Unit-II
Notes on Agile Software Development and Requirements Engineering

Agile Software Development

Rapid Software Development:

- Rapid development and delivery have become the most important requirement for software systems because businesses operate in a fast-changing environment, making it difficult to establish stable software requirements.
- Software needs to evolve quickly to reflect these changing business needs.
- Rapid software development involves interleaving specification, design, and implementation.
- The system is developed as a series of versions with stakeholder involvement in version evaluation.

Agile Methods:

- Agile methods emerged due to dissatisfaction with the overheads associated with software design methods prevalent in the 1980s and 1990s.
- These methods prioritize code over design, adopt an iterative approach to software development, and aim to deliver working software quickly while allowing for evolution to accommodate changing requirements.
- The primary objective of agile methods is to reduce overheads in the software development process (e.g., by minimizing documentation) and enable rapid responses to evolving requirements.

Agile Manifesto:

- The Agile Manifesto emphasizes discovering better ways of developing software through practical experience and collaboration.
- It prioritizes:
    - Individuals and interactions over processes and tools
    - Working software over comprehensive documentation
    - Customer collaboration over contract negotiation
    - Responding to change over following a plan

Principles of Agile Methods:

- Customer Involvement: Customers should be closely involved throughout the development process, providing and prioritizing system requirements, and evaluating iterations.
- Incremental Delivery: Software development occurs incrementally, with the customer specifying the requirements for each increment.

- People Not Process: Recognizing and leveraging the skills of the development team is paramount. Team members should have the autonomy to define their own working methods without being restricted by prescriptive processes.
- Embrace Change: System requirements are expected to evolve. Designing the system to accommodate these changes is essential.
- Maintain Simplicity: Focus on simplicity in both the software and the development process. Active efforts should be made to eliminate complexity.

Agile Method Applicability:

- Agile methods are suitable for:
    - Product development where a company is creating a small or medium-sized product for sale.
    - Custom system development within an organization, where there is strong customer commitment to the development process and minimal external constraints from rules and regulations.
- Scaling agile methods to large systems presents challenges due to the emphasis on small, tightly-integrated teams.

Agile Methods and Software Maintenance:

- Since organizations typically allocate more resources to maintaining existing software than to new development, agile methods must support maintenance as effectively as they support original development to be successful.
- Key Issues:
    - Maintainability of systems developed using an agile approach, especially given the reduced emphasis on formal documentation.
    - Effective application of agile methods for evolving a system in response to customer change requests.

Plan-Driven and Agile Specification:

- Plan-based development involves distinct stages for requirements engineering, requirements specification, and design and implementation.
- Agile development combines requirements engineering and design and implementation, allowing for iterations and adjustments based on requirements change requests.

Technical, Human, and Organizational Issues:

- Most projects incorporate elements of both plan-driven and agile processes. Determining the appropriate balance depends on several factors:
    - The Need for Detailed Specification: If a very detailed specification and design are required before implementation, a plan-driven approach may be more suitable.
    - Incremental Delivery and Feedback: If an incremental delivery strategy with rapid feedback from customers is feasible, agile methods should be considered.
    - System Size: Agile methods work best with smaller systems that can be developed by small, co-located teams.
    - System Type: Plan-driven approaches may be necessary for systems requiring extensive analysis before implementation (e.g., real-time systems with complex timing requirements).

    - Expected System Lifetime: Long-lifetime systems may require more design documentation to communicate the original design intent to future support teams.
    - Available Technologies: Agile methods rely heavily on effective tools to track evolving designs.
    - Development Team Organization: Distributed development teams or teams with partially distributed members may face challenges in applying agile methods effectively.
    - Cultural and Organizational Issues: Traditional engineering organizations, accustomed to plan-based development, may resist adopting agile methods.
    - Team Skill Levels: It is argued that agile methods may require higher skill levels compared to plan-based approaches.
    - External Regulation: Systems subject to external regulatory approval (e.g., aviation systems) might necessitate a plan-driven approach with detailed documentation.

