NEC Registration Examination

Computer Engineering (ACtE)

Chapter 8: Software Engineering and Object-Oriented Analysis & Design (ACtE08)

Computer Engineering (ACtE)
Software Engineering
Information Technology

Complete MCQ-Focused Notes — Key Points • Diagrams • Easy Explanations • Practice MCQs

COMPLETE NOTES WITH SOLVED MCQ AND PYQS

EASY EXPLANATION



EASY EXPLANATION

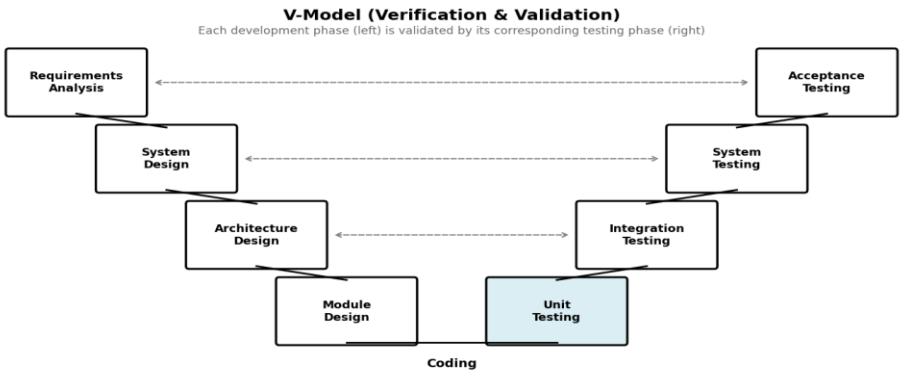
8.1 Software Process and Requirements

Software Characteristics & Quality Attributes

- Software is **developed/engineered, not manufactured**; it **doesn't wear out** (but does suffer from **deterioration/obsolescence** due to changing requirements); mostly **custom-built** rather than assembled from existing components.
- **Quality attributes:** `Correctness, Reliability, Efficiency, Usability, Maintainability, Portability, Reusability`.

Software Process Models

Model	Key Idea
Waterfall	Strictly sequential phases; simple but inflexible to change
V-Model	Each dev phase paired with a matching test phase (see diagram below)
Iterative Model	System built and refined in repeated iterations/cycles
Prototype Model	A working prototype built early to clarify unclear requirements
Agile Model	Short iterations (sprints), continuous customer feedback, embraces change
Big Bang Model	Minimal planning; resources thrown at development with no formal process — highly risky



Requirements

- **Functional requirements:** WHAT the system should **do** (specific behaviors/features).
- **Non-functional requirements:** HOW WELL the system performs — `performance, security, usability, reliability, scalability` (quality constraints, not features).
- **User requirements:** high-level statements in **natural language**, from the customer's perspective.
- **System requirements:** detailed, **technical** specification of functions/constraints — used by developers.
- **SRS (Software Requirements Document):** the formal document capturing all agreed requirements — serves as a **contract** between client and developer.
- **Requirements Engineering process:** `Elicitation` (gathering requirements from stakeholders) → `Analysis` (resolving conflicts/ambiguity) → `Specification` (documenting in the SRS) → `Validation` (confirming correctness with stakeholders) → `Management` (handling changes over time).

Easy Explanation

Quick distinction: Functional = 'what it does' (e.g. 'system shall allow login'); Non-functional = 'how well it does it' (e.g. 'login shall respond within 2 seconds'). Non-functional requirements are often just as important but easier to overlook.

? Probable MCQs: Software Process & Requirements

1. Which software process model pairs every development phase with a corresponding testing phase?

a) Waterfall Model
b) V-Model
c) Prototype Model
d) Big Bang Model
✓Answer: b) V-Model
Why: The V-Model explicitly mirrors each development stage with a matching verification/testing stage.

2. A requirement stating 'the system shall respond within 2 seconds' is an example of a:

a) Functional requirement
b) Non-functional requirement
c) User requirement only
d) Design constraint only
✓Answer: b) Non-functional requirement
Why: Performance/response-time constraints are non-functional requirements (quality attributes, not features).

3. The Agile model is best characterized by:

a) Long, rigid sequential phases
b) Short iterations with continuous customer feedback
c) No testing at all
d) A single final delivery only
✓Answer: b) Short iterations with continuous customer feedback
Why: Agile emphasizes short sprints, iterative delivery, and constant collaboration with the customer.

