1

(a) Function of Each Task in Requirement Engineering

1. **Elicitation**: Gathering requirements from stakeholders, users, and other sources to understand their needs and expectations.
2. **Analysis**: Analyzing and refining the elicited requirements to ensure they are clear, complete, and consistent.
3. **Specification**: Documenting the requirements in a clear and structured manner, often using techniques like use cases, user stories, and requirements documents.
4. **Validation**: Reviewing and validating the requirements with stakeholders to ensure they accurately capture their needs and expectations.
5. **Verification**: Ensuring that the implemented system meets the specified requirements through testing, inspection, and other verification techniques.
6. **Management**: Managing changes to the requirements throughout the project lifecycle, ensuring traceability, and maintaining version control.

(b) Main Actors and Use Cases for ATM Cash Withdrawal

Main Actors:

1. Customer
2. ATM Machine

Use Cases:

1. **Insert Card**: The customer inserts their ATM card into the card reader.
2. **Enter PIN**: The customer enters their Personal Identification Number (PIN) using the keypad.
3. **Enter Amount**: The customer enters the amount of cash they wish to withdraw using the keypad.

4. **Take Card**: The customer retrieves their ATM card from the card reader.
5. **Take Money**: The customer takes the dispensed cash from the cash dispenser.

(c) Functional and Non-Functional Requirements for Auction Web-Market

Functional Requirements:

1. Sellers publish offerings with product descriptions and sales conditions.
2. Buyers can search with a hierarchy of categories or with free-text keywords.
3. Buyers can bid on offerings.
4. Sellers can sell to the highest bid or at a fixed price.
5. Sellers can be business sellers or private persons.
6. Buyers can rate sellers.

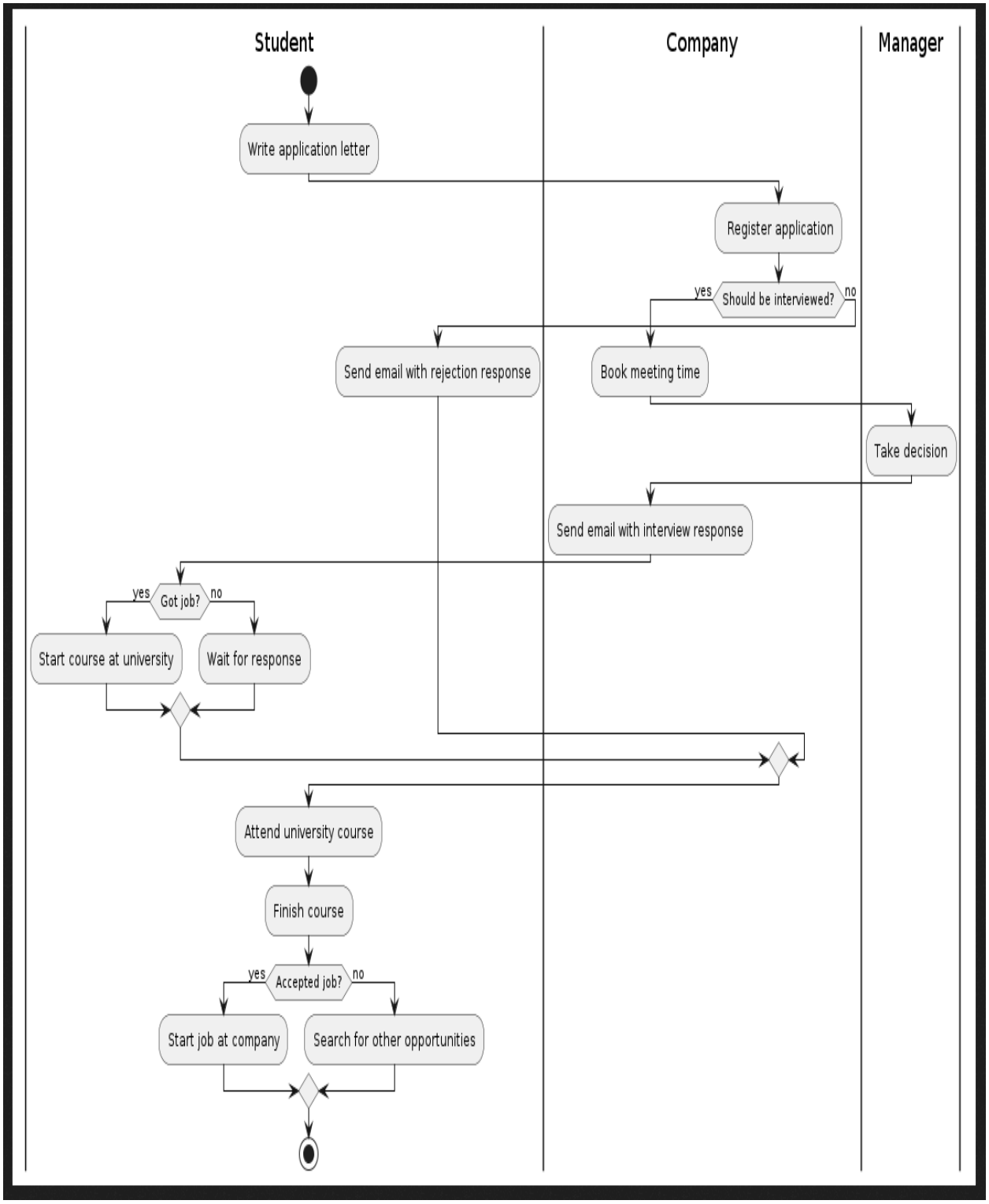
Non-Functional Requirements:

1. **Performance**: The system should respond to search queries and bidding actions within a reasonable time frame, even under heavy load.
2. **Usability**: The user interface should be intuitive and easy to navigate, allowing both buyers and sellers to interact with the system effortlessly.
3. **Security**: The system should ensure the confidentiality of user information, secure transactions, and protect against unauthorized access.
4. **Scalability**: The system should be able to handle a growing number of users, offerings, and transactions without degradation in performance.
5. **Reliability**: The system should be available and operational at all times, with minimal downtime for maintenance or upgrades.

6. **Compatibility**: The system should be compatible with various devices and browsers to ensure access for a wide range of users.

2

b)



4

(a) Various Types of Coupling and Problems of High Coupling

Types of Coupling:

1. **Content Coupling**: Occurs when one module accesses or modifies the internal data of another module.
2. **Common Coupling**: Happens when multiple modules share the same global data.
3. **Control Coupling**: Occurs when one module controls the behavior of another module through parameters, flags, or function calls.
4. **Stamp Coupling**: Occurs when modules share a complex data structure, such as an array or structure.
5. **Data Coupling**: Happens when modules communicate by passing data between them using parameters or global data.
6. **Message Coupling**: Occurs when modules communicate by sending messages, such as method calls or function invocations.

Problems of High Coupling:

1. **Dependency**: High coupling creates strong dependencies between modules, making it difficult to modify or replace one module without affecting others.
2. **Complexity**: Coupled modules are more complex and harder to understand, leading to increased development and maintenance effort.
3. **Rigidity**: Changes in one module may require modifications in multiple other modules, leading to a rigid and inflexible system.
4. **Testing Difficulty**: High coupling makes testing more challenging as changes in one module can have unexpected impacts on other modules.
5. **Reusability**: Coupled modules are less reusable as they are tightly integrated with other modules, limiting their applicability in different contexts.

(b) Benefits of High Cohesion and Low Coupling

High Cohesion:

1. **Better Understandability**: High cohesion makes modules more focused and easier to understand as each module has a clear and specific purpose.
2. **Improved Maintainability**: Modules with high cohesion are easier to maintain and modify as changes are localized within the module, reducing the risk of unintended side effects.
3. **Enhanced Reusability**: High cohesion promotes modular design, allowing modules to be reused in different parts of the system or in other projects.
4. **Reduced Complexity**: High cohesion reduces complexity by organizing code into logical and manageable units, making it easier to manage and evolve over time.

Low Coupling:

1. **Improved Modifiability**: Low coupling reduces dependencies between modules, making it easier to modify or replace one module without affecting others.
2. **Enhanced Testability**: Low coupling simplifies testing as modules can be tested independently without needing to consider the behavior of other modules.
3. **Increased Flexibility**: Low coupling allows modules to be developed, tested, and deployed independently, providing greater flexibility in system design and evolution.
4. **Better Scalability**: Low coupling facilitates modular development, allowing new modules to be added or existing modules to be modified without disrupting the overall system architecture.

(c) Benefits of Describing System Architecture Using Multiple Views

Describing system architecture using multiple views offers several benefits:

1. **Comprehensiveness**: Different stakeholders have different concerns and perspectives on the system. Multiple views allow architects to address these diverse concerns comprehensively, ensuring that all aspects of the system are adequately represented.
2. **Clarity**: Each view focuses on specific aspects of the system, providing clarity and simplifying the understanding of complex architectures for different stakeholders.
3. **Consistency**: By defining multiple views of the system, architects can ensure consistency across different architectural artifacts, such as diagrams, documents, and models, reducing ambiguity and confusion.
4. **Communication**: Multiple views facilitate effective communication between stakeholders by providing tailored representations of the system that are relevant to their roles and concerns.
5. **Analysis**: Different views enable different types of analysis, such as performance analysis, security analysis, and usability analysis, allowing architects to evaluate the system from various perspectives and make informed decisions.
6. **Evolution**: As the system evolves over time, different views provide a means to capture and communicate changes in the architecture, ensuring that stakeholders remain informed and aligned with the evolving system requirements and goals.

