

CSC582 Object-oriented Analysis and Design
Assignment 3 - Chapters 7, 8, and 11
(Group 5)

Team Members

Ratnesh Pawar [16.66]

Rohan Sonawane [16.66]

Abhishek Dhale [16.66]

Mayank Sharma [16.66]

FNU Sumit Kumar [16.66]

Neil Duraiswami [16.66]

1.

a.

Analysis Model:

The analysis model involves techniques like use case diagrams, activity diagrams, and object identification. It focuses on understanding the requirements of the system from a user's perspective.

Design Model:

The design model employs techniques like class diagrams, sequence diagrams, and state diagrams. It is concerned with translating the requirements identified in the analysis phase into a detailed and implementable design.

b.

Balancing the Models:

- Balancing the models, emphasizes consistency and coherence between different models. Any changes made in one model should be reflected in others to maintain alignment.
- If a change is made in a use case diagram during the analysis phase, corresponding adjustments should be made in the design phase's class diagrams and sequence diagrams to ensure consistency.

c.

- The book explains interrelationships by emphasizing that functional models (e.g., use case diagrams) drive the creation of structural models (e.g., class diagrams) and behavioral models (e.g., sequence diagrams).
- Use case diagrams help identify the functionalities the system needs. These functionalities are then translated into classes (structural model) and their interactions (behavioral model).

2.

a.

- Factoring involves breaking down a complex system into smaller, more manageable subsystems or components.
- Abstraction involves representing the system at a higher level, while refinement involves breaking down abstractions into more detailed and concrete specifications. Factoring supports both processes by managing system complexity.

b.

- Partitioning, as discussed in the book, involves dividing a system into distinct parts or partitions that can collaborate to achieve the overall system functionality.
- In a distributed system, partitions might represent different servers or nodes. Collaboration between these partitions could involve data exchange or communication protocols.

3

a.

A layer, in the context of software architecture and design, is a conceptual division or abstraction that represents a specific aspect or component of a software system. Each layer is responsible for a distinct set of functionalities or

concerns, and it helps organize and modularize the software architecture, making it easier to manage, maintain, and extend.

The textbook suggests five different layers:

- Foundation Layer
- Problem Domain Layer
- Data Management Layer
- Human–Computer Interaction Layer
- Physical Architecture Layer

b.

Certainly, here's a combined explanation of the purpose and benefits of organizing a software system into different layers:

Foundation Layer: The foundation layer contains fundamental classes and components that provide basic functionality. Its purpose is to establish a solid base for the software. Developers benefit from this layer by relying on standardized data types and utility functions without the need for modification. It ensures consistency and reduces the risk of errors in basic operations.

Problem Domain Layer: This layer focuses on the core functionality of the software, addressing specific application requirements. Its purpose is to organize and encapsulate the unique features of the software. Developers benefit by keeping the application logic modular and separate from other concerns. This isolation facilitates easier maintenance, testing, and updates to the core functionality.

Data Management Layer: The data management layer handles data storage and retrieval, abstracting underlying storage mechanisms. Its purpose is to enable flexible data handling within the software. Developers benefit from the layer's abstraction, allowing problem domain classes to work with data without tight coupling to specific storage solutions. This enhances data portability, flexibility, and adaptability.

Human–Computer Interaction Layer: This layer is responsible for creating and managing the user interface (UI) components and interactions. Its purpose is to separate UI concerns from the core logic. Developers benefit by designing and updating UI elements independently, improving usability, supporting various UI platforms, and simplifying UI testing.

Physical Architecture Layer: This layer addresses how the software interacts with the hardware and network infrastructure. Its purpose is to abstract hardware complexities and enhance adaptability. Developers benefit by isolating hardware-related concerns, enabling scalability, efficient resource management, and portability across different environments.

c.

Foundation Layer:

- **Fundamental Data Types:** This layer can include classes representing basic data types such as integers, real numbers, characters, and strings. These classes define the core building blocks for data manipulation.
- **Data Structures:** It may contain classes representing fundamental data structures like lists, trees, graphs, sets, stacks, and queues. These classes provide efficient data organization and manipulation tools.
- **Utility Classes:** Utility classes for common operations, such as date and time manipulation or currency handling, can be part of this layer. They offer standardized functions to simplify coding tasks.

Problem Domain Layer:

- **Entities:** Classes in this layer represent domain-specific entities, such as "Employee," "Customer," or other objects directly related to the problem domain.

- **Business Logic:** This layer includes classes that encapsulate the core business logic of the application. They define the rules, calculations, and algorithms unique to the software's purpose.
- **Service Classes:** Service classes may be present to provide specific functionalities or services within the problem domain, abstracting complex operations from higher layers.

