26

1

(a) Combined View of Software Development Lifecycle

The combined view of the software development lifecycle integrates both the activity-oriented and entity-oriented perspectives. This approach maps the activities involved in software development to the deliverables produced at each stage, ensuring clarity and coherence in the development process.

Combined View

1. **Requirements Engineering**
    - **Activities**:
        - Requirements elicitation
        - Requirements analysis
        - Requirements documentation
        - Requirements validation
    - **Deliverables**:
        - Requirements specification document
        - Use case diagrams
        - User stories
2. **System and Software Design**
    - **Activities**:
        - High-level system design
        - Detailed software design
        - Architecture design
        - Database design
    - **Deliverables**:
        - System architecture diagrams
        - Detailed design documents
        - Data models and schemas
        - Prototypes and wireframes
3. **Implementation (Coding)**
    - **Activities**:

        - Coding
        - Code reviews
        - Unit testing
    - **Deliverables**:
        - Source code
        - Unit test cases and results
        - Code review reports
4. **Integration and Testing**
    - **Activities**:
        - Integration of modules
        - System testing
        - Performance testing
        - Security testing
    - **Deliverables**:
        - Integrated system
        - Test plans and test cases
        - Test reports and logs
        - Defect reports
5. **Deployment**
    - **Activities**:
        - Deployment planning
        - Deployment execution
        - User training
    - **Deliverables**:
        - Deployment guides
        - User manuals
        - Training materials
6. **Maintenance**
    - **Activities**:
        - Bug fixing
        - System enhancements
        - Performance tuning
    - **Deliverables**:
        - Updated source code
        - Patch releases

        - Updated documentation

(b) Fundamental Distinctions Between the Spiral Model and the Waterfall Model

1. **Approach**:
    - **Waterfall Model**: Sequential approach where each phase (requirements, design, implementation, testing, deployment, and maintenance) is completed before moving to the next. Once a phase is completed, there is little scope for revisiting it.
    - **Spiral Model**: Iterative approach that combines elements of both design and prototyping in stages. It emphasizes risk assessment and reduction at regular intervals through its cycles.
2. **Flexibility**:
    - **Waterfall Model**: Inflexible due to its linear progression. Changes are difficult and costly once a phase is completed.
    - **Spiral Model**: Highly flexible and adaptable to changes. It allows for iterative refinement of requirements and design based on feedback and risk assessment.
3. **Risk Management**:
    - **Waterfall Model**: Minimal focus on risk management. Risks are often identified late in the process.
    - **Spiral Model**: Explicitly incorporates risk analysis and management at each iteration, enabling early identification and mitigation of risks.
4. **Customer Involvement**:
    - **Waterfall Model**: Limited customer involvement after the initial requirements phase until the testing phase.
    - **Spiral Model**: Continuous customer involvement throughout the development process, ensuring feedback is incorporated in each iteration.

(c) When is Evolutionary Model Preferred to Waterfall Model?

Evolutionary Model:

- The evolutionary model, which includes iterative and incremental models, is preferred when requirements are not well understood at the beginning or are expected to evolve over time. It allows for the software to be developed and delivered in increments, enabling continuous feedback and adaptation.

Preferred Situations:

1. **Unclear or Evolving Requirements**: When the project requirements are not well-defined initially and are likely to change based on user feedback or market conditions. The evolutionary model accommodates these changes more gracefully than the waterfall model.
2. **High-Risk Projects**: When the project involves high uncertainty or risk, such as new technology adoption or innovative solutions. The iterative nature of the evolutionary model allows for early identification and mitigation of risks.
3. **User Involvement and Feedback**: When continuous user involvement and feedback are critical to the success of the project. The evolutionary model allows for regular user feedback and incorporation of changes, ensuring the final product meets user needs.

Explanation:

- **Flexibility**: The evolutionary model's iterative approach allows for continuous refinement and improvement of the software based on feedback and changing requirements, making it more suitable for dynamic and uncertain environments.
- **Risk Management**: By delivering the software in increments, the evolutionary model helps identify and mitigate risks early, reducing the overall project risk.

