2017

a) What is class diagram?

Class diagrams are a type of UML (Unified Modeling Language) diagram used in software engineering to visually represent the structure and relationships of classes in a system. UML is a standardized modeling language that helps in designing and documenting software systems. They are an integral part of the software development process, helping in both the design and documentation phases.

A class diagram in the Unified Modeling Language (UML) is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects.

b) List the characteristics of unified process.

Ans.

i. Iterative and Incremental.

ii. Use Case Driven :The Unified Process focuses on identifying and prioritizing use cases that represent the system's functionality from the user's perspective.

iii. Architecture-Centric: The Unified Process emphasizes defining and refining the system architecture throughout the development process.

iv. Risk Management: Unified Process identifies and manages project risks proactively to minimize their impact on the project's success.

v Continuous Validation: Unified Process ensures continuous validation of the system's requirements, design, and implementation through reviews, testing, and feedback.

c) Define actor in use case diagram.

Ans.An actor represents a role that interacts with the system. This role could be a person, another system, or an external entity that interacts with the system to achieve a goal. Actors are not part of the system itself but are external entities that either use or are affected by the system's functionalities.

d) What are the types of interaction diagram?

Ans. In UML (Unified Modeling Language), interaction diagrams are used to model the dynamic aspects of a system, focusing on the flow of control and data among objects or components. There are two main types of interaction diagrams:

Interaction diagrams in UML are used to visualize how objects in a system interact with each other to accomplish a task. The main types of interaction diagrams are:

i. Sequence diagrams
ii. Collaboration diagrams

e) What is dynamic model?

Ans. A dynamic model in software engineering represents the behavior of a system over time, focusing on how the system responds to various events, changes, and interactions. Unlike static models, which capture the structure and components of a system (such as class diagrams), dynamic models emphasize the flow of control, data, and interactions within the system as it operates.

f) Define a conceptual class.

Ans. In Object-Oriented Analysis and Design (OOAD), a conceptual class is an abstract representation of a key entity or concept within the problem domain, capturing its essential properties and behaviors without concern for implementation details. It serves as a foundational building block in the analysis phase, helping to model and understand the domain by identifying significant objects, their attributes, and relationships.

An conceptual class is an idea, thing or object.

g) When an aggregation is said to be composite aggregation?

Ans. An aggregation is said to be a composite aggregation (also known as a composition) when the relationship between the whole and its parts is so strong that the parts cannot exist independently of the whole. In this relationship, the lifetime of the part is tied to the lifetime of the whole; if the whole is destroyed, the parts are also destroyed.

h) What is activity diagram?

Ans. An Activity Diagram in UML is a type of behavioral diagram that visually represents the flow of activities or actions in a system or process. It's similar to a flowchart but focuses on the dynamic aspects of the system, particularly the flow of control from one activity to another.

i) List any two types of common association.

Ans.

In Object-Oriented Analysis and Design (OOAD), common types of associations that describe relationships between classes or objects include:

1. Aggregation:

Description: Aggregation represents a "whole-part" relationship where one class (the whole) is composed of one or more instances of other classes (the parts). However, the parts can exist independently of the whole.

Example: A "Library" and "Books" relationship where a library contains books, but the books can exist outside of the library.

2. Composition (Composite Aggregation):

Description: Composition is a stronger form of aggregation where the parts are dependent on the whole. If the whole is destroyed, the parts are also destroyed, indicating a lifecycle dependency.

Example: A "House" and "Rooms" relationship, where rooms are part of a house, and if the house is demolished, the rooms cease to exist as well.

j) What is observer pattern?

Ans. The Observer Design Pattern is a behavioral design pattern that defines a one-to-many dependency between objects so that when one object (the subject) changes state, all its dependents (observers) are notified and updated automatically.

2018

a) What does actor represent in use case diagram?

Ans. In a use case diagram, an actor represents a user, external system, or entity that interacts with the system being modeled. Actors are not part of the system itself but rather interact with it to achieve specific goals or perform particular tasks. They help define the system's external interfaces and the functionality it needs to provide.

b) What is layered architecture?

Ans. Layered architecture in Object-Oriented Analysis and Design (OOAD) is a design pattern that structures a system into distinct layers, each with specific responsibilities. This approach helps to manage complexity by separating concerns and promoting modularity, making the system more manageable, maintainable, and scalable.

c) Define 100% rule.