Data Management Layer:

- **Data Access and Manipulation (DAM) Classes:** These classes enable the software to interact with different data storage solutions (e.g., databases). They handle tasks like data retrieval, updates, and transactions.
- **Data Models:** Classes representing data models or schemas define how data is structured and mapped in the data storage systems.
- **Query Builders:** Some classes assist in constructing and executing database queries, improving efficiency and security.

Human-Computer Interaction Layer:

- **User Interface Components:** This layer includes classes for creating user interface elements, such as buttons, text fields, windows, and dialogs. These classes manage the visual and interactive aspects of the application.
- **Controller Classes:** Controllers may be present to handle user input and coordinate interactions between the UI components and the core application logic.
- **Event Handlers:** Classes that define event handlers or listeners, responsible for responding to user actions like button clicks or mouse movements.

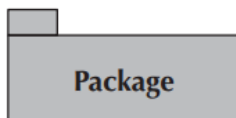
Physical Architecture Layer:

- **Hardware Interaction Classes:** This layer can contain classes for interacting with specific hardware components, such as classes that manage communication with peripheral devices or sensors.
- **Network Communication:** Classes for managing network communication, including protocols and socket handling, are part of this layer.
- **Server or Cloud Interaction:** For cloud-based or server-dependent applications, classes for interacting with remote servers, APIs, or cloud services are included.

4.

a.

A package in UML is a logical grouping of UML elements. It is used to simplify UML diagrams by grouping related elements into a single, higher-level element.



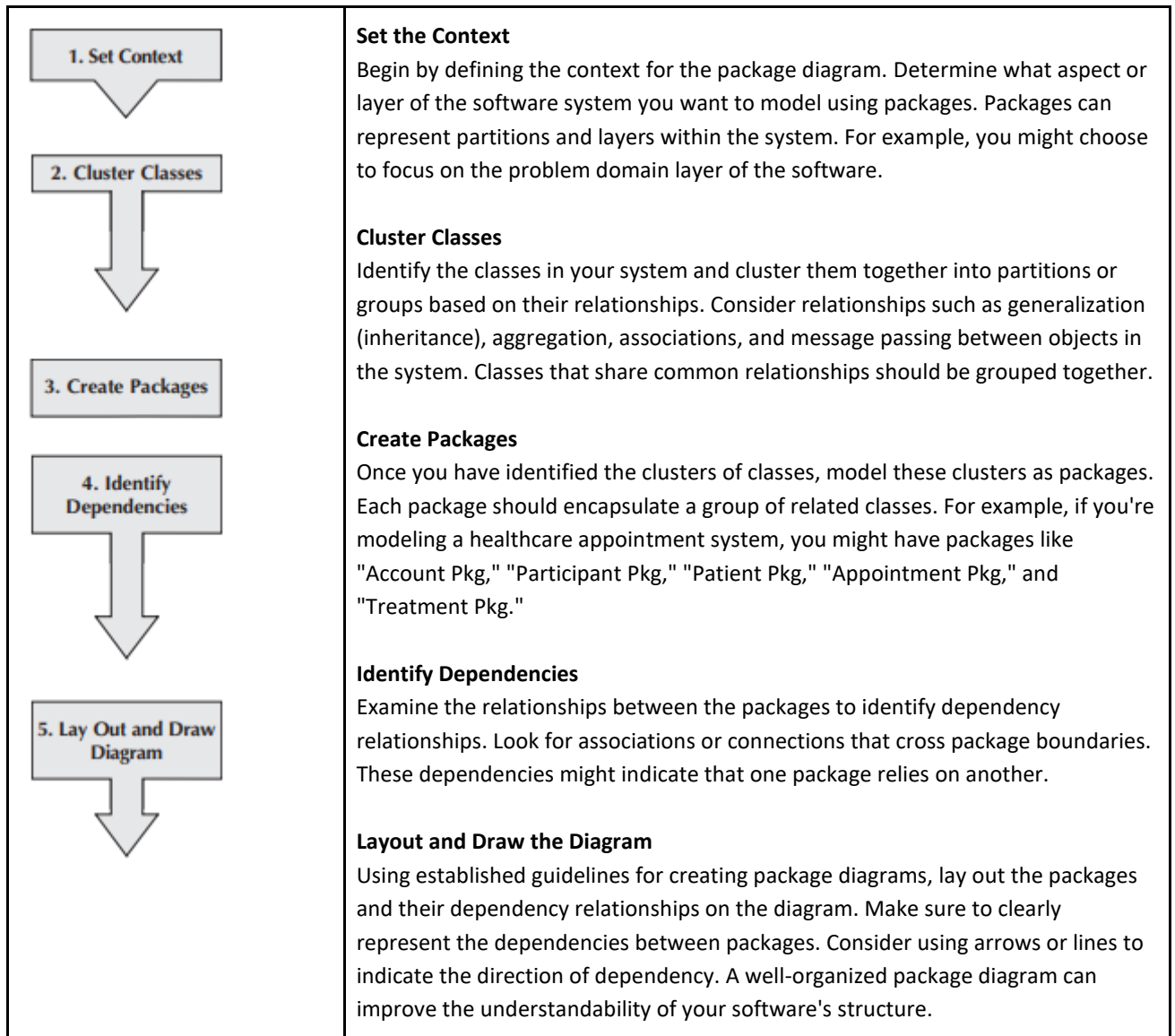
Packages and Their Relation to Partitions and Layers