Extreme Programming (XP):

- XP is the most widely recognized and implemented agile method.
- It takes an 'extreme' approach to iterative development:
    - New versions can be built multiple times per day.
    - Increments are delivered to customers every two weeks.
    - Every build must pass all tests before acceptance.

XP and Agile Principles:

- Incremental development is facilitated through small, frequent system releases.
- Customer involvement translates to full-time customer engagement with the development team.
- "People Not Process" is implemented through practices like pair programming, collective code ownership, and a sustainable work pace that avoids excessive overtime.
- Change is supported through regular system releases.
- Simplicity is maintained through continuous refactoring of the code.

Extreme Programming Practices:

- Incremental Planning: Requirements are documented on story cards, and the stories for a release are prioritized based on available time. Developers break these stories down into development tasks.
- Small Releases: The minimal set of functionalities that delivers business value is developed first, with subsequent releases incrementally adding functionality.
- Simple Design: Design is limited to meeting current requirements and avoiding unnecessary complexity.
- Test-First Development: Automated unit tests are written for new functionality before the functionality itself is implemented.
- Refactoring: Continuous code refactoring is expected to maintain code simplicity and maintainability.
- Pair Programming: Developers work in pairs, providing mutual support and quality assurance.

- Collective Ownership: Developers collaborate on all areas of the system, fostering shared expertise and responsibility.
- Continuous Integration: Code is integrated into the system as soon as a task is complete. All unit tests must pass after integration.
- Sustainable Pace: Overtime is discouraged to avoid negative impacts on code quality and long-term productivity.
- On-Site Customer: A customer representative is fully engaged with the XP team, providing immediate feedback and clarification on requirements.

Requirements Scenarios:

- In XP, a customer or user representative is a core part of the development team, responsible for making decisions on requirements.
- User requirements are articulated as scenarios or user stories written on cards.
- The development team breaks down these stories into implementation tasks, forming the basis for schedule and cost estimations.
- The customer selects stories for each release based on priorities and schedule estimates.

XP and Change:

- While conventional software engineering wisdom advocates designing for anticipated change, XP takes the stance that predicting changes is not always reliable.
- Instead, it prioritizes continuous code improvement (refactoring) to make accommodating future changes easier.

Examples of Refactoring:

- Reorganizing a class hierarchy to eliminate duplicate code.
- Refining and renaming attributes and methods to improve clarity.
- Replacing inline code with calls to methods consolidated in a program library.

Key Points:

- Agile methods are incremental development methods that focus on rapid development, frequent releases, reducing process overheads, and generating high-quality code. They emphasize direct customer involvement in the development process.
- The decision of whether to use an agile or a plan-driven approach should consider the type of software being developed, the capabilities of the development team, and the culture of the development organization.
- Extreme programming (XP) is a well-known agile method that incorporates good programming practices such as frequent releases, continuous software improvement, and active customer participation.

Test-First Development:

- Writing tests before writing code helps clarify the requirements to be implemented.
- Tests are written as executable programs (often utilizing testing frameworks like JUnit) to enable automated execution and verification.
- All existing and new tests are run automatically when new functionality is integrated, ensuring that the new code has not introduced errors.

Customer Involvement:

- The customer's role in testing focuses on helping develop acceptance tests for the stories to be implemented in the next release.
- Since the customer is an integral part of the XP team, all new code is validated against their needs.
- However, customer representatives often have limited time and may not be able to work full-time with the development team. They may also feel their contribution is primarily defining the requirements.

Test Automation:

- Test automation involves writing tests as executable components before the corresponding functionality is implemented.
- These testing components should be self-contained, simulate the submission of input data, and verify that the output meets the specifications. Automated testing frameworks (e.g., JUnit) simplify the process of writing and executing tests.
- Automated testing ensures that a comprehensive set of tests is always available for quick and easy execution.
- Whenever new functionality is added, running these tests allows for prompt detection and resolution of any problems introduced by the new code.

XP Testing Difficulties:

- Programmers often prefer programming to testing and may take shortcuts when writing tests, leading to incomplete tests that don't cover all potential exceptions.
- Certain tests can be very challenging to write incrementally. For instance, in complex user interfaces, it is often difficult to write unit tests for the code handling display logic and workflow between screens.
- Assessing the completeness of a test suite is difficult. Even with a large number of tests, there is no guarantee that they cover all possible scenarios.

