SE WINTER 22 (BTCOC503) SOLVED	
Q.NO	QUESTIONS
Q1 A	What are the four important attributes that all professional software should have? state and explain.
ANS:	Professional software should ideally possess several attributes to ensure its effectiveness, reliability, and suitability for its intended purpose. Four important attributes include: **1. Functionality:** • **Definition**: Refers to the ability of the software to perform the functions or tasks it was designed for. • **Explanation**: The software should meet the specified requirements and fulfill the needs of its users. It should perform all the intended tasks accurately and effectively, providing the expected features and capabilities. **2. Reliability:** • **Definition**: Indicates the software's ability to consistently perform as expected under specified conditions. • **Explanation**: Reliable software should operate without failure, errors, or unexpected behavior. It should produce consistent results even when facing varying inputs or usage scenarios. **3. Usability:** • **Definition**: Refers to how easily and effectively users can interact with and use the software to accomplish their tasks. • **Explanation**: Usable software features an intuitive and user-friendly interface, minimizing the learning curve and allowing users to navigate and utilize its functionalities efficiently. **4. Maintainability:** • **Definition**: Indicates how easily the software can be modified, updated, or extended after deployment. • **Explanation**: Maintainable software is designed with clear and modular code, facilitating modifications or enhancements without causing unintended side effects. It allows for easier debugging, updating, and scaling as needed. **Importance of these Attributes:** • **Customer Satisfaction**: Ensures that the software meets user needs, functions reliably, and is easy to use. • **Reduced Risks**: Enhances software quality, reducing the chances of errors, failures, or user dissatisfaction. • **Adaptability and Scalability**: Facilitates changes and updates, allowing the software to evolve with changing requirements or technology advancements. Professional software should strive to balance and incorporate these attributes to ensure high-quality, reliable, and user-centric products that align with user expectations and needs. Achieving these attributes requires careful planning, thorough testing, and continuous improvement throughout the software development lifecycle.
Q1 B	Explain software Process activities.
ANS:	Software process activities encompass the various phases, steps, or tasks involved in developing software from conception to deployment and maintenance. These activities form the backbone of the software development lifecycle. Here are the primary software process activities: **1. Requirements Engineering:** • **Purpose**: Involves understanding and documenting user needs, functionalities, and constraints. • **Activities**: Requirement elicitation, analysis, specification, validation, and management. **2. Design and Architecture:** • **Purpose**: Translating requirements into a blueprint or plan for software structure and behavior. • **Activities**: Architectural design, detailed design, interface design, database design, and component specification. **3. Implementation:**

	• **Purpose**: Actual coding or building of the software based on the design specifications. • **Activities**: Writing code, creating modules, integrating components, and conducting unit testing. **4. Testing:** • **Purpose**: Evaluating software to identify defects, errors, and ensure it meets specified requirements. • **Activities**: Test planning, test case development, execution, defect tracking, and regression testing. **5. Deployment:** • **Purpose**: Releasing the software for users or clients to access and use. • **Activities**: Installation, configuration, data migration, user training, and release management. **6. Maintenance:** • **Purpose**: Addressing bugs, enhancing features, and supporting users after deployment. • **Activities**: Bug fixing, updates, upgrades, user support, and performance monitoring. **7. Documentation:** • **Purpose**: Recording and maintaining information related to the software's design, usage, and maintenance. • **Activities**: Writing technical documents, user manuals, system documentation, and code comments. **8. Process Management and Improvement:** • **Purpose**: Monitoring and optimizing the software development process itself. • **Activities**: Process assessment, metrics collection, process improvement initiatives, and adoption of best practices. **Importance of Software Process Activities:** • **Structured Development**: Provides a systematic approach to software development, ensuring consistency and quality. • **Error Reduction**: Helps in identifying and rectifying issues at early stages, reducing risks and costs. • **Efficiency and Quality**: Enhances productivity, promotes better collaboration, and leads to higher-quality software. Each of these activities is essential and interrelated, contributing to the overall success of a software project. The effectiveness and efficiency of these activities significantly impact the quality, reliability, and success of the final software product.
Q1 C	Explain the software Engineering code of ethics and professional practice
ANS:	The Software Engineering Code of Ethics and Professional Practice outlines ethical principles and guidelines to guide the conduct of software engineers. Developed by the IEEE Computer Society and ACM (Association for Computing Machinery), it's a set of ethical norms and responsibilities that software engineers should uphold in their professional practice. The code is structured around eight primary principles: **1. Public:** • Software engineers should act consistently with the public interest. They must ensure their work benefits society, avoids harm, and prioritizes public safety and well-being. **2. Client and Employer:** • Engineers should act in the best interests of their clients and employers, ensuring professional integrity and maintaining confidentiality. They should provide accurate and truthful representations concerning the software or systems they develop. **3. Product:** • Engineers must ensure the products they develop meet high professional standards. This includes quality, reliability, maintainability, and performance. **4. Judgment:** • Engineers should make fair and impartial judgments when designing and testing software systems, avoiding bias and ensuring decisions are based on objective analysis. **5. Management:** • Engineers should promote an ethical approach to software development and management. This includes advocating for policies that enhance the profession's integrity and promoting responsible decision-making within their organizations. **6. Profession:**