Packages can also be used to model partitions and layers within a system.

For example: when creating package diagrams, the first step often involves setting the context, which could be a particular layer or partition in the system.

The subsequent step involves clustering classes into partitions based on their relationships, such as generalization, aggregation, and various associations. This process aids in identifying which elements should be grouped together in a package within the specified context.

b.



c. **Guideline to follow while drawing Package Diagrams**

Logical Organization

Use package diagrams to logically organize your system's design. Packages are used to group related classes, and this grouping is based on relationships such as inheritance, aggregation, composition, or collaboration. This helps in representing the structure of your system in a more organized and comprehensible manner.

Relationship Placement

When relationships like inheritance, aggregation, or association exist between packages, consider their placement for readability. Vertical placement is often used to represent inheritance relationships, with the superclass package above the subclass package. Horizontal placement can be used for aggregation and association relationships, with the packages placed side by side.

Dependency Relationships

When showing dependency relationships between packages, make sure to indicate the direction of the dependency. Typically, this is from the subclass/package to the superclass/package, from the part to the whole, or from the client to the server. Dependencies represent semantic relationships, indicating that one element relies on the existence of another.

Include Actors and Associations

If you're using packages to group use cases together, make sure to include actors and their associations with the use

cases within the package. This provides context and helps users understand how actors interact with the use cases contained in the package.

Descriptive Package Names

Give each package a simple but descriptive name. The name should convey the purpose and content of the package. This helps users quickly understand what the package encapsulates without needing to drill down into it.

Cohesive Packages

Ensure that packages are cohesive, meaning the classes contained within a package should, in some sense, belong together. This cohesion is often based on the level of interdependence among the classes. If classes within a package rely heavily on each other, they are more likely to belong together in that package.

5.

a.

When designing a specific class, several types of additional specifications may be necessary to ensure its proper implementation and functionality. Some of these specifications include:

Attributes and Methods

Ensure that the class contains all the necessary attributes and methods required to fulfill its designated purpose. Check for any redundant or unnecessary attributes or methods that might be present and remove them if needed.

Visibility

Define the visibility of attributes and methods within the class, which determines whether they are accessible or hidden to other classes. Decide whether each attribute and method should be private, public, or protected based on the class's intended usage and security considerations.

Method Signatures

Clearly define the method signatures, including the method name, the parameters to be passed, their data types, and the type of value that the method will return. Ensure that the method signatures are consistent with the class's functionality and adhere to the overall design architecture.

Constraints

Specify any constraints that the class and its attributes must adhere to, such as preconditions, postconditions, and invariants. Determine the conditions under which the class can operate and the restrictions that should be enforced to maintain data integrity and ensure proper behavior.

Error Handling

Define how the class should handle various types of errors and exceptions, including how the system should respond to violations of specified constraints. Determine whether the system should simply abort, automatically undo the action, or provide the user with options to rectify the error.

By considering these additional specifications during the design phase, we can ensure that the class functions as intended, maintains data integrity, and adheres to the overall system architecture.

b.

Patterns, frameworks, class libraries, and components are all software engineering concepts that aid in the development of complex systems and promote reusability. They each serve distinct roles in enhancing the evolving design of a system:

Patterns

Software design patterns are reusable solutions to common design problems. They provide a template for solving recurring design issues and offer a set of guidelines for structuring code to achieve specific objectives. Design patterns are categorized into creational, structural, and behavioral patterns, each serving a different purpose in software development.

Frameworks

Software frameworks are a collection of pre-implemented classes, modules, and functions that provide a foundation for building applications. They offer a structure and set of tools to streamline the development process by providing predefined functionalities, standardizing development practices, and enforcing coding conventions. Frameworks can be specific to certain domains or general-purpose, such as web application frameworks, testing frameworks, and more.