Pair Programming:

- Pair programming involves two programmers working collaboratively at the same workstation to develop software.
- Pair assignments are dynamic, encouraging knowledge sharing among team members and mitigating project risks associated with personnel changes.
- While it might seem inefficient, evidence suggests that pair programming can be more efficient than individual programmers working in isolation.

Advantages of Pair Programming:

- Collective Ownership and Responsibility: It fosters collective ownership and responsibility for the system, preventing individual blame for code issues and promoting team-based problem resolution.
- Informal Review Process: Each line of code is reviewed by at least two people, providing an inherent quality assurance mechanism.

- Support for Refactoring: Pair programming, when combined with collective code ownership, facilitates refactoring and continuous software improvement.

Scrum:

- Scrum is a general agile method that primarily focuses on managing iterative development rather than specific agile practices.
- Phases:
    - Initial Planning: Establish the general objectives for the project and outline the software architecture.
    - Sprint Cycles: A series of iterative cycles, each developing an increment of the system.
    - Project Closure: Wraps up the project, completing required documentation, help frames, and user manuals.
    - Sprint Cycle:
        - Sprints are fixed-length iterations, typically 2-4 weeks, corresponding to the release cycles in XP.
        - Planning starts with the product backlog, a prioritized list of work items for the project.
    - Selection Phase: The entire project team collaborates with the customer to select features and functionalities for the sprint.
    - Development Phase: Once the features are agreed upon, the team focuses on development. During this stage, the team is shielded from external distractions, with all communication channeled through the Scrum master.
    - The Scrum master protects the team from external distractions, ensuring focus on the sprint goals.
    - Sprint Review: At the end of the sprint, the completed work is reviewed and presented to stakeholders. The next sprint cycle then begins.

Teamwork in Scrum:

- The Scrum master is a facilitator who coordinates daily meetings, manages the product backlog, tracks progress, records decisions, and acts as a communication liaison between the team and external stakeholders.
- Daily Meetings: The entire team participates in short daily meetings to share information, report progress, discuss problems, and plan the next day's work.

Scaling Agile Methods:

- Agile methods have proven successful for small to medium-sized projects that can be managed by a small, co-located team.
- Improved communication facilitated by close collaboration is considered a key factor in the success of agile methods.
- Scaling up agile methods for larger, longer projects with multiple development teams, potentially working in different locations, requires modifications to address the complexities of communication and coordination.

Large Systems Development:

- Large systems are typically composed of interconnected subsystems, often developed by separate teams working in different locations and potentially different time zones.

- These are often "brownfield systems" meaning they integrate with and interact with a number of existing legacy systems. Many system requirements relate to this integration, which can limit flexibility and incremental development.
- When integrating multiple systems, a significant portion of the development effort is dedicated to managing the interactions and dependencies between them.

Scaling Up to Large Systems:

- For large systems, it's not feasible to focus solely on code. Upfront design and system documentation become necessary.
- Cross-team communication mechanisms, including regular phone and video conferences, as well as frequent, short electronic meetings, are essential to ensure effective coordination and progress updates.
- While continuous integration, where the entire system is built every time a developer commits a change, is practically impossible in large systems, maintaining frequent integration is crucial.

Scaling Out to Large Companies:

- Project managers without experience in agile methods may be hesitant to embrace the risks associated with a new approach.
- Large organizations often have established quality procedures and standards, which, due to their bureaucratic nature, may not align well with agile methodologies.
- Agile methods tend to work best when team members possess relatively high skill levels. Large organizations may have a wider range of skill sets, potentially impacting the effectiveness of agile practices.
- Cultural resistance to agile methods, particularly within organizations with traditional hierarchical structures, can hinder their adoption.