	• Engineers should enhance their professionalism through continuous learning, sharing knowledge, and supporting the growth and integrity of the software engineering field. **7. Colleagues:** • Engineers should be fair and supportive to their colleagues, providing a work environment that is respectful, inclusive, and conducive to professional growth. **8. Self:** • Engineers should maintain and improve their professional competence and uphold high ethical standards. They should avoid conflicts of interest and be honest about their qualifications and capabilities. **Importance of the Code:** • **Professionalism**: Upholds professional conduct and standards, fostering trust in the software engineering field. • **User Protection**: Prioritizes user safety, privacy, and well-being, emphasizing responsible and ethical software development. • **Continuous Improvement**: Encourages lifelong learning and professional growth among software engineers. Following this code of ethics is crucial in maintaining trust, integrity, and accountability within the software engineering community. It sets a standard for professional behavior, ensuring that software engineers consider not only the technical aspects of their work but also the ethical implications and societal impact of the software they develop.
Q2 A	Compare Plan driven approach and Agile methods.
ANS:	**Plan-Driven Approach:** 1. **Predictive and Sequential**: • **Characteristic**: Emphasizes detailed planning and documentation before development begins. The entire project is mapped out in advance in a sequential manner. • **Focus**: Comprehensive documentation, detailed requirements gathering, and a fixed plan to follow throughout the project. 2. **Extensive Upfront Planning**: • **Characteristics**: Requires extensive requirements gathering and documentation in the early stages of the project. A detailed project plan, often with a rigid schedule and scope, is created. • **Advantages**: Clarity in project scope, milestones, and deliverables at the outset. 3. **Rigidity**: • **Characteristics**: Less flexibility once the project starts. Changes to requirements or scope can be challenging to accommodate and might lead to delays or increased costs. • **Advantages**: Clear structure and predictability in terms of timelines and deliverables. 4. **Waterfall Model**: • **Example**: Follows a linear sequence of phases (requirements, design, implementation, testing, deployment) where each phase relies on the completion of the previous one. **Agile Methods:** 1. **Adaptive and Iterative**: • **Characteristics**: Emphasizes adaptability and flexibility. Projects are broken down into smaller increments or iterations, allowing for continuous feedback and adaptation. • **Focus**: Delivering working software in smaller, incremental cycles with frequent reassessment and adaptation. 2. **Iterative Development**: • **Characteristics**: Embraces iterations or sprints, typically lasting a few weeks. Each iteration delivers a working piece of software, incorporating user feedback for future iterations. • **Advantages**: Flexibility to accommodate changing requirements and priorities, allowing for quicker responses to market changes. 3. **Customer Collaboration**:

	• **Characteristics**: Encourages frequent collaboration and feedback from customers or stakeholders throughout the development process. • **Advantages**: Increased customer satisfaction and alignment with evolving needs. 4. **Scrum, Kanban, XP**: • **Examples**: Agile methodologies like Scrum, Kanban, or Extreme Programming (XP) promote adaptability, collaboration, and iterative development. **Comparison:** • **Flexibility**: Plan-driven approaches are more rigid, whereas Agile methods are adaptive and flexible, allowing for changes throughout the project. • **Documentation vs. Adaptation**: Plan-driven approaches emphasize extensive documentation and planning, while Agile focuses on adapting to changes and delivering working software iteratively. • **Customer Involvement**: Agile methods stress frequent customer collaboration and feedback, ensuring alignment with customer needs, while plan-driven approaches often involve less frequent interactions with stakeholders. Both approaches have their strengths and weaknesses, and the choice between them often depends on the project's nature, requirements, and the organization's culture and preferences. Some projects might benefit from a more predictive and structured approach, while others might thrive with the adaptability and flexibility offered by Agile methods.
Q2 B	An automated ticket-issuing system sells rail tickets. Users select their destination and input a credit card and a personal identification number. The rail ticket is issued, and their credit card account charged. When the user presses the start button, a menu display of potential destinations is activated. along with a message to the user to select a destination. Once a destination has been selected, users are requested to input their credit card. Its validity is checked, and the user is then requested to input a personal identifier. When the credit transaction has been validated, the ticket is issued. For above case study find functional and non-functional requirements.
ANS:	**Functional Requirements:** 1. **Destination Selection**: • Users should be able to select their destination from a menu displayed after pressing the start button. • The system should prompt the user to choose a destination from the available options. 2. **Credit Card Input**: • Users should input their credit card information for payment. • The system should validate the credit card details provided by the user. 3. **Personal Identification Input**: • Users should input a personal identifier after credit card validation. • The system should verify the personal identifier for transaction validation. 4. **Ticket Issuance**: • After successful credit card validation and personal identification, the system should issue the rail ticket to the user. **Non-Functional Requirements:** 1. **Usability**: • The system should have an intuitive and user-friendly interface for selecting destinations and inputting payment information. 2. **Security**: • The credit card information input by the user should be securely handled and encrypted to prevent unauthorized access or data breaches. 3. **Performance**: • The system should process transactions swiftly, with minimal delay in credit card validation and ticket issuance. 4. **Reliability**: • The system should operate reliably, minimizing errors or system failures during credit card validation or ticket issuance.

