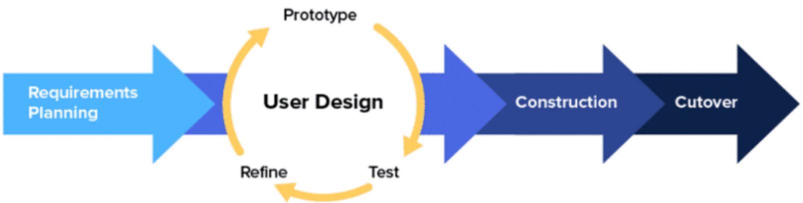
## ***V MODEL:***

- Referred to as verification and validation model
- Introduces testing phases corresponding to each development stage
- It ensures quality and reliability
- This thorough testing approach can extend project timelines and inflate costs



## ***RAPID APPLICATION DEVELOPMENT (RAD) MODEL:***

- Designed for speed and adaptability
- It emphasizes swift prototyping
- Teams concurrently tackle various components, enabling the early development of essential features
- Has iterative cycles, modifications and improvements are integrated



## ***SOFTWARE CRISIS:***

- It refers to the challenges faced in developing efficient and useful computer programs due to increasing complexity and demands
- Problems
  - Limited reusability
  - Complexity overhead
  - Absence of standards

    - Lack of formalization
    - Fail to meet performance metrics
- Causes
    - Project running over budget and time
    - Inefficient software
    - Software of low quality
    - Software did not match user requirements
    - Software never delivered
    - Lack of skilled personnel
    - Code difficult to maintain
- Factors contributing
    - Poor project management
    - Inadequate testing
    - Hardware limitations
- Solutions
    - Adopting agile methodologies
        - It focuses on iterative development, adapting to changing requirements effectively
    - Enhanced skill development
        - Continuous learning and development can equip professionals with up-to-date skills and knowledge
    - Effective project management
        - Robust project management ensures better planning, resource allocation, and timeline management

***RATIONAL UNIFIED PROCESS (RUP)***

- It is an iterative and incremental approach to improving problem knowledge through consecutive revisions
- Phases:
    - Inception - establish the business case for the system
    - Elaboration - develop an understanding of the problem domain and the system architecture
    - Construction - system design, programming and testing
    - Transition - deploy the system in its operating environment
- Advantages
    - Develop software iteratively
    - Manage require
    - Use component-based architecture
    - Visually model software
    - Verify software quality
    - Control changes to software

***AGILE MODEL:***

- Iterative and incremental model where development is carried out in small, manageable units called sprints or iterations
- Breaks task into smaller iterations
- Project scope and requirements are laid down at the beginning of the development process
- It helps to identify and address small issues on projects before they evolve into more significant problem
- It engages business stakeholders to give feedback throughout the development process
- Phases
    - Requirements gathering
    - Design the requirements
    - Construction / iteration
    - Testing / quality assurance
    - Deployment

    - Feedback
- Advantages
    - Iterative
    - Collaborative
    - Adaptive
    - Continuous feedback
    - Frequent delivery
    - Cross functional teams
    - Simplicity
    - Efficient design and fulfills the business requirements
    - Reduced total development time
- Challenges
    - Requires significant cultural change within the organization
    - Can be difficult to scale in large organizations
    - Requires close collaboration, which can be challenging with distributed teams
    - May lead to scope creep if changes are not well managed
- When to use?
    - When frequent changes are required
    - When a highly qualified and experienced team is available
    - When a customer is ready to have a meeting with the software team all the time
    - When project size is small
- Testing models
    - Scrum
    - Crystal
    - Dynamic software development method
    - Feature driven development
    - Lean software development
    - eXtreme programming

## ***LEAN MODEL***

- Principles
    - Eliminate waste
    - Amplify learning
    - Decide as late as possible
    - Deliver as fast as possible
    - Empower the team
    - Build in integrity
    - See the whole
- It is about working only on what must be worked on at the time, no multitasking

## ***DevOps***

- Developers and operations teams work together closely to accelerate innovation and deployment of higher quality and more reliable products and functionalities
- Hallmarks
    - Updates are small and frequent
    - Discipline
    - Continuous feedback and process improvement
    - Automation of manual development processes