Ans. The 100% rule in Object-Oriented Analysis and Design (OOAD) refers to the principle that every element of a system should be represented in the design model. This rule ensures that the design captures all necessary aspects of the system without leaving any gaps, which helps in creating a complete and accurate model of the system.

The 100% rule helps in ensuring that the design process is thorough and that all necessary components and interactions of the system are considered and documented, leading to a more complete and reliable system design.

d) What is abstract class?

Ans.

In Object-Oriented Analysis and Design (OOAD), an abstract class is a class that serves as a blueprint for other classes. It cannot be instantiated on its own, meaning you cannot create objects directly from an abstract class. Instead, it is intended to be subclassed, with its subclasses providing implementations for its abstract methods (methods that are declared but not defined within the abstract class).

An abstract class is a class in object-oriented programming that cannot be instantiated on its own and is meant to be subclassed. It serves as a base class for other classes and can define common characteristics and behaviors that its subclasses will inherit.

e) When "includes" is used in use case diagram?

Ans. In a use case diagram, the <<include>> relationship is used to represent a scenario where a use case explicitly incorporates the behavior of another use case. This inclusion helps to avoid duplication of common functionality across multiple use cases, making the model more modular and easier to maintain.

f) What is the use of activity diagram?

Ans. An activity diagram is a type of UML diagram used to model the workflow of a system or process, showing the sequence of activities and their interactions. It provides a visual representation of how different activities are coordinated to achieve a specific outcome, capturing both the flow of control and the flow of data.

An activity diagram is a type of UML (Unified Modeling Language) diagram that is used to model the dynamic aspects of a system. It focuses on representing the flow of activities or actions and the sequence in which they occur. Activity diagrams are particularly useful

for visualizing workflows, business processes, and the logic of complex operations within a system.

i. Modeling Business Processes:
ii. Describing System Workflows:
iii. Visualizing Use Case Scenarios:
iv. Modeling Algorithms and Logic:
v. Decision Making and Branching:
vi. Parallel Processes and Concurrency
vii. Error Handling and Exceptions:

g) List three kinds of actor.

Ans. In use case diagrams, actors represent entities that interact with the system. Here are three common kinds of actors:

i. Primary Actor:
ii. Secondary Actor:
iii. External System:

h) What is domain model?

Ans. Domain modeling in Object-Oriented Analysis and Design (OOAD) is the process of systematically identifying, analyzing, and representing the essential concepts, behaviors, and relationships within a specific problem domain. It involves translating real-world domain knowledge into software artifacts, such as classes, attributes, methods, and associations, to create a conceptual framework that accurately reflects the structure and dynamics of the domain.

i. A domain is a collection of related concepts, relationships, and workflows.
ii. A domain model is a package containing class and activity diagrams.
iii. In software engineering, a domain model is a conceptual model of the domain that incorporates both behavior and data
iv. Is not a description of software objects instead it is a visual representation of conceptual classes in a problem domain.

i) What is meant by low?

Ans. In Object-Oriented Analysis and Design (OOAD), the term "low" typically refers to low-level design or low-level details in the context of software development. This stage of the design process deals with the more detailed and concrete aspects of how a system is constructed, as opposed to the high-level design, which focuses on the overall architecture and structure.

j) What is component diagram?

Ans. A component diagram is used to break down a large object-oriented system into the smaller components, so as to make them more manageable. It models the physical view of a system such as executables, files, libraries, etc. that resides within the node.

It visualizes the relationships as well as the organization between the components present in the system. It helps in forming an executable system. A component is a single unit of the system, which is replaceable and executable. The implementation details of a component

are hidden, and it necessitates an interface to execute a function. It is like a black box whose behavior is explained by the provided and required interfaces.