5

(a) Key Elements of a Test Case

1. **Test Case ID**: A unique identifier for the test case, which helps in tracking and referencing.
2. **Test Description**: A brief description of the functionality or scenario being tested.
3. **Test Inputs**: The input data or conditions required to execute the test case, including any preconditions.
4. **Expected Results**: The expected outcome or behavior of the system under test when the test case is executed.
5. **Actual Results**: The observed outcome or behavior of the system when the test case is executed, which is compared with the expected results to determine the test outcome.

(b) Black Box Test Suite for Palindrome Function

1. **Test Case 1**: Input a single character palindrome.
    - Input: 'a'
    - Expected Output: True
2. **Test Case 2**: Input a single character non-palindrome.
    - Input: 'b'
    - Expected Output: False
3. **Test Case 3**: Input an empty string.
    - Input: ''
    - Expected Output: True
4. **Test Case 4**: Input a palindrome string.
    - Input: 'radar'
    - Expected Output: True
5. **Test Case 5**: Input a non-palindrome string.
    - Input: 'hello'
    - Expected Output: False

(c) Difference Between White-Box and Black-Box Testing

White-Box Testing:

- Testing technique that examines the internal structure or workings of the software being tested.
- Test cases are derived from an understanding of the code, including branches, loops, and internal logic.
- Examples include statement coverage, branch coverage, and path coverage.
- Testers require access to the source code or knowledge of the internal implementation details.

Black-Box Testing:

- Testing technique that focuses on the external behavior or functionality of the software being tested.
- Test cases are derived from the software requirements and specifications.
- Examples include equivalence partitioning, boundary value analysis, and use case testing.
- Testers do not need access to the source code or knowledge of the internal implementation details.

6)

a)
Verification and validation (V&V) activities take place throughout the software development lifecycle (SDLC), but their frequency and intensity vary depending on the phase of the lifecycle. Here's how verification and validation activities typically align with different phases of the SDLC:

1. **Requirements Phase**:
    - **Verification**: Ensuring that the requirements documentation accurately captures the needs and expectations of stakeholders. Verification activities in this phase involve reviews, walkthroughs, and inspections of the requirements documents. Typically, the quality assurance (QA) team or requirements analysts are in charge of verification during this phase.
    - **Validation**: Validating the requirements with stakeholders to ensure they meet their intended purpose. This often involves eliciting feedback from stakeholders through meetings, interviews, and prototypes. Business analysts or product owners are typically responsible for validation activities in this phase.
2. **Design Phase**:
    - **Verification**: Ensuring that the design documents, such as architecture diagrams, system design specifications, and interface specifications, meet the specified requirements. Verification activities involve reviews and inspections of the design documents to identify design flaws and inconsistencies. Designers, architects, and system analysts are typically in charge of verification during this phase.
    - **Validation**: Validating the design with stakeholders to ensure it aligns with their expectations and requirements. This may involve presenting prototypes or mockups to stakeholders for feedback and approval. Designers and architects collaborate

with stakeholders to validate the design decisions made during this phase.

3. **Implementation Phase**:
    - **Verification**: Ensuring that the implemented software meets the design specifications and coding standards. Verification activities involve code reviews, unit testing, and static analysis to identify defects and coding errors. Developers and QA engineers collaborate to perform verification activities during this phase.
    - **Validation**: Validating the implemented software with stakeholders to ensure it meets their needs and expectations. This may involve conducting acceptance testing, user acceptance testing (UAT), and system integration testing (SIT) to validate the functionality and performance of the software. QA engineers and end-users collaborate to perform validation activities during this phase.
4. **Testing Phase**:
    - **Verification**: Ensuring that the software meets the specified requirements and quality standards. Verification activities involve executing test cases, analyzing test results, and verifying that the software behaves as expected. QA engineers are primarily responsible for verification activities during the testing phase.
    - **Validation**: Validating the software against the user's needs and expectations. This may involve conducting exploratory testing, usability testing, and user feedback sessions to ensure the software meets user requirements. QA engineers and end-users collaborate to perform validation activities during the testing phase.

Overall, verification and validation activities are iterative and continuous throughout the SDLC, with different team members taking on responsibilities based on their expertise and role in the

c)

Risk Management Plan for Software Development Project

1. Technology Risk:

- **Risk**: Integration Complexity: The integration of third-party APIs or legacy systems may introduce technical challenges and compatibility issues.
- **Acceptance Strategy 1 (Avoidance)**: To avoid this risk, the project team will conduct thorough research and feasibility studies before selecting third-party APIs or integrating with legacy systems. Alternative solutions or workarounds will be explored to minimize dependencies on complex integrations.
- **Acceptance Strategy 2 (Mitigation)**: Mitigation strategies will include conducting comprehensive testing and validation of integration points early in the development process. Prototyping and proof-of-concept exercises will be utilized to identify and address integration complexities proactively.