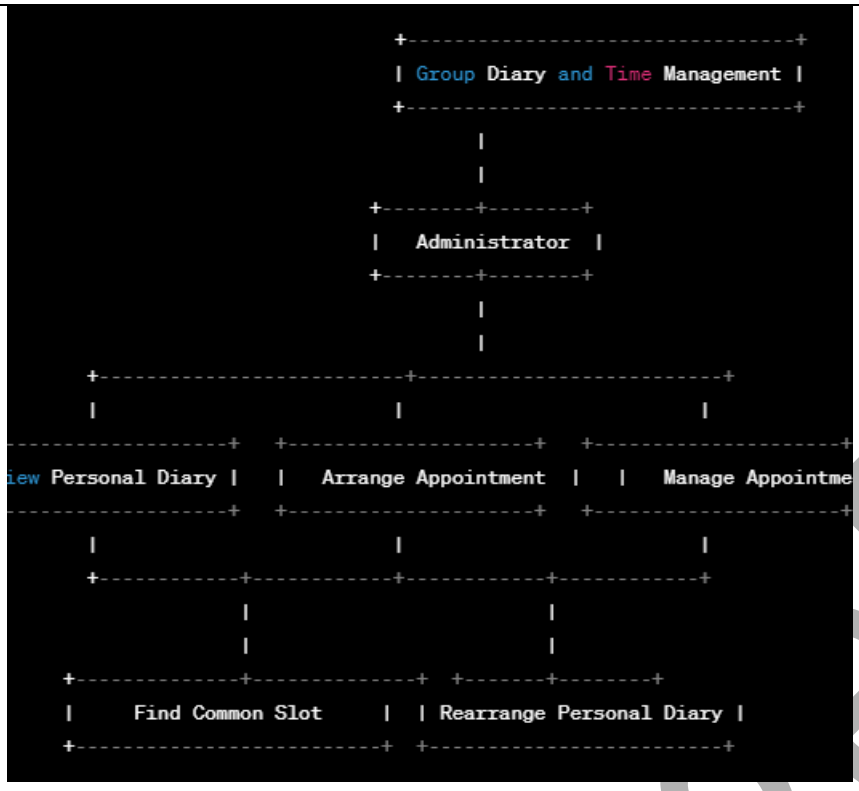
	5. **Scalability**: • The system should be capable of handling multiple users simultaneously, especially during peak times, without a decrease in performance. 6. **Availability**: • The system should be available for use during operational hours, with minimal downtime for maintenance or upgrades. 7. **Compliance**: • The system should comply with relevant industry standards and regulations regarding credit card transactions and user data protection. These requirements, both functional and non-functional, outline the expected functionalities and characteristics that the automated ticket-issuing system should possess to meet user needs effectively while ensuring security, usability, and performance.
Q2 C	What is requirements elicitation? explain requirement elicitation and analysis process in brief
ANS:	Requirements elicitation is the process of gathering, identifying, and defining the needs and expectations of stakeholders for a software system or product. It involves interacting with stakeholders to understand their perspectives, goals, and requirements that the system should fulfill. The goal is to capture comprehensive and accurate requirements that serve as the foundation for the software development process. **Requirement Elicitation and Analysis Process:** 1. **Stakeholder Identification**: • Identify and engage with stakeholders, including end-users, customers, sponsors, and subject matter experts (SMEs), to understand their roles and perspectives. 2. **Requirement Gathering Techniques**: • Use various techniques like interviews, workshops, surveys, observations, and brainstorming sessions to extract requirements from stakeholders. 3. **Documentation and Analysis**: • Document gathered requirements in various forms like use cases, user stories, or requirement specifications. • Analyze and categorize requirements into functional (what the system should do) and non-functional (qualities and constraints) aspects. 4. **Requirement Prioritization**: • Prioritize requirements based on their importance, urgency, and feasibility. This helps in focusing on critical aspects and managing constraints effectively. 5. **Validation and Verification**: • Validate requirements with stakeholders to ensure accuracy, completeness, and alignment with their needs. • Verify requirements for consistency, feasibility, and conformity to standards and constraints. 6. **Traceability and Management**: • Establish traceability to link requirements to their origins and other related artifacts, enabling better management and change tracking. • Maintain and manage requirements throughout the software development lifecycle, accommodating changes and updates as needed. 7. **Communication and Collaboration**: • Maintain open communication channels among stakeholders, ensuring clarity and consensus regarding requirements. • Collaborate with various teams (development, testing, management) to ensure a shared understanding of requirements. 8. **Feedback and Iteration**: • Seek feedback continuously from stakeholders and incorporate necessary changes or refinements into requirements. • Iteratively refine and revisit requirements as the project progresses or as new information becomes available.