2019(Make up)

a) List the relationships that can exist between use cases.
Ans. There can be 5 relationship types in a use case diagram.
   i. Association between actor and use case
   ii. Generalization of an actor
   iii. Extend between two use cases
   iv. Include between two use cases
   v. Generalization of a use case

b) What is the role of creator?
Ans.
In Object-Oriented Analysis and Design (OOAD), the creator is a design pattern role that is responsible for instantiating objects of a particular class. This role is typically defined in the context of the Creator design pattern, which helps in managing object creation in a way that promotes flexibility and encapsulation.
In OOAD, the creator role is pivotal in design patterns for managing object instantiation. It defines the class or object responsible for creating instances of other classes. This role helps encapsulate the creation logic, promoting a clean and modular design.
Who creates an Object? Or who should create a new instance of some class?

c) Why is high cohesion important for software?
Ans. High cohesion refers to the degree to which the elements within a module, class, or component are related to each other and work together to fulfill a single, well-defined purpose. In software engineering, high cohesion is crucial for several reasons:
   i. Importance of High Cohesion:
   ii. Improved Maintainability:
   iii. Enhanced Reusability:
   iv. Simplified Debugging and Testing:
   v. Better Understandability:
   vi. Reduced Coupling:
   vii. Improved Scalability:

Example:
Consider a class in a software system:
   i. High Cohesion: A class Invoice Processor that handles only the creation, validation, and formatting of invoices. All methods and attributes in this class are related to invoice processing.
   ii. Low Cohesion: A class Utilities that includes methods for diverse functionalities such as file handling, string manipulation, and invoice processing. This class has mixed responsibilities and lacks a clear focus.

In the high cohesion example, the Invoice Processor class is easier to understand, test, and maintain because it has a single, well-defined responsibility. In contrast, the Utilities class is harder to manage due to its broad range of unrelated functionalities.

Summary:

High cohesion is important for software because it leads to better organized, more understandable, and maintainable code. It enhances reusability, simplifies debugging and testing, and promotes a modular design that is adaptable and scalable.

High cohesion is essential in software design because it leads to a more maintainable, understandable, and reusable codebase. It simplifies testing, debugging, and system design by ensuring that each module or class has a well-defined, focused responsibility. By promoting high cohesion, developers can create more robust, organized, and manageable software systems.

d) Define conceptual class.

Ans.

e) What does role specify while assigning relationship between classes?

Ans. In object-oriented design, a role specifies the part or function that an object of a class plays within the context of a relationship with another class. When assigning relationships between classes, such as associations, the role defines the nature and purpose of the connection from the perspective of each class involved in the relationship.

f) Which UML diagram is used to describe functional requirement?

Ans. To describe functional requirements in UML, the use case diagram is typically used.

Purpose: The use case diagram helps in identifying and visualizing the functional requirements of a system by showing the interactions between users (actors) and the system itself.

Components:

Actors: Represent external entities (users, systems) that interact with the system.

Use Cases: Represent specific functionalities or services that the system provides to the actors.

Relationships: Show how actors interact with use cases, including associations, include, and extend relationships.

g) Which symbols are used to represent private and protected members in class diagram?

Ans. In a UML class diagram, symbols are used to represent the visibility of class members (attributes and methods). The visibility symbols for private and protected members are as follows:

1. **Private Members:**
   - **Symbol:** `-` (minus sign)
   - **Description:** Private members are only accessible within the class they are defined in. They are not accessible from outside the class or by subclasses.
   - **Example:**

```plaintext
- attributeName: Type
- methodName(): ReturnType
```

2. **Protected Members:**
   - **Symbol:** `#` (hash sign)
   - **Description:** Protected members are accessible within the class they are defined in and by subclasses. They are not accessible from outside the class.
   - **Example:**

```plaintext
# attributeName: Type
# methodName(): ReturnType
```

h) Differentiate between abstract class and concrete class.
Ans.

Abstract Class	Concrete Class
An abstract class is declared using abstract modifier.	A concrete class is not declared using abstract modifier.
An abstract class cannot be directly instantiated using the new keyword.	A concrete class can be directly instantiated using the new keyword.
An abstract class may or may not contain abstract methods.	A concrete class cannot contain an abstract method.
An abstract class cannot be declared as final.	A concrete class can be declared as final.
Implement an interface is possible by not providing implementations of all of the interface's methods. For this a child class is needed..	Easy implementation of all of the methods in the interface.

i) List the possible relationships between classes.
Ans. In UML class diagrams, several types of relationships can exist between classes. These relationships help in defining how classes interact with and depend on each other. Here are the primary relationships:

i. Association:
ii. Aggregation:
iii. Composition
iv. Inheritance (Generalization):
v. Realization:
vi. Association Class:

j) What is the task performed in elaboration phase?