Class Libraries

Class libraries are collections of reusable classes and components that can be utilized to simplify and expedite the development process. They provide a set of pre-defined functionalities and data structures that can be incorporated into applications. Class libraries can be specific to certain functionalities like numerical processing, file management, or user interface development, and they are often utilized to develop frameworks and components.

Components

Software components are self-contained, encapsulated units of software that provide specific functionalities and can be easily integrated into larger systems. They have well-defined APIs (Application Programming Interfaces) that allow other components or systems to interact with them. Components are designed to be plug-and-play, making them easily interchangeable and reusable across different projects.

These tools are used to enhance the evolving design of a system in various ways:

Reusability

All of these concepts promote reusability by providing pre-defined solutions and functionalities that can be integrated into multiple projects without having to redevelop them from scratch.

Consistency

They enforce coding standards and best practices, ensuring consistency across different parts of the system and facilitating easier maintenance and collaboration among team members.

Efficiency

By leveraging patterns, frameworks, class libraries, and components, developers can streamline the development process, reduce development time, and improve overall efficiency.

Scalability

These tools often provide a scalable structure that allows for the seamless expansion and modification of the system as the project evolves and requirements change.

Overall, the use of these tools and concepts promotes a more structured, efficient, and maintainable software development process, allowing developers to focus on building the unique components of the system while leveraging well-established solutions for common design problems.

6.

a.

In the context of software engineering and object-oriented design, constraints refer to the rules and conditions that must be adhered to by various elements within the system to ensure its proper functioning and integrity. These constraints are essential for maintaining the expected behavior and structure of the software. There are three different types of constraints typically considered in object-oriented design:

Preconditions

Preconditions represent the set of conditions that must be satisfied before a method can be executed. They specify the requirements that the system expects to be in place before a particular operation is performed. For example, preconditions might ensure that the parameters passed to a method are valid and meet the required criteria for the method to execute without errors.

Postconditions

Postconditions represent the set of conditions that must be met after the execution of a method or operation. They define the expected outcomes or states that the system should achieve after the method has completed its execution. Postconditions ensure that the method does not leave the system in an invalid state and that the intended changes have been applied successfully.

Invariants

Invariants are constraints that must hold true throughout the execution of the program. They represent the properties or conditions that remain constant for all instances of a class. Invariants define the rules that every object of a certain class must adhere to, ensuring the consistency and validity of the data and relationships within the system. Examples of invariants include the types of attributes, their valid values, multiplicity constraints, and the integrity of association and aggregation relationships.

b.

The purpose of a contract in the context of object-oriented design is to document the message passing between objects, specifically focusing on the methods that can receive messages from other objects. Contracts serve as a means of communication between different components of the system, providing a clear understanding of what each method is intended to do without detailing the specific algorithmic implementations. They aid in maintaining a declarative understanding of the responsibilities of a method, ensuring that the developer comprehends the expected interactions and outcomes of the method.

Contracts are utilized in the following ways:

Method Documentation

Contracts document essential information related to a method, such as its name, class name, ID number, associated use cases, description, arguments received, type of data returned, and pre- and postconditions. They provide a concise yet comprehensive overview of the method's purpose and functionality.

Establishing Clarity

Contracts establish clarity by outlining the expectations and responsibilities of the method, helping both developers and stakeholders understand the intended behavior and results of the method's execution.

Guiding Implementation

By specifying pre- and postconditions, contracts guide the implementation of methods, ensuring that developers adhere to the defined constraints and requirements while designing and coding the software.

Maintaining Consistency

Contracts aid in maintaining consistency throughout the development process by serving as a reference point for understanding the interactions and expectations associated with a particular method.

Facilitating Collaboration

Contracts enable effective collaboration between different teams of developers working on various parts of the system, providing a common understanding of the functionalities and behaviors of different components.

Overall, contracts act as a crucial communication tool in the design and development of software systems, promoting clarity, consistency, and effective collaboration among development teams. They serve as a bridge between the

problem domain and the solution domain, facilitating a smooth transition from the conceptual understanding of the system to its practical implementation.

c.