## ***SOFTWARE CHARACTERISTICS***

- Functionality
    - Suitability

    - Accuracy
    - Interoperability
    - Compliance
    - security
- Efficiency
    - In time
    - In resource
- Reliability
    - Recoverability
    - Fault tolerance
    - Maturity
- Maintainability
    - Testability
    - Stability
    - Changeability
    - operability
- Portability
    - Adaptability
    - Installability
    - replaceability
- Usability
    - Understandability
    - Learnability
    - operability

## ***OBJECT ORIENTED METHODOLOGIES***

- OO methodologies are set of methods, models and rules for developing systems
- OO methodologies for UML
    - Booch methodology
    - James Rumbagh methodology
    - Ivar Jacobson Methodology
- Object Oriented Analysis
    - Initial phase in the software development process
    - Understanding the problem domain
    - Capturing and modeling the requirements
    - Defining the system's behavior
    - It organizes requirements around objects, which integrate both behaviors (processes) and states (data) modeled after real world objects that the system interacts with
- Object Oriented Design
    - Planning a system of interacting objects to solve a software problem
    - Defining objects, creating class diagram from conceptual diagram
    - Identifying attributes and their models
    - Using design patterns (description of a solution to a common path)
- Features of OO concepts
    - Objects and class
    - Data hiding
    - Data abstraction
    - Inheritance
    - Polymorphism
- OO modeling
    - It divides into two aspects of work

        - Modeling of dynamic behaviors like business processes and use cases
        - Modeling of static structures like classes and components

## GRADY BOOCH APPROACH

- Macro development process
    - Drives the overall business strategy
    - Conceptualization - establish code requirements, goals and develop prototype
    - Analysis and development of the model
    - Design the system architecture
    - Evolution or implementation
    - Maintenance
- Micro development process
    - Support the execution of that strategy
    - Identify classes and objects
    - Identify class and object semantics
    - Identify class and object relationships
    - Identify class and object interfaces and implementation

## RUMBAUGH METHODOLOGIES

- Stages
    - Analysis - determines important properties and domain
    - Systems design - outlines the basic system design, accounts for data storage and concurrency
    - Object design - determines operations and data structures, inheritance and different associations
    - Implementation - conveys design through code
- Model
    - Object model - static, divides the model into objects, concerns itself with classes and their associations with attributes
    - Dynamic model - concerns with interactions between objects through events, states and transitions
    - Functional model - concerts with data flows, data storage, constraints and processes

## JACOBSON METHODOLOGY

- Also known as object oriented software engineering or objectoryx
- Requirements - create problem domain object diagram and specifies use case diagrams
- Analysis - analysis diagrams
- Design - state transition diagrams and interaction diagrams
- Implementation
- Testing

## UML

- Building blocks of UML
    - Things
        - Structural
            - Conceptual or physical elements
            - Class
            - Active class - processes / threads
            - Components - replaceable part, realizes interfaces
            - Interface - collection of externally visible ops
            - Node - computational resource at run-time, processing power with memory
            - Use case - a system service - sequence of interactions with actor
            - Collaboration - chain of responsibility shared by a web of interacting objects, structural and behavioral
        - Behavioral

      - Dynamic parts of UML
      - Interaction - set of objects exchanging messages, to accomplish a specific purpose
      - State machine - specifies the sequence of states an object or an interaction goes through during its lifetime in response to events
    - Grouping things - packages
      - Only exists at development time
      - Variations -framework, models, and subsystems
    - Annotational things - note
      - Explanatory part
  - Relationships
    - Associations
      - Structural relationship that describes a link between objects
      - Variants - aggregation, composition
    - Generalization
      - A specialized element is more specific that the generalized element
    - Realization
      - One element guarantees to carry out what is expected by the other element
      - Between interfaces and class/components
      - Between use cases and collaborations
    - Dependency
      - A change to one thing (independent) may affect the semantics of the other thing (dependent)
  - Diagrams
    - Structural diagrams
      - Class
        - Shows the existence of classes and their relationships
        - Class - collection of objects with common structure, common behavior, common relationships and common semantics
      - Object
      - Component - shows the organizations and dependencies among a set of components
      - Deployment - shows the configuration of run-time processing elements and the software processes living on them
    - Behavioral diagrams
      - Use case
      - Sequence - Displays object interactions arranged in a time sequence
      - Collaboration - Display object interactions organized around objects and their direct links to one another
      -
      - Statechart
      - Activity
        - A special kind of statechart diagram that shows the flow from activity to activity
        - Initial, activity, fork/span, synchronization, condition, final