Ans. The elaboration phase is a critical stage in the Unified Process and other iterative software development methodologies. Its primary focus is on refining and detailing the system's architecture and design based on the initial requirements gathered during the inception phase. Here are the key tasks performed in the elaboration phase:

Tasks in the Elaboration Phase:

Refine Requirements:
Develop the Architecture:
Define Detailed Design:
Plan for Implementation:
Risk Analysis and Management:
Create Prototypes:
Establish Test Plans:

Summary:

The elaboration phase is focused on solidifying the system's architecture, refining requirements, and preparing for implementation. It involves detailed design work, risk management, and planning to ensure that the project is well-prepared for the construction phase. This phase aims to address and mitigate uncertainties and lay a solid foundation for successful implementation.

2019

a) What do you mean by low coupling?
Ans. Coupling is a measure of how strongly one element is connected to another element.

Low coupling refers to a design principle in object-oriented programming where the dependencies between classes or components are minimized. In a system with low coupling, each class or component has little or no reliance on the internal workings of other classes, reducing the risk of changes in one class affecting others. This leads to a more modular, flexible, and maintainable system, where components can be easily modified, reused, or tested independently without causing a ripple effect across the system.

Low coupling enhances the robustness and adaptability of the software, making it easier to evolve over time.

b) What is 'includes' in use case?
Ans.

c) What are the strategies to find conceptual classes?
Ans. Finding conceptual classes is a key part of object-oriented analysis, and it involves identifying the fundamental entities and concepts that are central to the problem domain. Here are some strategies to help in finding and defining these conceptual classes:

i. Reuse of modify existing model
ii. Use a category list
iii. Identify noun phrase

d) When are Specification Conceptual Classes Required?
Ans. Specification Conceptual Classes are required when the system needs to handle complex, flexible, or evolving business rules and requirements. They enable a clean separation of concerns, ensuring that varying conditions or criteria are encapsulated in reusable, maintainable classes. This approach makes the system easier to extend, test, and modify while adhering to object-oriented design principles like the Open-Closed Principle and promoting flexibility in behavioral modeling.

e) Why an activity diagram is needed in behavioral modeling?
Ans. An activity diagram is needed in behavioral modeling to visually represent the flow of control and data within a system or process. It illustrates the sequence of activities, decisions, and interactions that occur during a particular scenario, use case, or process.

i. Draw the activity flow of a system–
ii. Describe the sequence from one activity to another–
iii. Describe the parallel, branched and concurrent flow of the system

An activity diagram is essential in behavioral modeling because it provides a clear and visual representation of the dynamic aspects of a system. It models the flow of control or data within a system, focusing on the sequence of activities or actions performed.

f) What is collaboration diagram?

Ans. A collaboration diagram, also known as a communication diagram, is an illustration of the relationships and interactions among software objects in the Unified Modeling Language (UML).

A collaboration diagram (also known as a communication diagram) is a type of interaction diagram in UML (Unified Modeling Language) that illustrates how objects interact with one another through message exchanges in the context of a particular use case or operation. It focuses on the structural organization of objects and their relationships, highlighting the sequence of messages passed between objects to achieve a specific functionality.

g) Define adapter design pattern.

Ans. Adapter is a Structural Design Pattern that allows incompatible interfaces between classes to work together without modifying their source code.

In object-oriented analysis and design (OOAD), the Adapter Design Pattern is a structural pattern that allows incompatible interfaces to work together. It acts as a bridge between two classes or components that otherwise could not interact due to differing interfaces. The adapter wraps an existing class or object, translating its interface into one that a client expects, enabling the client to work with the class without any changes.

h) List any five GRASP patterns.

Ans. GRASP (General Responsibility Assignment Software Patterns) provides guidelines for assigning responsibilities to classes and objects in object-oriented design. Here are five commonly used GRASP patterns:

i. Creator–
ii. Information
iii. Expert–
iv. Low Coupling–
v. High Cohesion–
vi. Controller–
vii. Indirection–
viii. Polymorphism–
ix. Protected Variations–
x. Pure Fabrication

i) Why high cohesive design is needed?