	The process of requirement elicitation and analysis is iterative and collaborative, involving continuous interactions with stakeholders to ensure that the identified requirements accurately reflect their needs and expectations. It lays the groundwork for successful software development by providing a clear understanding of what the software system should accomplish.
Q3 A	What is system model? Explain types of system models in brief
ANS:	A system model is a simplified representation or abstraction of a real-world system that helps in understanding, visualizing, and analyzing its components, interactions, and behaviors. It provides a structured way to conceptualize complex systems, aiding in design, development, and decision-making processes. **Types of System Models:** 1. **Physical Models**: • **Description**: Represents the physical structure and attributes of a system using physical objects or replicas. • **Example**: Scale models of buildings, prototypes of machinery. 2. **Mathematical Models**: • **Description**: Represents system behavior using mathematical equations, formulas, or algorithms. • **Example**: Differential equations in physics, statistical models in finance. 3. **Simulation Models**: • **Description**: Uses computer simulations to mimic the behavior of a system under different conditions. • **Example**: Flight simulators, weather prediction models. 4. **Conceptual Models**: • **Description**: Provides a high-level abstract representation of a system, focusing on concepts and relationships. • **Example**: Entity-relationship diagrams in database design, UML diagrams in software engineering. 5. **Hierarchical Models**: • **Description**: Organizes a system into hierarchical levels or layers, depicting relationships and dependencies. • **Example**: OSI (Open Systems Interconnection) model in networking, systems hierarchy in organizational structures. 6. **Process Models**: • **Description**: Represents the flow of processes or activities within a system. • **Example**: Flowcharts, business process models. 7. **State Models**: • **Description**: Represents the different states a system can be in and transitions between these states. • **Example**: Finite State Machines (FSM), state diagrams in software design. 8. **Control Models**: • **Description**: Focuses on the control mechanisms and processes within a system. • **Example**: Control flow diagrams, PID (Proportional-Integral-Derivative) controllers in engineering. **Importance of System Models:** • **Understanding and Analysis**: Helps in comprehending complex systems, their components, and interactions. • **Design and Development**: Aids in planning, designing, and developing systems by providing a structured representation. • **Communication**: Facilitates communication and collaboration among stakeholders by providing a visual or abstract representation of the system. Different types of system models cater to various aspects of a system, and often, a combination of these models is used to provide a comprehensive view and understanding of the system under consideration.

Q3 B	What are Architectural views? Explain 4 + 1 view model of software architecture.
ANS:	Architectural views in software engineering represent different perspectives or aspects of a system's architecture, focusing on specific concerns or stakeholders. They provide a structured way to understand and document the architecture of a complex system by breaking it down into multiple viewpoints or views. Each view emphasizes certain aspects of the system's structure, behavior, or functionality. **4 + 1 View Model of Software Architecture:** The 4 + 1 view model, introduced by Philippe Kruchten, provides five distinct views to describe software architecture comprehensively: 1. **Logical View**: • **Focus**: Describes the functionality and behavior of the system from a logical or functional perspective. • **Elements**: Shows components/modules, their relationships, and interactions without considering the physical structure. 2. **Process View**: • **Focus**: Emphasizes the dynamic aspects of the system, including concurrency, processes, and interactions between components during runtime. • **Elements**: Depicts tasks, processes, threads, and how they communicate or synchronize. 3. **Development View**: • **Focus**: Illustrates the system's architecture from a developer's perspective, focusing on the organization of the code, modules, and development environment. • **Elements**: Shows software modules, development tools, and their relationships. 4. **Physical View**: • **Focus**: Represents the system's physical or deployment aspects, such as hardware, networks, and distribution of components across different nodes. • **Elements**: Depicts hardware infrastructure, network topology, and allocation of software components to hardware. 5. **Scenarios (Use Cases) View**: • **Focus**: Presents the system from the viewpoint of user interactions or use cases, providing scenarios to understand how the system behaves in different situations. • **Elements**: Uses scenarios, use cases, or user stories to illustrate system behavior in specific contexts. **Importance of 4 + 1 View Model:** • **Comprehensiveness**: Provides multiple perspectives to understand different aspects of the architecture, catering to various stakeholders. • **Clarity and Communication**: Facilitates communication among stakeholders by offering distinct views that focus on specific concerns. • **Consistency**: Helps in maintaining consistency across different views, ensuring coherence in the overall architecture. By using this model, software architects can effectively communicate, analyze, and document the various facets of a software system's architecture, accommodating the diverse needs and concerns of stakeholders involved in the development process.
Q3 C	What are the Architectural patterns? Explain Model view controller (MVC) pattern
ANS:	Architectural patterns are reusable solutions to common architectural problems in software design. They provide proven structures for organizing code, facilitating scalability, maintainability, and improving overall system design. One such pattern is the Model-View-Controller (MVC) pattern. **Model-View-Controller (MVC) Pattern:** **Purpose**: MVC separates an application into three interconnected components to separate internal representations of information from the way information is presented to and accepted from the user. 1. **Model**: • Represents the application's data and business logic.