Here is an example of a contract for a `computePay` method associated with an `HourlyEmployee` class, specifying the preconditions and postconditions using the Object Constraint Language (OCL):

Method Name: computePay	Class Name: HourlyEmployee	ID: HEMP001
Clients (Consumers): PayrollSystem, TimeTrackingSystem		
Associated Use Cases: CalculateEmployeePay, GeneratePayrollReport		
Description of Responsibilities: Compute the pay for the hourly employee based on the number of hours worked and the hourly rate.		
Arguments Received: hoursWorked: Integer, hourlyRate: Real		
Type of Value Returned: Real (representing the pay amount)		
Pre-Conditions: <ul style="list-style-type: none">• The hoursWorked argument must be greater than or equal to 0.• The hourlyRate argument must be greater than 0.		
Post-Conditions: <ul style="list-style-type: none">• The computed pay should be greater than or equal to 0.• The computed pay should be equal to the product of hoursWorked and hourlyRate.		

7.

a.

We can use Structured English or a similar formal language to specify the algorithm of the method. Structured English is a way of writing clear and structured instructions describing the steps of a product. It looks like a simple programming language and is used to define what the method does and how it manipulates the data. Note that the format of Structured English may vary between organizations.

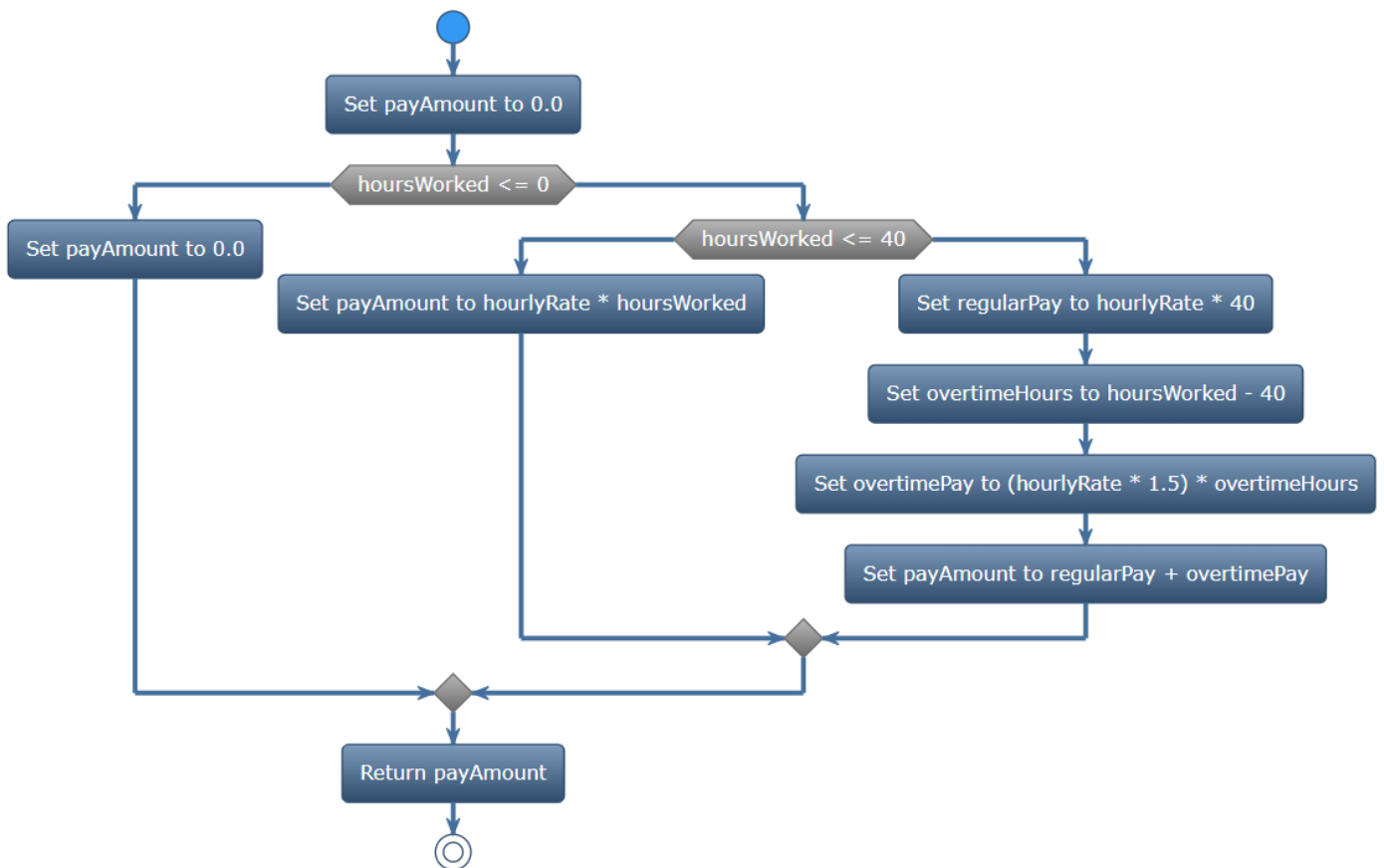
b.

```
Begin
  Set payAmount to 0.0 # Initialize payAmount to zero.

  If hoursWorked <= 0 Then
    Set payAmount to 0.0 # No work, no pay.
  Else If hoursWorked <= 40 Then
    Set payAmount to hourlyRate * hoursWorked # Regular pay for up to 40 hours.
  Else
    Set regularPay to hourlyRate * 40
    Set overtimeHours to hoursWorked - 40
    Set overtimePay to (hourlyRate * 1.5) * overtimeHours
    Set payAmount to regularPay + overtimePay # Pay for regular and overtime hours.
  End If

  Return payAmount
End
```