Ans. A highly cohesive design is important in software development because it ensures that a class or module has a well-defined, focused responsibility. This means that all the functionalities within the class are closely related to each other, which leads to several benefits:

i. Improved Maintainability: Since a class or module handles a single, well-defined task, it is easier to understand, update, and modify without affecting unrelated parts of the system.
ii. Enhanced Reusability: High cohesion makes classes or modules more reusable, as they focus on a specific responsibility that can be utilized in other parts of the system or in different projects.

iii. Simplified Testing: A cohesive design makes it easier to test individual components because each one has a distinct and limited scope, reducing the complexity of the tests.
iv. Reduced Complexity: A highly cohesive design breaks down the system into smaller, more manageable pieces, minimizing confusion and interdependencies within the system.
v. Increased Reliability: By isolating functionality within highly cohesive modules, there is less chance of unintended side effects, leading to more stable and reliable code.

In summary, high cohesion helps in building systems that are easier to manage, modify, and scale, promoting overall software quality.

Extra

Cohesion refers to the degree to which the elements within a class or module in software design are related and work together to achieve a single purpose. In a highly cohesive system, each class or module has a focused responsibility and all its components are strongly related to that purpose.

Types of Cohesion:

i. High Cohesion: Each module or class has a single, clear purpose, making it easier to understand, maintain, and reuse.
ii. Low Cohesion: A module or class performs multiple, unrelated tasks, which makes the code more complex and harder to manage.

High cohesion is desirable because it leads to cleaner, more efficient, and maintainable code, where each part of the system has a clear, singular responsibility.

j) What do you mean by iterative development method?

Ans. Iterative development is a software development methodology where the system is built incrementally through repeated cycles (iterations). Instead of delivering the entire system at once, smaller portions of the system are developed, tested, and improved through multiple iterations. Each iteration involves refining the system by gathering feedback and making adjustments to the design, functionality, or requirements.

Key Features:

i. Incremental Build: The system is developed piece by piece, with each iteration adding more features or refining existing ones.
ii. Continuous Feedback: After each iteration, feedback is collected from stakeholders to identify improvements or changes.
iii. Flexibility: Iterative development allows for adjustments to requirements or design based on feedback, ensuring that the final product meets user needs more effectively.
iv. Risk Mitigation: Risks are identified and addressed early in the process as each iteration provides opportunities to test and evaluate the system.

Benefits:

i. Better adaptability to changes in requirements.
ii. Early detection of issues and risks.

iii. Continuous improvement of the system over time.

Examples of iterative methods include Agile and Rational Unified Process (RUP).

2022

a) Why do we need interface?

Ans. An interface is crucial in object-oriented programming (OOP) for several reasons, as it helps design flexible, reusable, and maintainable systems. An interface defines a contract or blueprint for classes, specifying what methods a class must implement without dictating how these methods should be implemented. (Abstraction, Polymorphism, multiple inheritance, loose couplying)

b) What is the importance of Use-Case Generalization?

Ans. Use-case generalization helps in creating a more organized, efficient, and maintainable model by capturing and reusing common behaviors, which enhances the overall design and management of the system.

Use-Case Generalization is important in use-case modeling because it helps to simplify and organize complex systems by capturing common behaviors and interactions in a more abstract and reusable way. Here's why it is valuable:

i. Reduces Redundancy
ii. Enhances Reusability
iii. Improves Clarity
iv. Facilitates Maintenance
v. Supports Extensibility

c) List any two symbols and their task used in activity diagram.

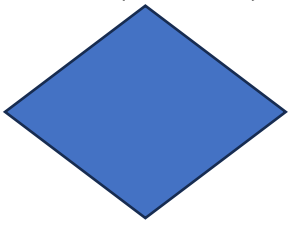
Ans. In an activity diagram, various symbols are used to represent different elements and tasks in a process. Here are two key symbols and their tasks:

Activity (Rounded Rectangle):

Symbol: A rounded rectangle.

Task: Represents a specific action or task that needs to be performed within the process. Each activity describes a single step or operation, such as "Process Order" or "Generate Report." Activities are the core elements of an activity diagram, showing the individual steps in the workflow.

Decision Node (Diamond):



Symbol: A diamond shape.

Task: Represents a point in the process where a decision is made, leading to different paths based on the outcome of the decision. For example, a decision node might determine whether an order is approved or rejected, directing the flow to different activities depending on the decision.

d) Why should we prepare deployment diagram?
Ans. A deployment diagram is crucial for several reasons:

i. Visualizes System Architecture.
ii. Supports Resource Planning
iii. Enhances Understanding of System Deployment
iv. Facilitates Troubleshooting
v. Supports Resource Planning
vi. Improves Scalability and Performance
vii. Guides Deployment Strategy

In summary, a deployment diagram is essential for understanding, planning, and managing the physical deployment of software systems, ensuring that all components are effectively distributed and integrated within the infrastructure.