- **Customer Satisfaction**: Regular user involvement and feedback ensure that the software evolves according to user expectations, leading to higher customer satisfaction.

2
a)

The statement "The requirement analysis document serves as a contract between development team and the customer" can be justified with the following reasoning:

1. **Clarity and Agreement on Scope**:
    - The requirement analysis document explicitly outlines what the system will do, specifying the functionalities and features that need to be developed. This ensures that both the customer and the development team have a shared understanding of the project scope.
    - By documenting requirements, both parties agree on what is to be included, thus minimizing the risk of scope creep where additional features are requested after the project has started.
2. **Basis for Project Planning**:
    - The document provides a detailed description of the requirements which serves as the foundation for project planning, including time estimates, resource allocation, and budgeting.
    - It allows the development team to create realistic schedules and milestones that the customer can review and approve, ensuring that both parties are aligned on the timeline.

3. **Benchmark for Deliverables**:
    - The requirement analysis document acts as a benchmark against which the development team's progress and the final deliverables can be measured.
    - It helps in verifying and validating that the developed system meets the specified requirements. This ensures that the customer receives exactly what was agreed upon, and the development team has clear criteria for completion.
4. **Mitigation of Misunderstandings and Disputes**:
    - Having a formal document reduces the possibility of misunderstandings about what the system should do, as everything is clearly stated and agreed upon.
    - In case of any disputes or disagreements during the project, the document can be referred to as the authoritative source of the agreed requirements, thus helping to resolve conflicts amicably.
5. **Legal and Formal Binding**:
    - The requirement analysis document can serve a legal function by acting as a formal agreement or contract. It can be used in legal settings to resolve disputes over the project scope and deliverables.
    - This formalization adds a layer of accountability for both the customer and the development team, as both are bound to the commitments made in the document.

6. **Facilitation of Communication**:
    - It ensures ongoing communication and collaboration between the customer and the development team. Regular reviews and updates to the document can keep both parties informed of any changes or new insights that may affect the project.
    - This continuous dialogue helps maintain alignment and address any issues promptly, reducing the risk of project failure.

In conclusion, the requirement analysis document serves as a contract by clearly defining the project scope, setting expectations, providing a basis

for planning and verification, mitigating misunderstandings, offering legal protection, and facilitating effective communication between the customer and the development team.

b)

Classifying and separating functional requirements and non-functional requirements in a requirements specification document is a good idea for the following reasons:

1. **Clear Understanding**:
    - **Functional Requirements**: These are the specific tasks or features the system must perform, like user login or data processing. Keeping these separate makes it easy to see exactly what the system needs to do.
    - **Non-Functional Requirements**: These describe how the system performs those tasks, such as speed, security, or usability. Separating them ensures that everyone understands the expected quality and performance standards of the system.
2. **Better Focus and Management**:
    - When functional and non-functional requirements are separated, it helps the development team focus on building the system's functions first and then ensuring it meets the necessary quality standards.
    - This separation also makes it easier to manage and test each type of requirement independently, ensuring nothing is overlooked.

c)

For the given description of an alarm clock system, the functional and non-functional requirements can be identified as follows:

Functional Requirements

1. **Time Display**:
    - The clock shows the current time of day.
    - Users can choose between 12-hour and 24-hour display formats.
2. **Time Setting**:
    - Users can set the hours and minutes fields individually using buttons.
3. **Alarm Setting**:
    - Users can set one or two alarms.
4. **Alarm Activation**:
    - When an alarm fires, it will sound a noise.
    - The user can turn off the alarm.
    - The user can choose to 'snooze' the alarm.
    - If the user does not respond, the alarm will automatically turn off after 2 minutes.
5. **Snooze Function**:
    - 'Snoozing' turns off the sound but will cause the alarm to fire again after a predefined delay.
    - Users can adjust the 'snoozing time'.