c.



Activity diagram illustrating the algorithm for computing pay based on hours worked and hourly rate.

8.

a.

Server-Based Architectures:

Characteristics:

- The server performs all four software functions: data storage, data access logic, application logic, and presentation logic.
- Clients are essentially input/output devices, capturing and sending user input to the server for processing.

Advantages:

- Simplicity and centralized control.
- All software and data are stored on a single server.

Disadvantages

- Overloaded servers as demands for applications increase.
- Upgrades are challenging and expensive, usually in large increments.

Client-Based Architectures:

Characteristics:

- Clients are personal computers on a local area network (LAN), responsible for presentation, application, and data access logic.
- Server's role is mainly to store data.

Advantages

- Simplicity and often works well.
- Decentralized processing on client computers.

Disadvantages:

- Network circuits can become overloaded.
- All data on the server must travel to the client for processing, leading to potential network and client power overload.

Client-Server Architectures:

Characteristics

- Processing is balanced between the client and the server.
- Clients handle presentation logic, while servers manage data access logic and data storage.
- Application logic may reside on either the client, server, or be split between both.

Advantages

- Scalability: Easy to increase or decrease storage and processing capabilities by adding servers.
- Supports diverse types of clients and servers.
- More reliable than server-based architectures; no central point of failure.

Disadvantages

- Complexity in developing and maintaining software on both clients and servers.
- Updating software requires modifications on all clients and servers.
- Higher development and maintenance costs compared to server-based architectures.

b.

Aspect	Two-Tiered Architecture	Three-Tiered Architecture	n-Tiered Architecture
Responsibilities	<ul style="list-style-type: none"> • Client: Application and presentation logic. • Server: Responsible for data. 	<ul style="list-style-type: none"> • Client: Presentation logic. • Application Server(s): Application logic. • Database Server(s): Data access logic and data storage. 	<ul style="list-style-type: none"> • Client: Presentation logic. • Database Servers: Data access logic and data storage. • Application Servers: Distribute application logic across multiple servers.
Configuration	<ul style="list-style-type: none"> • Uses only two sets of computers: clients and servers. 	<ul style="list-style-type: none"> • Involves three sets of computers: clients, application servers, and database servers. 	<ul style="list-style-type: none"> • Utilizes more than three sets of computers, with application logic spread across multiple servers.
Advantages	<ul style="list-style-type: none"> • Simplicity and straightforward design. • Direct communication between the client and server. 	<ul style="list-style-type: none"> • Better separation of concerns between presentation, application, and data layers. 	<ul style="list-style-type: none"> • Enhanced scalability by balancing the load on different servers. • Flexibility to replace or add servers based on specific needs.
Differences	<ul style="list-style-type: none"> • Scalability: Limited scalability compared to n-tiered architectures. • Configuration Complexity: Simpler design and understanding. • Network Load: Lower network load. 	<ul style="list-style-type: none"> • Scalability: Intermediate scalability. • Configuration Complexity: Moderate complexity. • Network Load: Moderate network load. 	<ul style="list-style-type: none"> • Scalability: High scalability. • Configuration Complexity: Higher complexity. • Network Load: Higher network load.

9.

a.

Nonfunctional requirements are an essential set of criteria that describe how a system operates, as opposed to specific behaviors or functionalities. These requirements have a significant impact on the design of a system's physical architectural layer. Based on the provided information, the primary nonfunctional requirements and how they influence physical architecture layer design can be described as follows:

- **Operational Requirements:**

The environments in which the system needs to operate are defined by the system software, operating systems, and interoperability. Because of this, the physical architecture must be able to work with a variety of operating systems, databases, and hardware configurations. For instance, if an application is meant to function in a loud industrial setting, the design must account for this without relying on sound alerts. These requirements directly affect the choice of hardware and software that the system will work on. The architecture must be designed to operate within the defined environments, and compatibility with specific operating systems or databases is a must. It should also perform in specific physical environments.