## ***USE CASE DIAGRAM***

- Presents an outside view of the system
- Actor - someone or something that must interact with the system under development
  - Primary - a user whose goals are fulfilled by the system
  - Secondary / supporting - provides a service to the system
  - Offstage - has an interest in the behavior but is not primary or supporting
- Use case - sequence of interactions between an actor and the system
- <<Uses>> - this relationship shows behavior common to one or more use case
- <<include>> - one use case invoke the behavior defined by another use case

- <<Extends>> - this relationship shows optional / exceptional behavior
- Flow of events - normal flow, and alternate / exceptional flow
- Use cases provide a basis planning and scheduling incremental development and provide a basis for system testing

***INTERACTION DIAGRAM***

- It describes how use cases are realized in terms of interacting objects
- Two types
    - Sequence diagram
    - Collaboration (communication) diagram

***SEQUENCE DIAGRAM***

- Components
    - Life lines
    - Messages - create, delete, self, reply, found, lost, guard
    - Time
    - Delete / destroy
    - Self loop
    - Iteration
    - Recursive
- Message flow notations
    - Synchronous - the sender waits until the responder finishes
    - Asynchronous - the sender doesn't wait for anything from the responder, but it continues its' own activity
    - Return - a message that returns from an object to which a message was previously sent. Return messages are valid only for synchronous messages and are themselves synchronous
- Types of message flow
    - Lost message - the recipient is not known to the system
    - Found message
    - Guards - defines the constraints attached to a system or a particular process, applied for entire branch
    - Iterating - represented by *, or by numbers
    - Conditional - specifies to whether a particular message is sent, based on a condition
- Stereotypes
    - Boundary objects - interface between the system and external entities
    - Control objects - manages the flow of information and system logic
    - Entity objects - represents data or business logic, typically persists data

***VIEWS***

- Use case view
    - Encompasses the behavior as seen by users, analysts and testers
    - Static aspects in use case diagram, dynamic aspects in interaction diagram
- Design view
    - Encompasses classes, interfaces, and collaborations that define the vocabulary of the system
    - Static aspects in class diagram, dynamic aspects in interaction diagram
- Process view
    - Encompasses the threads and processes defining concurrency and synchronization
    - Addresses performance, scalability and throughput
    - Static and dynamic aspects captured as in design view, emphasis on active classes
- Implementation view
    - Encompasses components and files used to assemble and release a physical system
    - Addresses configuration management
    - Static aspects in component diagram, dynamic aspects in interaction diagram

- Deployment view
    - Encompasses the nodes that form the system hardware topology
    - Addresses distribution, delivery, and installation
    - Static aspects in deployment diagram, dynamic aspects in interaction diagram

***DESIGN PATTERNS***

- Repeatable solution to a commonly occurring problem in software design
- Types
    - Creational - designed for class instantiation
    - Structural - designed with regard to a class's structure and composition
    - Behavioral - designed depending on how one class communicates with others

***CREATIONAL DESIGN PATTERN (6)***

- Abstract factory - creates an instance of several families of classes
- Builder - separates object construction from its representation
- Factory method - creates an instance of several derived classes
- Object pool - avoid expensive acquisition and release of resources by recycling objects that are no longer in use
- Prototype - a fully initialized instance to be copied or cloned
- Singleton - a class of which only a single instance can exist

***STRUCTURAL DESIGN PATTERN (8)***

- Adapter - match interfaces of different classes
- Bridge - separates an object's interface from its implementation
- Composite - a tree structure of simple and composite objects
- Decorator - add responsibilities to objects dynamically
- Facade - a single class that represents an entire subsystem
- Flyweight - a fine-grained instance used for efficient sharing
- Private class data - restricts accessor / mutator access
- Proxy - an object representing another object

***BEHAVIORAL DESIGN PATTERN (12)***

- Chain of responsibility - a way of passing a request between a chain of objects
- Command - encapsulate a command request as an object
- Interpreter - a way to include language elements in a program
- Iterator - sequentially access the elements of a collection
- Mediator - defines simplified communication between classes
- Memento - capture and restore an object's internal state
- Null object - designed to act as a default value of an object
- Observer - a way of notifying change to a number of classes
- State - alter an object's behavior when its state changes
- Strategy - encapsulates an algorithm inside a class
- Template method - defer the exact steps of an algorithm to a subclass
- Visitor - defines a new operation to a class without change

***IMPORTANCE OF CHOOSING THE RIGHT DESIGN PATTERN:***

- Scalability
- Flexibility
- Maintainability
- Reusability
- Performance
- Reducing errors