	• Manages the data, responds to queries, and interacts with the database or external data sources. • Independent of the user interface and views. 2. **View**: • Represents the visual presentation of the data. • Presents the model's data to the user in a specific format. • Receives input from users and sends commands to the controller. 3. **Controller**: • Acts as an intermediary between the model and the view. • Receives user input from the view, processes it (often modifying the model), and updates the view. • Controls the flow of the application and manages user interactions. **Workflow in MVC:** 1. **User Interaction**: • User interacts with the View component (e.g., clicks a button). 2. **View Notifies Controller**: • View notifies the Controller about the user's action. 3. **Controller Processes Input**: • Controller processes the input, interacts with the Model (updates data), and decides which View to render. 4. **Model Updates View**: • Model updates, and the Controller selects an appropriate View to represent the modified Model data. 5. **Updated View Displayed**: • Updated View is displayed to the user. **Advantages of MVC:** • **Separation of Concerns**: Clear separation between data (Model), presentation logic (View), and user interaction logic (Controller). • **Modularity and Reusability**: Components can be modified or replaced without affecting others, promoting code reusability. • **Scalability and Maintainability**: Easier to maintain and scale due to the clear structure and organization of components. **Use Cases of MVC:** • **Web Applications**: Often used in web development frameworks like Ruby on Rails, Django, and ASP.NET MVC. • **Desktop and Mobile Applications**: Can be applied to various GUI-based applications for better organization and maintainability. The MVC pattern promotes a clean separation of concerns, making code more modular, maintainable, and scalable, which is why it's widely adopted in software development across different platforms and domains.
Q4 A	What is software reuse? Explain software reuse levels.
ANS:	Software reuse refers to the practice of utilizing existing software assets, components, or artifacts in the development of new software systems or products. It aims to save time, effort, and resources by leveraging pre-existing solutions instead of starting development from scratch. **Levels of Software Reuse:** 1. **Code-Level Reuse**: • **Description**: Involves reusing specific code segments, functions, or modules within a new software project. • **Examples**: Reusing libraries, functions, or classes from previous projects or third-party libraries. 2. **Component-Level Reuse**: • **Description**: Reusing larger, self-contained software components or modules that encapsulate certain functionalities.

	• **Examples**: Reusing components like UI widgets, data access modules, or middleware components. 3. **Design-Level Reuse**: • **Description**: Reusing design patterns, architectures, or high-level design concepts across different projects. • **Examples**: Implementing known architectural patterns (like MVC), design templates, or standardized design principles. 4. **System-Level Reuse**: • **Description**: Reusing entire systems or subsystems in new projects, often involving complex systems or platforms. • **Examples**: Reusing entire software systems, platforms, or frameworks for similar projects or domains. **Advantages of Software Reuse:** 1. **Efficiency and Productivity**: • Reduces development time and effort by leveraging existing solutions, leading to faster time-to-market. 2. **Consistency and Quality**: • Promotes consistency and maintains quality across projects by reusing proven and tested components or designs. 3. **Cost Savings**: • Reduces development costs as it eliminates the need to reinvent the wheel for every new project. 4. **Risk Mitigation**: • Reduces the risk of errors or defects as reused components have often been tested and debugged in previous implementations. **Challenges of Software Reuse:** 1. **Finding and Understanding Reusable Components**: • Locating suitable and compatible components for reuse can be challenging. 2. **Maintenance and Customization**: • Reused components might require maintenance or customization to fit specific project requirements. 3. **Compatibility and Integration**: • Ensuring seamless integration and compatibility between reused components and new systems. 4. **Documentation and Knowledge Sharing**: • Adequate documentation and knowledge sharing are crucial for effective reuse, but these may be lacking or outdated. Efficient software reuse involves careful planning, documentation, and managing a repository of reusable assets. While it offers numerous benefits, successful reuse requires considering the trade-offs and ensuring that reused components are appropriate and well-suited for the new context.
Q4 B	A group diary and time management system are intended to support the timetabling of meetings and appointments across a group of co-workers. When an appointment is to be made that involves a number of people, the system finds a common slot in each of their diaries and arranges the appointment for that time. If no common slots are available, it interacts with the user to rearrange his or her personal diary to make room for the appointment. For above case study Draw the use-case diagram
ANS:	Based on the description provided for the group diary and time management system, here's a use-case diagram representing the functionalities and interactions of the system:



Use Cases:

1. **Administrator**:
   - Manages system functionalities and configurations.
2. **View Personal Diary**:
   - Allows users to view their personal schedule or diary.
3. **Arrange Appointment**:
   - Finds a common slot in each user's diary for arranging appointments involving multiple people.
4. **Manage Appointments**:
   - Handles various operations related to appointments, such as creation, modification, or cancellation.
5. **Find Common Slot**:
   - Searches for a suitable time slot that accommodates all involved participants for a group appointment.
6. **Rearrange Personal Diary**:
   - Interacts with the user to rearrange their personal schedule if no common slots are available for an appointment.

This diagram represents the main functionalities of the group diary and time management system, showcasing how users (co-workers) interact with the system to manage their schedules and arrange appointments efficiently.

Q4 C	For the Case study in last question Draw the sequence diagram for use case book an appointment (free slot)
ANS:	Creating a sequence diagram for the "Book an appointment (free slot)" use case involves illustrating the interactions and messages exchanged between different objects or components involved in this process. Given the context of the group diary and time management system, here's a simplified sequence diagram for this use case:

Sequence of Events:

1. **User Interaction**:
   - The User triggers the process by requesting to find a free slot for an appointment.
2. **Group Diary System**:
   - Upon receiving the request, the Group Diary System checks available free slots by invoking **checkFreeSlots()**.
3. **Response with Available Free Slots**:
   - The system responds with a list of available free slots to the User.
4. **User Selection**:
   - The User selects a specific time slot (**chooseTimeSlot(slot)**).
5. **Booking Process**:
   - The User confirms the selection, triggering the **bookAppointment()** method.
6. **Confirmation**:
   - The Group Diary System confirms the booking by invoking **confirmBooking()** and sends a confirmation message to the User.

This sequence diagram outlines the flow of messages between the User and the Group Diary System during the "Book an appointment (free slot)" use case, depicting the interactions involved in finding and booking an available slot for an appointment within the system.

Q5 A Compare software inspection and testing

ANS:

Aspect	Software Inspection	Software Testing
Purpose	Identifying defects through manual examination	Validating software behavior through execution
Methodology	Formal review process, peer examination	Executing software with specific test cases

Timing	Early stages of development	Throughout the development lifecycle
Participants	Group of reviewers, developers, experts	Dedicated testers, developers, automated tools
Goal	Detecting issues, improving quality early	Identifying defects, validating functionality
Focus	Finding defects, adhering to standards	Evaluating behavior, meeting requirements

Differences:

- **Purpose**: Inspection aims at identifying issues through examination, while testing validates behavior through execution.
- **Methodology**: Inspection involves a formal review process, whereas testing involves executing software with test cases.
- **Timing**: Inspections are done early; testing occurs throughout development.
- **Participants**: Inspection involves peer review, while testing involves dedicated testers and tools.
- **Goal**: Inspections focus on defect identification; testing validates functionality and compliance with requirements.

Both inspection and testing play crucial roles in ensuring software quality, but their approaches, timing, and goals differ, allowing them to complement each other in ensuring robust and high-quality software products.

Q5 B What is Test-driven development (TDD)? Explain TDD process activities

ANS: Test-driven development (TDD) is a software development approach where tests are written before the actual code is implemented. It follows a cycle of writing tests, writing code to pass those tests, and then refactoring the code to improve its structure without changing its functionality.