Key Points:

- A significant strength of extreme programming (XP) is the practice of developing automated tests before implementing features. All tests must pass successfully when an increment is integrated into the system.
- The Scrum method is an agile method that provides a project management framework centred around sprints, fixed-time periods for developing system increments.
- Scaling agile methods for large systems is complex and requires adjustments. Large systems need upfront design and some level of documentation.

Requirements Engineering

Topics Covered:

- Functional and non-functional requirements
- The software requirements document
- Requirements specification
- Requirements engineering processes
- Requirements elicitation and analysis
- Requirements validation
- Requirements management

Requirements Engineering:

- It is the process of defining the services a customer requires from a system, considering the constraints under which it operates and is developed.
- The requirements themselves are descriptions of the system services and constraints generated during this process.

Requirements Abstraction:

- In large software development projects, requirements need to be defined abstractly enough to avoid predefining the solution, allowing multiple contractors to bid on the project.
- The requirements document should provide sufficient detail for both the client and the contractor to understand and validate what the software will do.

Types of Requirement:

- User Requirements: Statements in natural language and diagrams illustrating the system's services and operational constraints, intended for customers.
- System Requirements: A structured document providing detailed descriptions of the system's functions, services, and operational constraints, defining what needs to be implemented and potentially forming part of a contract between the client and the contractor.

Functional and Non-Functional Requirements:

- Functional Requirements: Statements outlining the services the system should provide, its reactions to specific inputs, its behaviour in particular situations, and what it should not do.
- Non-Functional Requirements: Constraints on system services or functionalities, such as timing constraints, development process constraints, standards, etc. Often apply to the entire system rather than individual features.

Functional Requirements:

- They describe system functionalities or services.
- Vary based on software type, expected users, and the system's context of use.
- Functional user requirements are typically high-level statements of system behaviour.
- Functional system requirements describe system services in detail.

Requirements Imprecision:

- Imprecisely stated requirements can lead to misinterpretations.
- Ambiguous requirements can be interpreted differently by developers and users.

Requirements Completeness and Consistency:

- Ideally, requirements should be both complete (including descriptions of all necessary features) and consistent (without conflicting descriptions).
- In practice, achieving a fully complete and consistent requirements document is challenging.

Non-Functional Requirements:

- Define system properties and constraints, such as reliability, response time, storage requirements, I/O device capability, system representations, etc.
- May include process requirements mandating specific development tools, programming languages, or methodologies.
- Can be more critical than functional requirements. If not met, the system may be rendered unusable.

Non-Functional Requirements Implementation:

- Non-functional requirements can influence the overall system architecture rather than just individual components.
- A single non-functional requirement, such as a security requirement, may necessitate numerous related functional requirements to define the necessary system services.

Non-Functional Classifications:

- Product Requirements: Specify the desired behaviour of the delivered product, such as execution speed, reliability, etc.
- Organisational Requirements: Stem from organizational policies and procedures, such as process standards, implementation requirements, etc.
- External Requirements: Arise from factors external to the system and its development process, such as interoperability requirements, legal regulations, etc.

Goals and Requirements:

- Stating non-functional requirements precisely can be difficult, and imprecise requirements can be hard to verify.
- Goal: A general expression of user intent, such as ease of use.
- Verifiable Non-Functional Requirement: A statement using a measurable attribute that can be objectively tested.
- Goals are valuable for conveying user intentions to developers but need to be translated into testable requirements.

Domain Requirements:

- The system's operational domain imposes constraints and requirements on the system.
- Domain requirements may introduce new functional requirements, constrain existing ones, or define specific computations.
- Failure to satisfy domain requirements can render the system ineffective.

Key Points:

- Requirements for a software system define what the system should do and set constraints on its operation and implementation.
- Functional requirements specify the services the system must provide or describe how computations should be performed.
- Non-functional requirements often constrain the system being developed and the development processes employed.

The Software Requirements Document:

- It is the official statement of requirements for the system developers.
- Should encompass both user requirements and system requirements specifications.
- Focuses on WHAT the system should do, not HOW it should be implemented.

Requirements Document Variability:

- The content of the requirements document depends on the type of system and the development approach used.
- Systems developed incrementally typically have less detailed requirements documentation.
- Standards for requirements documents, such as the IEEE standard, are primarily applicable to large-scale systems engineering projects.