- **Technical Environment Requirements:**

For the system to be executed seamlessly, it is imperative that the physical architecture supports a range of hardware and software platforms, including but not limited to cloud computing, mobile platforms, and specialized operating systems like Windows or Linux. These requirements dictate the types of hardware and system software that the architecture must support. If an application must support mobile and cloud computing, the architecture needs to be designed with the necessary servers, databases, and network configurations to support these technologies.

- **System Integration Requirements:**

The system's physical architecture must confidently support the necessary communication protocols and data exchange formats to enable seamless integration with other systems. These requirements drive the need for an architecture that can seamlessly connect and exchange data with other systems. This may require specific network configurations, middleware, or APIs that enable smooth integration.

- **Portability Requirements:**

The system must meet standards that enable it to adapt to changing operational environments and business needs. The design should allow for these changes with minimal interruption. These requirements affect the choice of technologies and the design of the architecture, ensuring that the system can be easily moved to different platforms or technologies in the future.

- **Maintainability Requirements:**

The system should be easily changeable and maintainable, accommodating future modifications and allowing for simple updates. These requirements influence the architecture to be modular and flexible, allowing for easy updates and changes.

- **Performance Requirements:**

The architecture must be scalable, dependable, and able to handle projected user volume to manage variable loads and enable quick reaction times. These requirements have a direct impact on the selection of hardware and the structuring of software components. High-speed requirements may lead to the selection of more powerful servers, while high-capacity requirements may dictate scalable solutions like load balancing or cloud services.

- **Security Requirements:**

To guarantee security, it is imperative to fortify the system against unauthorized access and data breaches by implementing robust firewalls, encryption, and other necessary measures, particularly for sensitive data. These requirements require the architecture to incorporate secure communication protocols, encryption, access controls, and possibly isolated networks to protect sensitive data and operations.

- **Cultural and Political Requirements:**

It is crucial to customize global systems to meet the cultural and political requirements of each region, including language and compliance with local laws and regulations. To achieve this, the architecture must segregate presentation logic from application logic. These requirements may necessitate the design of a flexible user interface layer that can be customized for different languages and cultural norms without altering the underlying application logic and data layers.

- **Legal Requirements:**

Compliance with laws and regulations is non-negotiable for the architecture. The handling and maintenance of user data, for instance, must align with data privacy rules. These requirements can impose constraints on data storage, processing, and transmission, influencing choices such as data centre locations, compliance with international standards, and auditing capabilities.

b.

It is essential to define non-functional requirements in detail, even if the technical environment requirements specify a specific architecture. This is because non-functional requirements play a crucial role in refining the system's design and implementation stages. They provide a detailed roadmap for selecting specific hardware and software, guiding the system's scalability, security, and maintainability beyond just choosing the architecture.

Detailed non-functional requirements ensure that the system can adapt to future changes in the business environment or technology while considering portability and maintainability from the outset. Performance requirements like speed, capacity, and reliability must be detailed to appropriately size the infrastructure and make decisions that affect system efficiency and user satisfaction.

By detailing security requirements, developers can incorporate appropriate security measures at every layer of the architecture, creating a comprehensive security strategy that protects the system from various threats. Similarly, cultural, and political requirements ensure that the system complies with different laws and regulations of various regions and respects cultural sensitivities, which is particularly important for global systems.

Detailed operational requirements ensure that the system is compatible with the existing technical environment and can integrate smoothly with other systems and processes. Finally, understanding detailed non-functional requirements helps organizations optimize the cost and resources needed for the system, ensuring that it is not over-engineered for the required performance or under-prepared for future changes.

Nonfunctional requirements are critical for understanding how a system should operate and interact with its environment, users, and other systems. They are essential for ensuring the long-term success and sustainability of the system, regardless of the technical environment's specific architecture.

10.

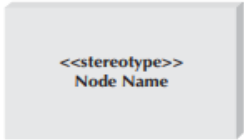
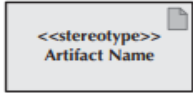
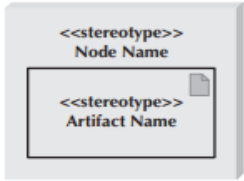
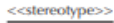
a.

A deployment diagram is a visual representation of how software components are physically deployed on hardware nodes. This type of diagram is created using the Unified Modelling Language (UML) to illustrate the hardware-software mapping and interactions within a system. By showing the physical layout of the hardware and the distribution of software components across it, a deployment diagram provides a comprehensive view of the system's physical infrastructure.

