Software Development Life Cycle (SDLC)

Agile Methods

1. Scrum
2. Kanban
3. Lean
4. Extreme Programming (XP)
5. Crystal
6. Feature-Driven Development (FDD)
7. Dynamic Systems Development Method (DSDM)
8. Scrumban
9. Agile Unified Process (AUP)
10. Disciplined Agile Delivery (DAD)
11. Scaled Agile Framework (SAFe)
12. Rapid Application Development (RAD)
13. Agile Modeling (AM)
14. Lean Startup

Non-Agile Methods

1. Waterfall
2. V-Model (Validation and Verification)
3. Spiral Model
4. Iterative and Incremental Development
5. Big Bang Model
6. Rapid Prototyping
7. Joint Application Development (JAD)
8. Rational Unified Process (RUP)
9. Unified Process (UP)
10. Structured Systems Analysis and Design Method (SSADM)
11. Capability Maturity Model Integration (CMMI)
12. Cleanroom Software Engineering
13. Formal Methods

Other Related Methods and Practices

1. DevOps
2. Site Reliability Engineering (SRE)
3. Six Sigma
4. Total Quality Management (TQM)
5. ITIL (Information Technology Infrastructure Library)

 **Iterative Development**:

* Break the project into small, manageable iterations or sprints.
* Each iteration delivers a potentially shippable product increment.

 **Incremental Delivery**:

* Continuously integrate and deliver software in small, incremental releases.
* Ensure each increment adds value and functionality.

**Sprint**:

* A time-boxed period (typically 2-4 weeks) during which work is completed.
* Each sprint results in a potentially shippable product increment.
* Sprints are of consistent duration throughout the project.

 **Product Backlog**:

* An ordered list of everything that might be needed in the product.
* Maintained by the Product Owner.
* Includes features, bug fixes, improvements, and technical tasks.

 **Sprint Backlog**:

* A subset of the Product Backlog items selected for implementation in a sprint.
* Includes a plan for delivering the product increment and achieving the sprint goal.
* Created and managed by the Development Team.

 **Increment**:

* The sum of all Product Backlog items completed during a sprint.
* Represents a potentially shippable product.
* Must meet the team’s Definition of Done (DoD).

**Limit Work in Progress (WIP)**:

* Set WIP limits for each stage to avoid overloading the team.
* Ensures focus on completing tasks before starting new ones.

**Kanban Board**:

* A visual representation of the workflow.
* Columns represent different stages (e.g., To Do, In Progress, Done).
* Swimlanes can be used to separate different types of work or teams.

**Definition of MVP**:

* An MVP is the simplest version of a product that allows a team to collect the maximum amount of validated learning about customers with the least effort.
* It includes only the core features necessary to solve a specific problem or meet a particular need of early adopters.

**Test-Driven Development (TDD)**: Writing automated test cases before writing the code to ensure functionality and facilitate refactoring.

**Continuous Process**

* **Continuous Integration**: Code changes are integrated and tested frequently to detect errors early.
* **Refactoring**: Continuously improving the design of existing code without changing its behavior.
* **Small Releases**: Releasing software in small, frequent increments to get early feedback and quickly deliver value.

**User Stories**: Requirements are captured as user stories, which describe the functionality from the user's perspective. These stories are typically small, manageable pieces of work.

 **Feasibility Study**: Assesses whether the project is viable from a technical and business perspective.

* Determine high-level requirements and risks.
* Develop a feasibility report and an initial project plan.

 **Foundations Phase**: Establishes a firm foundation for the project by defining the most critical aspects.

* Detailed planning and analysis.
* Set up project infrastructure, governance, and resources.
* Develop the outline design and prioritize requirements.

 **Exploration Phase**: Develops the solution iteratively and incrementally.

* Create detailed requirements in small, manageable sections.
* Develop and test components incrementally.
* Regularly review progress and adjust plans as needed.

**MoSCoW Prioritization**: Classifies requirements into four categories: Must have, Should have, Could have, and Won't have. This helps manage scope and ensures critical requirements are met.

**Prototyping**: Creating prototypes to clarify requirements and demonstrate functionality to stakeholders early in the development process.

**Pull System**

* Scrumban employs a pull system for task management. Tasks are pulled into the work in progress (WIP) column based on team capacity, rather than being pushed in all at once.
* This helps to avoid overloading the team and ensures a steady workflow.

**System Design**

* **Objective:** Plan the system’s architecture and design based on the requirements.
* **Activities:**
  + Create system design documents.
  + Define hardware and software requirements.
  + Design system architecture, data models, interfaces, and algorithms.
* **Output:** System Design Document (SDD).

The V-Model, also known as the Validation and Verification Model, is a software development methodology that emphasizes the importance of verification and validation at each phase of the development lifecycle. It is an extension of the Waterfall model and represents the development process in a V-shaped diagram, illustrating how testing activities correspond to each development stage. Here’s a detailed look at how the V-Model works:

**Structure of the V-Model**

**1. Requirements Analysis**

* **Objective**: Gather and analyze the system requirements from the customer.
* **Output**: System Requirements Specification (SRS).
* **Verification**: Ensures that the requirements are clear, complete, and feasible.

**2. System Design**

* **Objective**: Define the system architecture and high-level design based on the SRS.
* **Output**: System Design Document.
* **Verification**: Validates the design against the system requirements.

**3. Architecture Design**

* **Objective**: Break down the system design into smaller modules and define the relationships and dependencies between them.
* **Output**: High-Level Design (HLD) Document.
* **Verification**: Ensures that the architecture meets the system design specifications.

**4. Module Design**

* **Objective**: Design the detailed internal structure of each module.
* **Output**: Low-Level Design (LLD) Document.
* **Verification**: Validates the module designs against the architecture design.