Non-Functional Requirements

1. **Performance**:
    - The alarm should sound immediately at the set time.
    - The clock should update the time display in real-time without lag.
2. **Usability**:
    - The buttons for setting the time and alarms should be easy to use.
    - Switching between 12-hour and 24-hour formats should be intuitive and quick.
3. **Reliability**:
    - The alarm clock should maintain accurate time over extended periods.
    - The alarm should reliably sound at the set times.

4. **Sound Quality**:
    - The alarm sound should be loud enough to wake the user but not overly harsh.
5. **Durability**:
    - The buttons should withstand repeated use over time without malfunctioning.

By distinguishing these functional and non-functional requirements, the development team can better focus on building the specific features of the alarm clock (functional) while also ensuring that the overall quality and user experience standards (non-functional) are met

3.

b)

Two Different Readers of a Requirements Specification

1. **Project Manager**
2. **Quality Assurance (QA) Engineer**

What They Will Look for in the Specification

1. Project Manager

Focus Areas:

- **Scope**: Ensures the project goals and boundaries are clearly defined.
- **Timelines**: Checks the project schedule and key milestones.
- **Resources**: Look at the necessary personnel, technology, and budget.
- **Risks**: Identifies potential problems and mitigation plans.

2. Quality Assurance (QA) Engineer

Focus Areas:

- **Testability**: Ensures all requirements can be tested.
- **Acceptance Criteria**: Looks for conditions that define when a requirement is met.
- **Consistency**: Checks that all parts of the document are in agreement.
- **Traceability**: Ensures each requirement can be traced back to a specific need.

Both readers aim to ensure the requirements specification is clear, complete, and actionable.

4

a)

Coupling in Software

Coupling is a measure of how much different software components depend on each other. Lower coupling is generally preferred as it indicates a design where components can be modified independently.

1. Content Coupling

Explanation: Content coupling occurs when one module directly accesses or modifies the content of another module. This creates a very high degree of dependency, as any change in the accessed module can affect the other module directly.

Example: Consider two classes, `ClassA` and `ClassB`. If `ClassA` directly accesses and modifies the private data of `ClassB`, it is an example of content coupling.

python

```
class ClassB:

    def __init__(self):

        self._private_data = 42

 # Underscore indicates private data.


class ClassA:

    def manipulate_classB(self, b_instance):

        b_instance._private_data = 99  # Directly
accessing private data.
```

Here, `ClassA` is tightly coupled with `ClassB` because it directly manipulates `ClassB`'s internal state.

2. Control Coupling

Explanation: Control coupling occurs when one module controls the behavior of another module by passing it information on what to do (e.g., passing control flags). This type of coupling can lead to complex interdependencies between modules.

Example: Consider two functions where one function passes a control flag to another function to dictate its behavior.

python

```
def process_data(flag):
```

```
    if flag == "uppercase":
        return "data".upper()
    elif flag == "lowercase":
        return "data".lower()

def caller_function():
    result = process_data("uppercase")  # Passing a
control flag.
    print(result)
```

Here, `caller_function` is control-coupled with `process_data` because it tells `process_data` how to process the data by passing a control flag.

3. Data Coupling

Explanation: Data coupling occurs when modules communicate by passing only the data that is necessary. This is the most desirable form of coupling because it reduces dependencies and makes the modules more independent and reusable.

Example: Consider two functions where one function passes only the necessary data to another function.

python

```
def add(a, b):
    return a + b
```

```
def caller_function():
    result = add(5, 3)  # Passing only necessary data.
    print(result)
```

Here, `caller_function` is data-coupled with `add` because it only passes the necessary data (the numbers to be added), without dictating how `add` should perform its operation.

Summary

- **Content Coupling**: One module directly modifies the internal data of another module.
    - Example: A class directly accessing and modifying the private data of another class.
- **Control Coupling**: One module controls the behavior of another module by passing control information.
    - Example: A function passing a control flag to another function to dictate its behavior.
- **Data Coupling**: Modules share data through parameters, passing only the necessary data.
    - Example: A function passing two numbers to another function for addition.

b)

Achieving Low Coupling through Partitioning and Layering

1. Partitioning

Explanation: Partitioning involves dividing a system into smaller, more manageable components or modules. Each module is responsible for a specific set of functionalities, and communication between modules is limited to well-defined interfaces.