A deployment diagram is prepared in software engineering to model the physical deployment of artifacts (software components) onto nodes (hardware elements) in a system's infrastructure. It shows how software interacts with hardware and how different components are deployed on physical machines.

e) Give an example of association and composition.
Ans.

f) What is an iteration?
Ans. An iteration refers to a single cycle of the iterative development process, where a specific portion of the system is developed, tested, and refined. In iterative development methodologies, the system is built incrementally through repeated iterations, each adding more functionality or making improvements based on feedback and evolving requirements. Iteration is the process of repeating a set of operations or steps. It is like doing something over and over again to make it better. The essence of iteration is cyclical in nature, where each successive repetition (or iteration) is intended to bring one step closer to the final goal or to enhance the outcome of ongoing process.

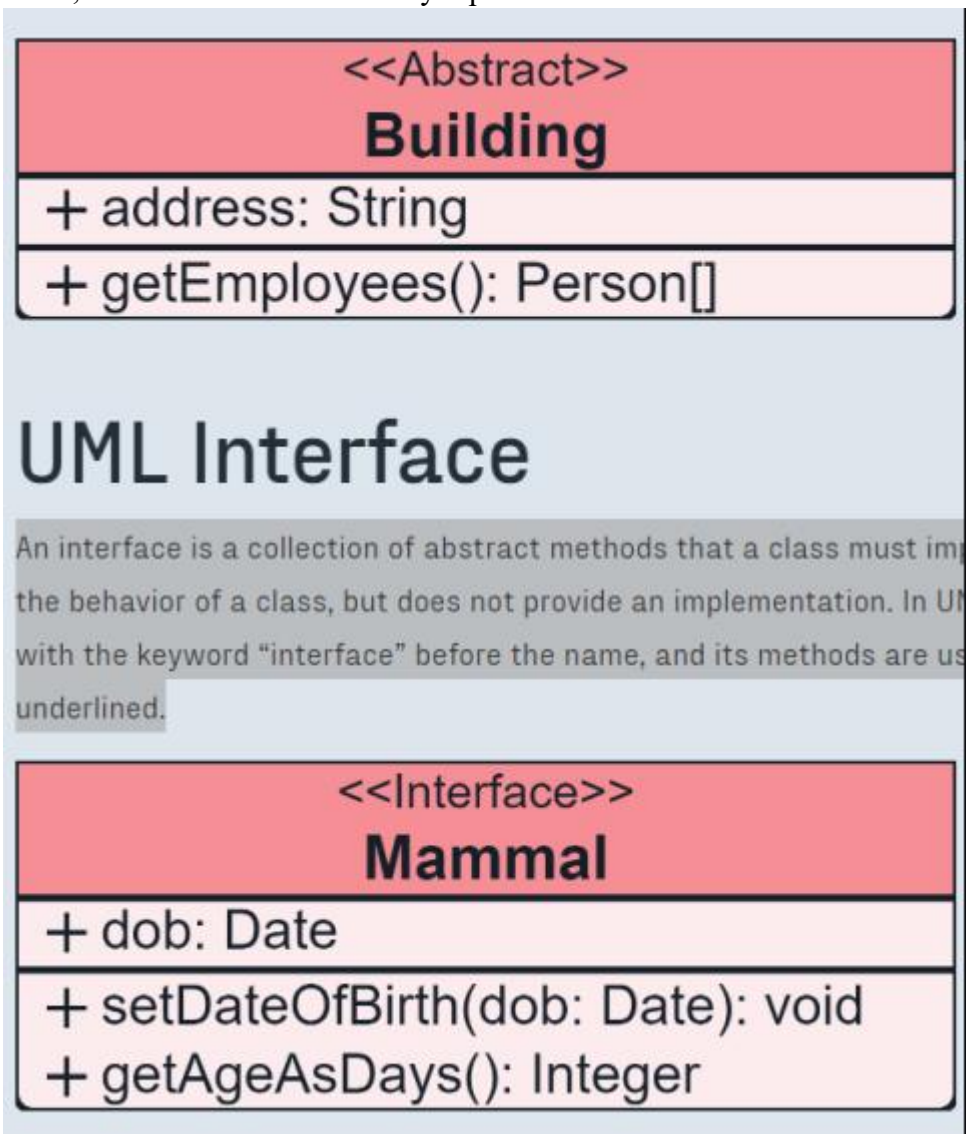
g) How can we achieve high cohesion?
Ans. High cohesion ensures that a class or module has a well-defined responsibility and focuses on a single task. By following principles like SRP, grouping related functionality, minimizing dependencies, and applying the Law of Demeter, you can create classes that are easier to maintain, test, and extend. This leads to a more modular, maintainable, and flexible codebase, reducing complexity and enhancing the clarity of your system design.