TDD Process Activities:

1. **Write Test**:
   - **Activity**: Developers start by writing a failing test that defines the desired functionality or behavior.
   - **Purpose**: Clarify the expected behavior or requirements before writing the code.
2. **Run Test (Fail)**:
   - **Activity**: Run the test and observe it fail (as there's no code yet to pass the test).
   - **Purpose**: Validate that the test is working and accurately reflects the functionality needed.
3. **Write Code to Pass Test**:
   - **Activity**: Implement the minimal code required to pass the failing test.
   - **Purpose**: Fulfill the requirements and make the failing test pass without adding unnecessary functionality.
4. **Run Test (Pass)**:
   - **Activity**: Run the test again and ensure it passes with the implemented code.
   - **Purpose**: Confirm that the code satisfies the test and delivers the expected functionality.
5. **Refactor Code**:
   - **Activity**: Optimize and improve the code's structure without altering its behavior.
   - **Purpose**: Enhance code quality, readability, and maintainability without changing its functionality.
6. **Run Tests (Validate)**:
   - **Activity**: Run all tests, including the newly passed one, to ensure the changes haven't broken existing functionalities.
   - **Purpose**: Validate that all existing functionalities remain intact after the code modifications.
7. **Repeat Cycle**:
   - **Activity**: Repeat the cycle for each new functionality or feature.
   - **Purpose**: Incrementally build the software while ensuring new code doesn't break existing functionalities.

Benefits of TDD:

1. **Improved Code Quality**:

	• Ensures code is designed to fulfill specific requirements and is thoroughly tested. 2. **Faster Debugging**: • Detects and fixes defects earlier in the development process, saving time in the long run. 3. **Incremental Development**: • Allows software to be built incrementally with small, manageable steps. 4. **Clearer Requirements**: • Provides a clear set of tests that document the expected behavior of the code. TDD emphasizes writing reliable, maintainable code by focusing on delivering specific functionalities through iterative testing and development, promoting code quality and reducing the chances of defects in the software.
Q5 C	Explain principal dependability properties in brief
ANS:	Dependability properties in software engineering ensure that a system operates reliably, securely, and consistently. Here are the principal dependability properties: 1. **Reliability**: • **Definition**: Refers to the ability of a system to perform its intended functions without failure over a specified period. • **Key Aspect**: Focuses on consistent and error-free operation, minimizing the likelihood of system failures. 2. **Availability**: • **Definition**: Represents the proportion of time a system is functional and operational when needed. • **Key Aspect**: Ensures that the system is available and accessible to users whenever required, minimizing downtime. 3. **Safety**: • **Definition**: Refers to the system's ability to operate without causing harm to users, environment, or other systems. • **Key Aspect**: Focuses on identifying and mitigating potential risks and hazards to prevent accidents or harm. 4. **Security**: • **Definition**: Ensures protection against unauthorized access, data breaches, and malicious attacks. • **Key Aspect**: Involves measures to safeguard data, resources, and functionalities from threats or vulnerabilities. 5. **Maintainability**: • **Definition**: Refers to the ease and effectiveness of making changes, updates, and repairs to a system. • **Key Aspect**: Includes factors like modularity, documentation, and simplicity, allowing for efficient system maintenance and evolution. 6. **Integrity**: • **Definition**: Ensures the accuracy and consistency of data throughout its lifecycle. • **Key Aspect**: Focuses on preventing unauthorized or accidental alterations to data, maintaining its trustworthiness. 7. **Resilience**: • **Definition**: Represents the system's ability to recover and adapt to adverse conditions or failures. • **Key Aspect**: Involves mechanisms to handle failures gracefully, ensuring continued operation or quick recovery. These properties collectively contribute to the overall dependability of a system, ensuring it operates reliably, securely, and consistently under varying conditions and demands. Achieving and maintaining these properties is crucial for building trustworthy and resilient software systems.