Achieving Low Coupling:

- **Isolation of Concerns**: Partitioning allows each module to focus on a specific aspect of the system's functionality. This isolation ensures that changes in one module have minimal impact on other modules, reducing coupling.
- **Clear Interfaces**: Modules communicate through well-defined interfaces, specifying the inputs, outputs, and behaviors they support. This clear separation of concerns and communication channels helps minimize dependencies between modules.
- **Encapsulation**: Each module encapsulates its internal details and exposes only the necessary information through its interface. This encapsulation prevents other modules from directly accessing or modifying internal data, reducing coupling.

Example: Consider a web application divided into modules for user authentication, database access, and user interface. Each module handles a specific aspect of the application's functionality and communicates with other modules through clearly defined interfaces. For instance, the user interface module may send user input data to the authentication module for verification without needing to know the internal workings of the authentication process.

2. Layering

Explanation: Layering involves organizing a system's components into distinct layers, where each layer provides a specific level of abstraction and functionality. Higher layers depend on lower layers, but communication between layers is typically restricted to well-defined interfaces.

Achieving Low Coupling:

- **Abstraction Levels**: Each layer in a layered architecture represents a different level of abstraction, such as presentation, business logic, and data access. This separation ensures that changes in one layer have minimal impact on other layers, reducing coupling.
- **Information Hiding**: Layers encapsulate their internal details and expose only the necessary functionality through well-defined interfaces. This information hiding prevents higher layers from directly accessing or modifying the internal state of lower layers, reducing coupling.
- **Decoupled Communication**: Communication between layers is typically decoupled, meaning that higher layers interact with lower layers through interfaces without needing to know the implementation details. This decoupling helps maintain flexibility and allows for easier modifications.

Example: In a typical web application, you might have a presentation layer (UI), a business logic layer, and a data access layer. The presentation layer interacts with the business logic layer through defined interfaces without needing to know how the business logic is implemented. Similarly, the business logic layer interacts with the data access layer through interfaces, allowing changes to one layer without affecting the others.

c)

Functional cohesion is generally considered more desirable than procedural cohesion. Here's why:

Functional Cohesion

Explanation: Functional cohesion occurs when the elements within a module perform tasks that are logically related and contribute to a single well-defined function or purpose.

Advantages:

1. **Clear Purpose**: Functional cohesion ensures that each module has a clear, well-defined purpose, making the code easier to understand and maintain.
2. **Encapsulation**: Modules with functional cohesion encapsulate related functionality, making it easier to make changes or updates without affecting other parts of the system.
3. **Reusability**: Because each module focuses on a specific function, they are more likely to be reusable in other parts of the system or in different projects.

Procedural Cohesion

Explanation: Procedural cohesion occurs when the elements within a module are grouped based on the sequence of execution, rather than logical relationships.

Disadvantages:

1. **Complexity**: Procedural cohesion can lead to complex and tangled code, as unrelated tasks may be mixed together within the same module.
2. **Difficulty in Maintenance**: Modules with procedural cohesion are often harder to understand and maintain, as changes to one part of the module may inadvertently affect other parts.

3. **Limited Reusability**: Because modules with procedural cohesion are focused on a sequence of operations rather than a specific function, they are less likely to be reusable in other parts of the system or in different projects.

Conclusion

Functional cohesion is generally considered more desirable because it promotes clearer, more maintainable, and reusable code. It helps in creating modules that have a single, well-defined purpose, making the system easier to understand and maintain over time. In contrast, procedural cohesion can lead to complex and brittle code that is difficult to maintain and reuse. Therefore, functional cohesion is preferred in most software design scenarios.