i. Single Responsibility Principle (SRP):
ii. Group Related Functionality Together:
iii. Encapsulate Data and Behavior Together:
iv. Limit Class Responsibilities:

v. Follow Domain-Driven Design (DDD): Use Clear, Self-Descriptive Class and Method Names:
vi. Keep Related Functions Together in the Same Module:

h) How do we represent abstract class and interface in the descriptive class diagram?
Ans. In UML class diagrams, an abstract class is a class that cannot be instantiated and is typically used as a base class for other classes that will provide the implementation. It is represented by a class with the keyword "abstract" before the name and its methods are usually represented as italicized and underlined.

An interface is a collection of abstract methods that a class must implement. An interface defines a contract for the behavior of a class, but does not provide an implementation. In UML, interfaces are represented by a rectangle with the keyword "interface" before the name, and its methods are usually represented as italicized and underlined.



i) Why does OOAD prefer low coupling?

Ans. Object-Oriented Analysis and Design (OOAD) prefers low coupling because it leads to several important benefits that enhance the overall quality and maintainability of the software system:

i. Modularity
ii. Flexibility
iii. Reusability
iv. Maintainability
v. Testability
vi. Scalability

In summary, low coupling in OOAD promotes a more modular, flexible, maintainable, and scalable system, improving the overall quality and adaptability of the software.

j) How can State Diagram be used for modeling dynamic behavior of the system?

Ans. A State Diagram is used for modeling the dynamic behavior of a system by visually representing how an object transitions between various states in response to events. It illustrates the lifecycle of an object, showing the different states it can be in, the events that trigger transitions between states, and the actions associated with each state. This helps in understanding how the system behaves over time, defining valid state sequences, and ensuring that all possible interactions and conditions are accounted for. By mapping out these dynamic aspects, state diagrams facilitate better design, testing, and validation of the system's behavior.

A state diagram (or state machine diagram) is a powerful tool used to model the dynamic behavior of a system by representing how an object or system transitions between different states in response to events or conditions. It helps in understanding how the system behaves over time, particularly when it responds to various inputs or actions.

2023

a) Why is UML regarded both as a tool and a language?

Ans. UML is both a language because it provides a standardized set of notations to represent systems, and a tool because it aids in the analysis, design, and documentation of software. It bridges the gap between design and implementation, making it an indispensable part of modern software engineering.

b) How do we represent abstract class and interface in a class diagram?

Ans.

c) Define description class.

Ans. A description class (also known as a descriptive class) in Object-Oriented Analysis and Design (OOAD) is a type of class that represents the metadata or information about other classes or objects. Unlike regular classes, which represent entities or objects with attributes and behaviors, a description class is used to capture and model the descriptive details or properties related to a specific entity. These details are typically external characteristics or attributes that help to describe or catalog objects but may not be part of the object's intrinsic behavior or identity.

A description class in OOAD is a class that primarily represents and stores the attributes and data of an entity, focusing on its characteristics rather than its behavior or interactions.

d) List the components of use case diagram.

Ans.

i. Actors
ii. Use cases
iii. System Boundary
iv. Relationship

e) Why should inception phase be of short duration?

Ans. The Inception phase should be of short duration to quickly define the project's scope, objectives, and feasibility without over-committing to specific solutions. This brevity helps in identifying and mitigating major risks early, preventing analysis paralysis, and ensuring that the project moves forward efficiently. By focusing on high-level requirements and early stakeholder alignment, a short Inception phase allows the team to swiftly transition into the iterative development process, maximizing resource efficiency and minimizing delays.

f) Define UML state diagram.