Structure of a Requirements Document:

- Preface: Defines the document's readership, its version history, rationale for new versions, and a summary of changes made in each version.
- Introduction: Describes the need for the system, its interaction with other systems, and how it aligns with the organization's business or strategic objectives.
- Glossary: Explains technical terms used in the document.
- User Requirements Definition: Describes the services provided to the user, using natural language, diagrams, or other easily understandable formats. It specifies product and process standards to be adhered to.
- System Architecture: Presents a high-level overview of the anticipated system architecture, outlining the distribution of functions across system modules.
- System Requirements Specification: Details functional and non-functional requirements, including interfaces to other systems.
- System Models: May include graphical models depicting the relationships between system components, its environment, and the data. Examples include object models, data-flow models, and semantic data models.
- System Evolution: Describes the fundamental assumptions underlying the system design and anticipates potential changes.
- Appendices: Provide supplementary information, such as hardware and database requirements, configurations, and data relationships.
- Index: Includes various indexes for navigating the document, such as an alphabetical index, an index of diagrams, an index of functions, etc.

Requirements Specification:

- It is the process of documenting user and system requirements in a requirements document.
- User requirements must be understandable to end-users and customers who lack a technical background.
- System requirements provide more detailed specifications and may include technical information.
- The requirements document may form part of a contract for system development.

Ways of Writing a System Requirements Specification:

- Natural Language: Requirements are articulated using numbered sentences in natural language, with each sentence expressing a single requirement.
- Structured Natural Language: Uses a standardized form or template to present requirements information in natural language.
- Design Description Languages: Employs a language similar to a programming language but with more abstract features to define an operational model of the system. This approach is less common now.
- Graphical Notations: Visual representations like UML use case and sequence diagrams are used to depict the system's functional requirements.
- Mathematical Specifications: Based on mathematical concepts like finite-state machines or sets, these specifications offer unambiguous representation. However, they are often difficult for non-technical stakeholders to understand.

Requirements and Design:

- While requirements should ideally state what the system needs to do and design should describe how, in practice, requirements and design are closely intertwined.
- A system architecture may be predetermined to structure the requirements.
- The need to interoperate with other systems can introduce design constraints.
- Using a particular architecture to meet non-functional requirements can become a domain requirement.
- Regulatory requirements can also impact design choices.

Natural Language Specification:

- Requirements are expressed using natural language sentences, supplemented by diagrams and tables.
- Natural language is commonly used due to its expressiveness, intuitiveness, and universality, making it accessible to a wide range of stakeholders.

Problems with Natural Language:

- Lack of Clarity: Achieving precision without sacrificing readability can be challenging.
- Requirements Confusion: Functional and non-functional requirements can be easily mixed up.
- Requirements Amalgamation: Multiple requirements may be expressed together, making it difficult to isolate individual requirements.

Structured Specifications:

- Structured specifications limit the freedom of the requirements writer, enforcing a standardized format for expressing requirements.
- Well-suited for certain types of requirements, such as those for embedded control systems, but can be overly rigid for business system requirements.

Form-Based Specifications:

- Utilize a predefined form or template to capture requirement details in a structured manner.
- Typical elements include:
    - Definition of the function or entity.

        - Description of inputs and their sources.
        - Description of outputs and their destinations.
        - Information about data and entities used in computations.
        - Description of the action to be taken.
        - Pre and post conditions (if relevant).
        - Side effects (if any).
        - Tabular Specification:
    - Used to supplement natural language specifications.
    - Particularly useful for defining multiple alternative courses of action.
    - Example: A tabular specification can effectively represent the insulin pump's calculations based on blood sugar level changes, outlining insulin dosage requirements for different scenarios.

Requirements Engineering Processes:

- The processes used for requirements engineering (RE) vary significantly depending on the application domain, the stakeholders involved, and the organization developing the requirements.
- Common Activities:
    - Requirements elicitation.
    - Requirements analysis.
    - Requirements validation.
    - Requirements management.
- In practice, RE is an iterative process, interleaving these activities.