2. Business Risk:

- **Risk**: Market Volatility: Changes in market conditions, customer preferences, or regulatory requirements may impact the demand for the software product.
- **Acceptance Strategy 1 (Transfer)**: The project team will consider outsourcing certain business functions or services to third-party vendors or partners. This will help distribute the risk associated with market volatility and ensure continuity of operations.
- **Acceptance Strategy 2 (Mitigation)**: Mitigation strategies will include diversifying the product portfolio to target multiple market segments or industries. Continuous market research and customer feedback loops will be established to identify emerging trends and adapt the product

  roadmap accordingly.

3. Project Risk:

- **Risk**: Resource Constraints: Limited availability of skilled resources or unexpected resource turnover may impact project timelines and deliverables.
- **Acceptance Strategy 1 (Acceptance)**: The project team acknowledges the risk of resource constraints and accepts that there may be occasional delays or challenges due to resource limitations. Contingency plans will be developed to prioritize critical tasks and reallocate resources as needed.
- **Acceptance Strategy 2 (Transfer)**: To transfer the risk of resource constraints, the project team will explore outsourcing certain project tasks or activities to external vendors or contractors. This will help augment internal resources and ensure continuity of project progress in case of resource shortages.

7

(a) Definitions and Explanation:

1. **Role**:
    - Definition: A role refers to a set of responsibilities, activities, and behaviors assigned to an individual or a group within a project or organization.
    - Explanation: Roles define the functions that individuals or teams perform within a project. Examples of roles include project manager, developer, tester, business analyst, etc.

2. **Task**:
    - Definition: A task is a specific activity or action that needs to be completed to achieve a project objective or deliverable.
    - Explanation: Tasks are smaller units of work that contribute to the overall project goals. They are usually identified during

project planning and are assigned to individuals or teams based on their skills and expertise.

3. **Work Package**:
    - Definition: A work package is a collection of related tasks or activities that are grouped together and assigned to a single individual or team for execution.
    - Explanation: Work packages break down the project scope into manageable chunks of work. They contain a set of tasks that can be completed within a defined timeframe and budget. Work packages are typically assigned to specific roles or resources within the project team.
4. **Communication Strategies**:
    - **Face-to-Face Communication**: Involves direct interaction between individuals or teams, such as meetings, workshops, or brainstorming sessions. It promotes immediate feedback and fosters collaboration and relationship building.
    - **Written Communication**: Involves exchanging information through written documents, emails, reports, or memos. It provides a permanent record of communication and allows for detailed explanations and documentation.
    - **Virtual Communication**: Involves communication through digital platforms, such as video conferences, instant messaging, or collaboration tools. It enables remote collaboration and allows team members to communicate across geographical locations and time zones.
    - **Formal Communication**: Involves structured and official communication channels, such as project status reports, progress meetings, or performance reviews. It ensures consistency and clarity in communication and helps in aligning stakeholders with project objectives.
    - **Informal Communication**: Involves casual and spontaneous communication between team members, such as water cooler conversations or social gatherings. It helps in building rapport and camaraderie among team members and promotes a positive work environment.

(b) Basic Principle for Project Scheduling and Importance of Critical Path and Slack Time:

Basic Principle for Project Scheduling:

- The basic principle of project scheduling is to sequence project activities in a logical order and allocate resources to ensure that project objectives are achieved within the specified time frame and budget.
- Project scheduling involves identifying tasks, estimating their durations, determining their dependencies, and creating a timeline or schedule for their execution.

Importance of Critical Path and Slack Time:

- **Critical Path**: The critical path represents the longest sequence of dependent tasks in a project, determining the shortest possible duration for completing the project. Tasks on the critical path have zero slack time, meaning any delay in these tasks will directly impact the project's overall duration. Identifying the critical path helps project managers prioritize tasks and allocate resources effectively to ensure timely project completion.
- **Slack Time**: Slack time, also known as float or buffer, represents the amount of time by which a task can be delayed without delaying the project's overall completion time. Tasks with slack time can be delayed without affecting the project's critical path. Slack time provides flexibility in project scheduling and allows project managers to manage uncertainties and changes in project timelines effectively. It helps in optimizing resource allocation and mitigating risks associated with task dependencies and uncertainties in project execution.

(c) Effort Calculation:

Given:

- Unadjusted Function Points (UFP) = 18

- Value Added Factor (VAF) = 0.87
- Performance Factor (PF) = 2

Effort (in person-days) = UFP * VAF * PF = 18 * 0.87 * 2 = 31.68 person-days

Therefore, the effort required for the project is approximately 31.68 person-days.