**Identify and Resolve Risks:** Analyze the identified risks, develop strategies for mitigation, and reference design patterns as needed to prevent errors already solved

**Iteration Cycles:**

* **Iteration Planning:** At the beginning of each iteration, plan the tasks and define the goals for that cycle.
* **Development and Testing:** Develop, integrate, and test the features planned for the iteration.

 **Quick Design:**

* Develop a preliminary design for the prototype, focusing on the core functionality.
* Keep the design simple and flexible to allow for easy modifications.
* Use tools such as sketches, storyboards, or wireframes to visualize the design.

 **Building the Prototype:**

* Create a working model of the software with the core features identified during the requirements gathering phase.
* Use rapid development tools and techniques to build the prototype quickly.
* The prototype may be incomplete or limited in scope, but it should be functional enough to provide a tangible representation of the software.

 **Architecture-Centric**:

* RUP places significant emphasis on developing a robust architectural foundation. Early iterations focus on establishing this architecture to support scalability, performance, and maintainability.

  **Use Component Architectures**: Build flexible, reusable software components.

 **Model Visually**: Use visual models to describe software architecture and design.

 **Verify Quality**: Continuously test the software to ensure it meets requirements.

 **Control Changes**: Manage changes systematically to maintain project stability.

**Visually Model Software**: Use UML (Unified Modeling Language) for design and documentation.

**Structured Systems Analysis and Design Method (SSADM)**

**Overview:** Structured Systems Analysis and Design Method (SSADM) is a systems approach to the analysis and design of information systems. It was developed in the United Kingdom in the early 1980s and has since been widely adopted as a standard methodology for system development. SSADM is known for its rigorous and structured approach, ensuring that systems are thoroughly analyzed and designed before implementation.

**Key Characteristics:**

* **Structured Approach:** Emphasizes a detailed and methodical process for system analysis and design.
* **Documentation:** Extensive documentation is a hallmark of SSADM, ensuring clarity and completeness.
* **Stages and Modules:** The methodology is divided into specific stages, each with clearly defined tasks and deliverables.
* **Data-Driven:** Focuses on defining and organizing data requirements before designing processes.

**Stages of SSADM:**

1. **Feasibility Study:**
   * Assess the viability of the proposed system.
   * Identify objectives, scope, and constraints.
   * Provide an initial cost-benefit analysis.
2. **Requirements Analysis:**
   * Gather detailed requirements from stakeholders.
   * Use tools like interviews, questionnaires, and observation.
   * Create a requirements specification document.
3. **Requirements Specification:**
   * Develop a logical model of the system.
   * Define data flow diagrams (DFDs) to represent the flow of information.
   * Create an entity-relationship diagram (ERD) to define data structures.
   * Specify user interface requirements.
4. **Logical System Specification:**
   * Convert requirements into a detailed logical design.
   * Define processing logic, data structures, and interfaces.
   * Ensure consistency and completeness of the logical model.
5. **Physical Design:**
   * Translate the logical design into physical specifications.
   * Select hardware and software platforms.
   * Optimize the system for performance and efficiency.
   * Develop detailed program specifications.

**Techniques and Tools:**

* **Data Flow Diagrams (DFDs):** Visual representation of data processes and flows.
* **Entity-Relationship Diagrams (ERDs):** Illustrates data entities and relationships.
* **Structured English:** A method for specifying process logic using plain English.
* **Decision Trees and Decision Tables:** Tools for defining complex decision logic.
* **Normalization:** A process for organizing data to reduce redundancy.

**Process Components:**

1. **Specification:**
   * Use of formal specification techniques to define what the software is supposed to do. **Formal Specification Languages:** Z, VDM (Vienna Development Method), B-Method
   * The specification serves as a contract between the customer and the developers.
2. **Design:**
   * Development of a detailed design from the formal specification.
   * Emphasis on correctness, simplicity, and understandability.
3. **Verification:**
   * Use of formal verification techniques to ensure the design meets the specifications.
   * Team reviews and inspections are critical parts of the verification process.

**Continuous Integration (CI):**

1. Developers frequently commit code to a shared repository.
2. Automated build and testing processes ensure code changes are integrated seamlessly.

**DevOps Practices**

1. **Version Control:**
   * Use tools like Git for source code management.
   * Track changes and collaborate on code development.
2. **Automated Testing:**
   * Implement unit, integration, and end-to-end tests.
   * Use tools like Jenkins, Travis CI, or CircleCI to automate testing.
3. **Continuous Integration/Continuous Deployment (CI/CD) Pipelines:**
   * Create automated pipelines for code integration, testing, and deployment.
   * Ensure code is always in a deployable state.
4. **Configuration Management:**
   * Use tools like Ansible, Puppet, or Chef to manage system configurations.
   * Maintain consistency across environments.
5. **Containerization:**
   * Use containers (e.g., Docker) to package applications and dependencies.
   * Ensure consistent environments across development, testing, and production.
6. **Orchestration:**
   * Use tools like Kubernetes to manage containerized applications.
   * Automate the deployment, scaling, and management of containerized applications.
7. **Service-Oriented Architecture (SOA)**
8. **Service-Oriented Architecture (SOA)** is an architectural pattern in software design where services are provided to other components by application components, through a network. SOA aims to enhance the flexibility and scalability of systems by enabling the reuse of existing services and the creation of modular applications.

**Unified Process (UP) and Rational Unified Process (RUP)**

* These iterative and incremental development frameworks emphasize the importance of architecture and design early in the project, followed by repeated cycles of development and refinement. They can be adapted to focus on the modular development of game components.

**Incorporate GDLC Phases**

* Structure your development process around the GDLC phases: Concept, Pre-production, Production, Testing, and Post-production.
* Within each phase, apply your iterative cycles of design, build, test, and release.