5

a)

In the diary and time management system described, we can identify several possible entity, boundary, and control objects along with their attributes and operations:

Entity Objects:

1. **User:**
    - **Attributes: Name, Email, Availability**
    - **Operations: View Diary, Update Diary, View Appointments**
2. **Appointment:**
    - **Attributes: Date, Time, Duration, Participants**
    - **Operations: Schedule, Reschedule, Cancel**

Boundary Objects:

1. **User Interface:**
    - **Attributes: Display, Input Fields, Buttons**
    - **Operations: Display Diary, Accept User Input, Display Notifications**
2. **Calendar Interface:**
    - **Attributes: Interface with Calendar Systems (e.g., Google Calendar, Outlook)**
    - **Operations: Retrieve Calendar Data, Update Calendar**

Control Objects:

1. **Appointment Scheduler:**
    - **Attributes: Meeting Slots, Availability Matrix**
    - **Operations: Find Common Slot, Reserve Slot, Send Notifications**
2. **Appointment Manager:**
    - **Attributes: Appointment Queue, Pending Appointments**
    - **Operations: Manage Appointments, Update Appointment Status, Notify Users**
3. **User Authentication Module:**
    - **Attributes: User Credentials, Authentication Status**
    - **Operations: Authenticate User, Verify Permissions**

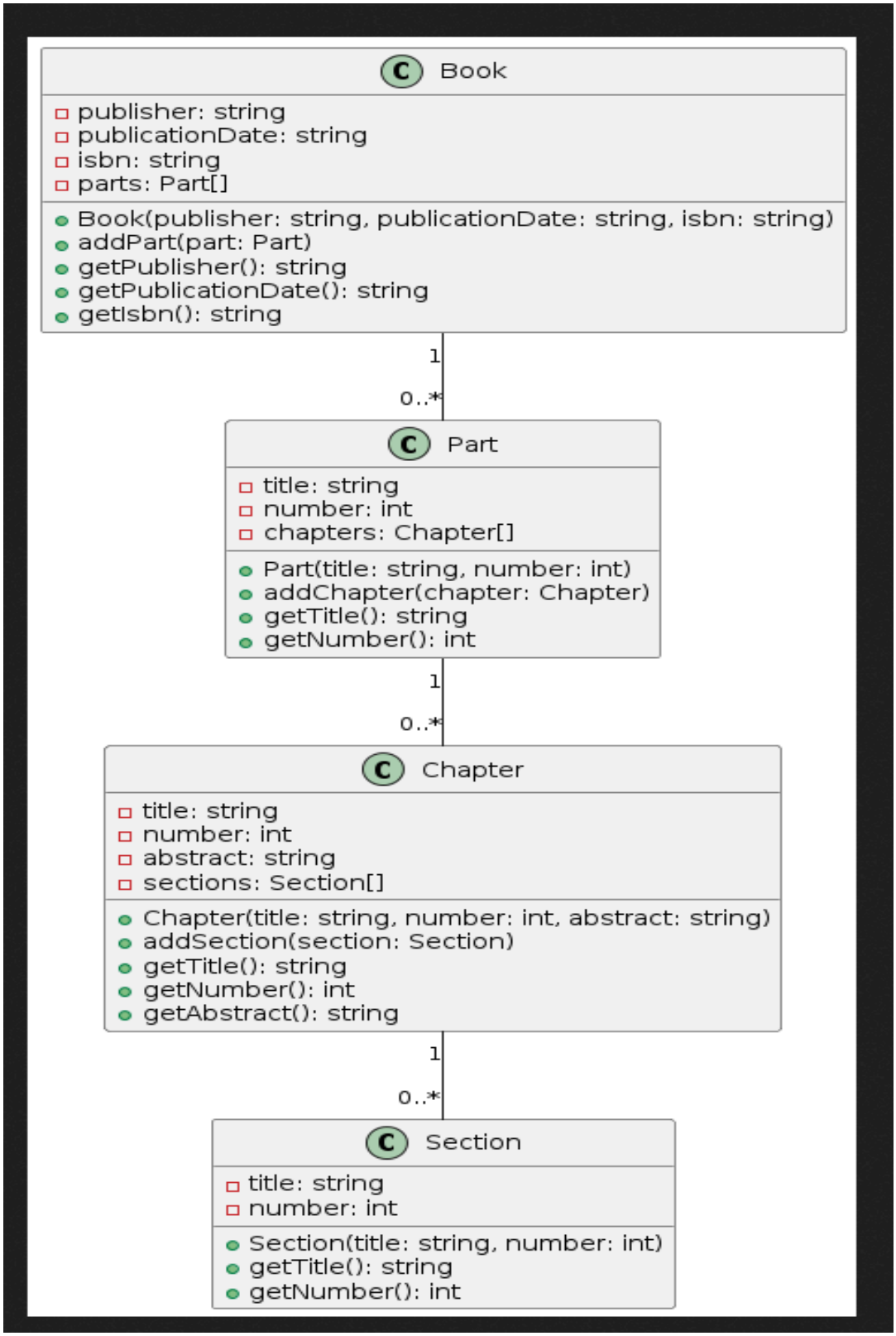
Explanation:

- **Entity Objects: These represent the main data entities in the system, such as users and appointments. They have attributes that describe their properties and operations that define how they interact with the system.**
- **Boundary Objects: These represent the interfaces through which the system interacts with external entities, such as users and calendar systems. They are responsible for displaying information to users and accepting user input.**

- **Control Objects: These represent the modules responsible for controlling the flow of data and logic within the system. They coordinate the interactions between entity and boundary objects to accomplish system tasks, such as scheduling appointments and managing user authentication.**

By identifying and defining these objects along with their attributes and operations, we can create a clear understanding of the system's structure and functionality.

b)

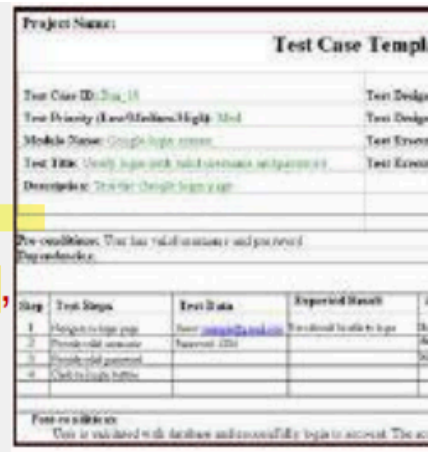

Book
publisher: string
publicationDate: string
isbn: string
parts: Part[]
Book(publisher: string, publicationDate: string, isbn: string)
addPart(part: Part)
getPublisher(): string
getPublicationDate(): string
getIsbn(): string
1
0..*
Part
title: string
number: int
chapters: Chapter[]
Part(title: string, number: int)
addChapter(chapter: Chapter)
getTitle(): string
getNumber(): int
1
0..*
Chapter
title: string
number: int
abstract: string
sections: Section[]
Chapter(title: string, number: int, abstract: string)
addSection(section: Section)
getTitle(): string
getNumber(): int
getAbstract(): string
1
0..*
Section
title: string
number: int
Section(title: string, number: int)
getTitle(): string
getNumber(): int

6

a)

A **complete test case** will contain

1. a test case identifier,
2. a brief statement of purpose,
3. a description of preconditions,
4. the actual test case inputs,
5. the expected outputs,
6. a description of expected post-conditions, and
7. an execution history.



(b) Branch Coverage and Path Coverage in White Box Testing

Branch Coverage: Branch coverage is a white box testing technique that aims to ensure that every possible branch or decision in the code is executed at least once during testing. It measures the percentage of branches in the code that have been executed by the test cases.

Example: Consider the following Python code snippet:

python

```
def is_positive(num):
    if num > 0:
        return True
    else:
```

```
        return False
```

To achieve 100% branch coverage for this code, we need to have test cases that execute both the `if` and `else` branches. For example:

- Test Case 1: Input `num = 5` (executes the `if` branch).
- Test Case 2: Input `num = -3` (executes the `else` branch).

Path Coverage: Path coverage, on the other hand, aims to ensure that every possible path through the code is executed at least once during testing. It considers all possible combinations of branches and loops in the code.