Ans. A UML state diagram, also known as a state machine diagram or state chart, is a type of behavioral diagram in the Unified Modeling Language (UML) that represents the dynamic behavior of an object by showing its states and the transitions between those states over time. It captures how an object responds to different events or stimuli by transitioning from one state to another, often depicting the life cycle of the object.

Key Components:

i. States: Represent the various conditions or situations in which an object can exist, typically shown as rounded rectangles.

ii. Transitions: Arrows connecting states, indicating the movement from one state to another in response to events or conditions.
iii. Events: Triggers that cause transitions between states.
iv. Initial State: Represented by a solid black circle, it indicates where the state diagram begins.
v. Final State: Represented by a circle with a dot inside, it shows the completion or termination of the state machine.

Example:
A state diagram might show how a "User Account" object moves from "Inactive" to "Active" when the user logs in, and then to "Suspended" if an account issue arises.

UML state diagrams are used to model the behavior of individual objects, capturing how they react to events, which is essential in understanding the system's dynamic aspects.

g) What are the input and output artifacts of elaboration phase?
Ans.
Inputs:
i. Vision Document
ii. Initial Use Case Model
iii. Project Charter
iv. Stakeholder Requirements
v. System Architecture
vi. Risk List
vii. Initial Design Artifacts

Outputs:
i. Refined Use Case Model
ii. Architecture Description
iii. Detailed Requirements Specification
iv. Risk Mitigation Plan
v. Project Plan and Iteration Plan
vi. Prototypes and Proof-of-Concepts
vii. Validated Architecture
viii. Updated Risk Register

h) How can we avoid a direct coupling between two or more elements?
Ans. To avoid direct coupling between two or more elements, we can use techniques like dependency injection, interfaces, and abstract classes, which allow classes to depend on abstractions rather than concrete implementations. Design patterns such as the observer, factory, and facade patterns also help in reducing direct dependencies. Additionally, following the Law of Demeter ensures that objects interact only with their direct collaborators, minimizing unnecessary dependencies. These strategies promote loose coupling, making the system more modular, flexible, and easier to maintain.

To avoid direct coupling between elements, we can use interfaces, abstract classes, dependency injection, observer patterns, design patterns, event-driven architectures, and service-oriented design. These approaches help in decoupling components, making the system more flexible, maintainable, and easier to modify or extend.

i) Define 100% rule.
   Ans.

j) What is domain model?
   Ans.

1. Inception Phase

Input Artifacts:

- Preliminary business case
- High-level system requirements
- Initial project scope

Output Artifacts:

- Preliminary conceptual model
- Requirements document (list of high-level use cases)
- Supplementary specifications
- Development schedule based on use case list

In the Inception Phase, the primary goal is to understand the system's extension and discover main requirements. The input artifacts provide a foundation for the project, while the output artifacts serve as a starting point for further development.

2. Elaboration Phase

Input Artifacts:

- Output from Inception Phase (conceptual model, requirements document, supplementary specifications, and development schedule)
- Risk analysis and prioritization

Output Artifacts:

- Detailed system architecture
- Iteration plans and schedules
- Executable architecture baseline
- Use cases with detailed descriptions and scenarios

The Elaboration Phase focuses on elaborating the system's architecture, identifying and addressing major risks, and planning iterations. The output artifacts provide a more detailed understanding of the system and serve as a foundation for the Construction Phase.

3. Construction Phase

Input Artifacts:

- Output from Elaboration Phase (architecture, iteration plans, executable architecture baseline, and use cases)

- Change requests and defect reports

Output Artifacts:

- Working software components
- Code for the whole application
- Implemented change requests
- Full-text use cases

In the Construction Phase, the primary goal is to produce working software components and implement change requests. The input artifacts provide the foundation for development, while the output artifacts demonstrate the system's functionality and usability.

4. Transition Phase

Input Artifacts:

- Output from Construction Phase (working software components, code, implemented change requests, and full-text use cases)
- Deployment plans and schedules

Output Artifacts:

- Deployed software system
- Post-deployment review and evaluation
- Lessons learned and improvement plans

The Transition Phase focuses on deploying the system, reviewing its performance, and identifying areas for improvement. The output artifacts document the system's deployment and provide insights for future development iterations.