At the heart of a deployment diagram are three main components: nodes, artifacts, and communication paths. Nodes represent the physical components of the system, such as servers, computers, and devices, where artifacts are deployed. Artifacts, on the other hand, represent the software entities of the system, including databases, executable files, and source code that are deployed on the nodes. Communication paths depict the connections between nodes, illustrating how they interact and communicate with each other within the system.

A deployment diagram is a valuable tool for understanding the physical infrastructure and software distribution of complex systems. It is particularly useful for systems that are distributed across different locations or that require a detailed understanding of the physical connections between its components. By providing a clear and detailed view of

the system's physical architecture, a deployment diagram helps developers, system administrators, and other stakeholders to make informed decisions about the design, deployment, and maintenance of the system.

<p>A node:</p> <ul style="list-style-type: none"> ■ Is a computational resource, e.g., a client computer, server, separate network, or individual network device. ■ Is labeled by its name. ■ May contain a stereotype to specifically label the type of node being represented, e.g., device, client workstation, application server, mobile device, etc. 	
<p>An artifact:</p> <ul style="list-style-type: none"> ■ Is a specification of a piece of software or database, e.g., a database or a table or view of a database, a software component or layer. ■ Is labeled by its name. ■ May contain a stereotype to specifically label the type of artifact, e.g., source file, database table, executable file, etc. 	
<p>A node with a deployed artifact:</p> <ul style="list-style-type: none"> ■ Portrays an artifact being placed on a physical node. 	
<p>A communication path:</p> <ul style="list-style-type: none"> ■ Represents an association between two nodes. ■ Allows nodes to exchange messages. ■ May contain a stereotype to specifically label the type of communication path being represented, (e.g., LAN, Internet, serial, parallel). 	

b.

A network model is a powerful tool that provides a graphical representation of the various components of an information system such as servers, communication channels, networks, and their geographical distribution within an organization. Analysts can use different symbols, notations, and diagramming tools like UML deployment diagrams to create these models.

The network model has two primary purposes:

Convey Complexity: It conveys the inherent complexity of an information system, which can be challenging to understand without a visual representation. The model helps stakeholders comprehend the intricate and interconnected elements of the system by providing an overview of its infrastructure. By visually representing the various elements and their relationships, the model helps stakeholders like project teams and decision-makers to grasp the overall structure and complexity of the system.

Show System Integration: The network model illustrates how the software components of the system will integrate with the underlying hardware and network infrastructure. It demonstrates the relationships between different elements like servers, clients, network equipment, and external systems or networks (e.g., Internet service providers). This visualization helps stakeholders understand how all the components work together to support the software applications, making it an essential tool for effective decision-making and successful project implementation.

Parameters	Top-Level Network Model	Low-Level Network Model
Scope and Level of Detail	This diagram provides a high-level overview of the network infrastructure, highlighting major components, locations, and their relationships. It does not delve into specific hardware, software, or configurations.	The low-level network model offers a highly detailed and comprehensive view of the network infrastructure. It not only provides specifics about hardware placements and connections but also includes vital information about software components and configurations at each location.
Purpose	The top-level model is crucial in providing stakeholders with a comprehensive understanding of the network's geographic scope, complexity, and system integration. This enables a thorough assessment of the overall architecture, making it an indispensable tool for decision-making.	Low-level models are an indispensable tool for project teams and IT professionals, providing precise hardware and software configurations required for detailed planning and implementation.
Representation	Top-level diagrams use simplified symbols and representations, such as cloud symbols for network locations and simplified lines for connections, to convey the general structure rather than detailed information.	In low-level models, hardware components are represented by realistic symbols and specific connections are detailed, often with labels or descriptions for individual hardware and software components.
Complexity	These models offer a simplified overview, conveying system complexity without overwhelming stakeholders.	Low-level diagrams capture network intricacies, including hardware placements and connections.
Communication vs. Implementation	Diagrams aid communication for project scope, infrastructure, and high-level decisions.	Low-level models aid IT teams in configuring hardware and software for optimal network performance.
Use Cases	It is useful for scoping projects, communicating with stakeholders, and making decisions about network architecture during early planning stages.	During the later stages of a project, when focus shifts to implementation and configuration, these diagrams guide network setup.

To gain a comprehensive understanding of an information system's network infrastructure, it's crucial to recognize the differences between the top-level and low-level network models. These two models serve different purposes and offer varying levels of detail. By acknowledging these distinctions, you'll be better equipped to make informed decisions regarding your network infrastructure, ultimately leading to a more efficient and effective system.