Example: For the same code snippet as above, achieving 100% path coverage would require test cases that execute every possible path through the code. This includes both branches (`if` and `else`), resulting in a total of two possible paths:

1. `num > 0` is true (executes the `if` branch).
2. `num > 0` is false (executes the `else` branch).

(c) Difference Between White Box Testing, Grey Box Testing, and Black Box Testing

Aspect	White Box Testing	Grey Box Testing	Black Box Testing
Definition	Testing based on an internal perspective of the system, involving knowledge of the code, structure, and implementation.	Testing with partial knowledge of the internal workings of the application, combining elements of both white and black box testing.	Testing based on external perspective of the system, without knowledge of the code, structure, or implementation.
Tester's Knowledge	Requires detailed knowledge of the internal workings of the application, including code, architecture, and logic.	Requires some knowledge of internal workings, but not as detailed as white box testing.	Requires no knowledge of internal workings; focuses solely on functionality and requirements.
Focus	Verifies internal structures, logic, data flow, and control flow.	Verifies functionality as well as some aspects of internal processes.	Verifies functionality against requirements without considering internal structures.
Techniques Used	Code coverage, path coverage, branch coverage, unit testing.	Regression testing, system testing, integration testing with partial knowledge of internal processes.	Functional testing, user acceptance testing, system testing, and integration testing based on requirements.
Test Cases	Derived from code logic, structure, and design specifications.	Derived from functional specifications, with some consideration of internal processes.	Derived from functional specifications, requirements, and use cases.
Typical Testers	Developers, white box testers, and sometimes security analysts.	Testers with experience in both black box and white box testing, such as integration testers.	End-users, QA testers, and black box testers.

7

(a)The communication structure among participants of a project can be visualized in a simple and clear manner using a hierarchical diagram that shows the flow of information and relationships. Here's a basic outline:

Communication Structure Among Project Participants

1. **Project Sponsor/Client**
    - **Role**: Provides funding, sets project goals, and oversees the project's overall direction.
    - **Communication**: Primarily with the Project Manager.
2. **Project Manager**
    - **Role**: Manages the project, coordinates between all participants, and ensures the project meets its goals.
    - **Communication**: With the Project Sponsor/Client, Team Leads, and Stakeholders.
3. **Team Leads**
    - **Role**: Lead specific functional teams (e.g., Development, Testing, Design).
    - **Communication**: With the Project Manager and their respective team members.
4. **Team Members**
    - **Role**: Execute tasks and work on specific project deliverables.
    - **Communication**: With their Team Leads and peers within their functional team.
5. **Stakeholders**
    - **Role**: Individuals or groups with an interest in the project's outcome (e.g., users, management, regulatory bodies).
    - **Communication**: Primarily with the Project Manager, occasionally with Team Leads.

(b) Associations Between Tasks, Activities, Roles, Work Products, and Work Packages

Associations between various project elements help define the relationships and dependencies within a project. Here's how tasks, activities, roles, work products, and work packages are typically associated:

1. Tasks and Activities:
    - Tasks are specific actions or units of work that need to be completed to achieve project objectives.
    - Activities are collections of related tasks that contribute to achieving a specific milestone or deliverable.
    - Associations: Each activity consists of one or more tasks, and tasks are grouped together based on their relationship to specific activities.
2. Roles and Work Products:
    - Roles represent the responsibilities and functions assigned to individuals or groups within the project.
    - Work products are tangible deliverables produced as a result of project activities.
    - Associations: Each role is responsible for producing or contributing to specific work products, and work products are associated with the roles responsible for their creation or review.
3. Work Packages and Tasks/Activities:
    - Work packages are collections of related tasks or activities that are grouped together for planning, scheduling, and tracking purposes.
    - Associations: Each work package consists of one or more tasks or activities, and tasks/activities are organized into work packages based on their logical grouping or dependency.

By establishing clear associations between these project elements, project managers can effectively plan, monitor, and control project activities, ensuring alignment with project objectives and stakeholder expectations.