Requirements Elicitation and Analysis:

- Also known as requirements elicitation or requirements discovery.
- Involves technical staff working closely with customers to understand the application domain, the desired services, and the operational constraints.
- Stakeholders may include end-users, managers, engineers involved in maintenance, domain experts, trade unions, etc.
- Stages:
    - Requirements discovery.
    - Requirements classification and organization.
    - Requirements prioritization and negotiation.
    - Requirements specification.

Process Activities:

- Requirements Discovery: Interacting with stakeholders to identify their requirements. Domain requirements are also uncovered at this stage.
- Requirements Classification and Organization: Grouping related requirements into coherent clusters.
- Prioritization and Negotiation: Prioritizing requirements and resolving conflicts.
- Requirements Specification: Documenting requirements for input into the next iteration.

Requirements Discovery:

- Involves gathering information about the required system, as well as existing systems, and extracting user and system requirements.
- Interaction occurs with a range of stakeholders, from managers to external regulators.
- Most systems have a diverse set of stakeholders.

Interviewing:

- Formal or informal interviews with stakeholders are integral to most RE processes.
- Types of Interview:
  - Closed Interviews: Based on a pre-determined set of questions.
  - Open Interviews: Explore various issues with stakeholders in a less structured format.
- Effective Interviewing:
  - Maintain an open mind, avoid preconceived notions about requirements, and actively listen to stakeholders.
  - Use prompts to stimulate discussions and gather insights.

Interviews in Practice:

- Typically involve a mix of closed and open-ended questions.
- Effective for gaining a general understanding of stakeholder roles and potential system interactions.
- Not ideal for understanding domain requirements, as requirements engineers may not fully grasp specific domain terminology, and stakeholders might not articulate familiar domain knowledge.

Scenarios:

- Scenarios are real-life examples illustrating system usage.
- Elements:
  - Description of the starting situation.
  - Description of the normal flow of events.
  - Description of potential issues or exceptions.
  - Information about concurrent activities.
  - Description of the state upon scenario completion.

Use Cases:

- Use case diagrams provide a visual representation of interactions between actors (users or external systems) and the system.
- Each use case represents a specific functionality or goal that an actor wants to achieve through interaction with the system.

Ethnography:

- Involves a social scientist spending a significant amount of time observing and analyzing actual work practices.
- Benefits:
  - Observes work practices without requiring explicit explanations.
  - Captures important social and organizational factors.

- Reveals the richness and complexity of work often missed by simplified system models.

Scope of Ethnography:

- Focuses on understanding requirements based on observed work practices, which may deviate from formally defined processes.
- Derives requirements from understanding cooperation and awareness of other people's activities.
- Effective for understanding existing processes but may not be suitable for identifying new system features.

Requirements Validation:

- Aims to demonstrate that the defined requirements truly reflect the customer's needs.
- Crucial because the cost of fixing requirements errors after delivery is significantly higher than fixing implementation errors.

Requirements Checking:

- Validity: Does the system provide functions that effectively support the customer's needs?
- Consistency: Are there any conflicting requirements?
- Completeness: Are all required functions included?
- Realism: Can the requirements be implemented within the constraints of budget and technology?
- Verifiability: Can the requirements be checked or tested?

Requirements Validation Techniques:

- Requirements Reviews: Systematic manual analysis of the requirements.
- Prototyping: Using an executable model to validate requirements.
- Test-Case Generation: Developing tests for requirements to assess testability.

Requirements Reviews:

- Regular reviews are essential during the requirements definition process.
- Involve both client and contractor staff.
- Can be formal (using completed documents) or informal. Effective communication between developers, customers, and users can help resolve issues early.

Review Checks:

- Verifiability: Is the requirement realistically testable?
- Comprehensibility: Is the requirement clearly understood by all stakeholders?
- Traceability: Is the source of the requirement clearly documented?
- Adaptability: Can the requirement be modified without significantly impacting other requirements?

Requirements Management:

- It is the process of managing changes to requirements throughout the requirements engineering process and system development.

- New

Question Bank

Agile Development

- What are the core principles of agile development and how do they differ from plan-driven approaches? Consider the emphasis on individuals and interactions, working software, customer collaboration, and responding to change. . How is this reflected in the process of requirements specification?
- What are the advantages and disadvantages of agile methods? Explore the benefits of rapid development and flexibility in meeting changing needs. Contrast these with potential challenges such as maintaining customer interest, team suitability, and managing change priorities.
- How do agile methods address the need for software maintenance? Given the emphasis on minimizing formal documentation in agile, discuss the issues of maintainability and the effectiveness of agile methods in system evolution in response to change requests.
- When are agile methods most applicable and what are the considerations for choosing between agile and plan-driven development? Consider factors such as project size, customer commitment, external regulations, team capabilities, company culture, and system lifetime.
- What are the key practices of Extreme Programming (XP) and how do they support agile principles? Discuss practices like frequent system releases, customer involvement, pair programming, collective ownership, and refactoring. Explore the concept of user stories and their role in requirements gathering and estimation.
- Explain the concept of refactoring in XP and provide examples of how it is used to maintain code simplicity. Discuss the rationale for refactoring as a response to unpredictable changes and its role in making future changes easier.
- How does testing work in XP and what are the benefits and challenges of the XP testing approach? Describe features such as test-first development, incremental test development, user involvement, and automated test harnesses. Analyse the potential difficulties, including programmer preferences, test complexity, and judging test completeness.
- What is pair programming and what are its advantages in XP? Examine its impact on collective ownership and responsibility, its role as an informal review process, and its support for refactoring
- Describe the Scrum approach to agile project management. Explain its focus on managing iterative development and the three phases involved: outline planning, sprint cycles, and project closure.
- Discuss the Sprint cycle, the product backlog, and the selection process.
- How is teamwork facilitated in Scrum? Explain the roles of the Scrum master and the structure of daily meetings for information sharing and progress updates.
- What are the challenges in scaling agile methods to large systems and organizations? Discuss the impact of larger teams, distributed development, and the need for up-front design and documentation.

Requirements Engineering

- What is requirements engineering and what are the key activities involved? Define the process of establishing system services and constraints and the different types of requirements, such as user and system requirements. Discuss the activities of requirements elicitation, analysis, validation, and management.
- What are the challenges in achieving requirements completeness and consistency? Explore the reasons why it is practically impossible to produce a complete and consistent requirements document.
- What are functional and non-functional requirements? Provide examples of each type and explain how they can be documented.
- Why is requirements validation important and what are the potential consequences of requirements errors? Highlight the significant cost implications of fixing requirements errors after delivery compared to implementation errors.
- Describe different techniques for requirements validation. Discuss techniques like requirements reviews, prototyping, and test-case generation.
- Explain the importance of review checks for verifiability, comprehensibility, traceability, and adaptability.
- Explain the concept of domain requirements and the challenges they present. Discuss their origin, potential impact if not satisfied, and the difficulties in understanding and making them explicit.
- What is the purpose and typical structure of a software requirements document? Discuss the key information it should include and the different users of the document and their specific needs.
- Discuss the different ways of writing a system requirements specification. Explain the use of natural language, structured natural language, design description languages, graphical notations, and mathematical specifications. Evaluate the strengths and weaknesses of each approach.
- What are the problems with using natural language for requirements specification? Discuss issues like lack of clarity, requirements confusion, and requirements amalgamation.
- What are structured specifications and what are their advantages and disadvantages? Explain the concept and consider their suitability for different types of systems.
- How can tabular specifications be used to supplement natural language in requirements documents? Provide examples to illustrate their use in defining alternative courses of action.
- What is requirements management and why is it important in software development? Define the process and discuss the need to track requirements, manage changes, and assess the impact of evolving requirements.
- Explain the process of requirements change management. Describe the steps involved, from problem analysis and change specification to change analysis and costing, and finally, change implementation.
- What are the different techniques for requirements elicitation? Describe the techniques of interviewing, scenarios, use-cases, and ethnography, providing examples where relevant. Evaluate the strengths and limitations of each technique

.