4. The SRS document primarily serves as a:

a) Source code repository
b) Contract between client and developer capturing requirements
c) Testing tool
d) Project management software
✓Answer: b) Contract between client and developer capturing requirements
Why: The Software Requirements Specification formally documents agreed requirements as a reference/contract.

5. Which stage of requirements engineering involves confirming requirements are correct with stakeholders?

a) Elicitation
b) Specification
c) Validation
d) Management
✓Answer: c) Validation
Why: Validation checks that documented requirements truly reflect what stakeholders need.

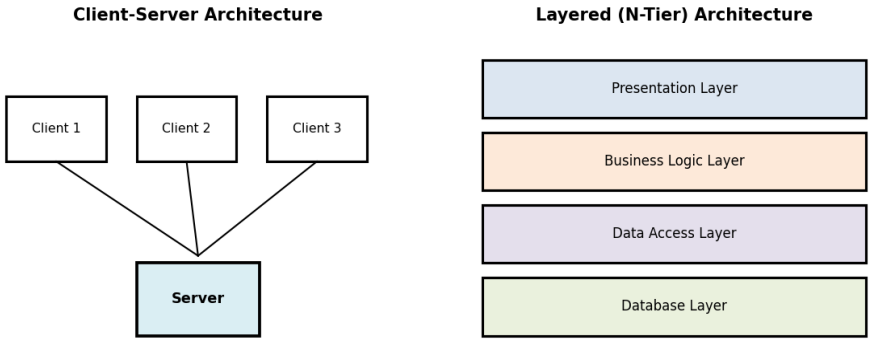
EASY EXPLANATION

8.2 Software Design

Design Concepts & Process

- **Design process:** transforms the **requirements model** (what) into a **design model** (how) — covering architecture, data, interface, and component-level design.
- **Design concepts:** `Abstraction, Modularity, Information Hiding, Cohesion (high is good), Coupling (low is good), Refinement`.
- **Design heuristics:** general rules of thumb — e.g. **keep modules small and focused, minimize coupling between modules, maximize cohesion within a module**.

Architectural Styles



Style	Description
Client-Server	Clients request services; a central server responds — simple, widely used
Layered (N-Tier)	System split into layers (Presentation, Business Logic, Data) — each layer only talks to the one below/above
Distributed Object	Objects spread across a network, communicate via middleware (e.g. CORBA, RMI)
Multiprocessor Architecture	Multiple processors share workload — for high-performance systems

- **Inter-organizational distributed computing:** systems spanning **multiple independent organizations** collaborating over a network (e.g. B2B systems, supply chain integration).
- **Real-time software design:** must meet **strict timing/deadline constraints** — often event-driven, prioritizes predictability over raw throughput.
- **Component-Based Software Engineering (CBSE):** builds systems by **assembling pre-built, reusable components** rather than writing everything from scratch — improves reuse and reduces development time.

> **Easy Explanation**
>
> *Design goal in one line: HIGH cohesion (a module does ONE well-defined job) + LOW coupling (modules depend on each other as little as possible) = easier to maintain, test, and reuse.*

Probable MCQs: Software Design

1. Which design principle refers to how strongly the elements WITHIN a single module are related?

a) Coupling
b) Cohesion
c) Abstraction

d) Modularity

✓Answer: b) Cohesion

Why: Cohesion measures the internal relatedness of a module's responsibilities — high cohesion is desirable.

2. Low coupling between modules is desirable because it:

a) Increases inter-module dependency

b) Makes modules easier to modify independently

c) Makes the system slower

d) Removes the need for testing

✓Answer: b) Makes modules easier to modify independently

Why: Low coupling means changes in one module are less likely to break others, improving maintainability.

3. In a layered (N-tier) architecture, which layer typically interacts directly with the database?

a) Presentation Layer

b) Business Logic Layer

c) Data Access Layer

d) None of these

✓Answer: c) Data Access Layer

Why: The Data Access Layer is responsible for direct interaction with the database/storage.

4. Component-Based Software Engineering (CBSE) emphasizes:

a) Writing all code from scratch every time

b) Assembling systems from pre-built reusable components

c) Avoiding all reuse

d) Manual testing only

✓Answer: b) Assembling systems from pre-built reusable components

Why: CBSE focuses on reusing existing, well-tested components to speed up development.

5. Real-time software design primarily prioritizes:

a) Maximum raw throughput regardless of timing

b) Meeting strict timing/deadline constraints

c) Minimizing lines of code

d) Avoiding all interrupts

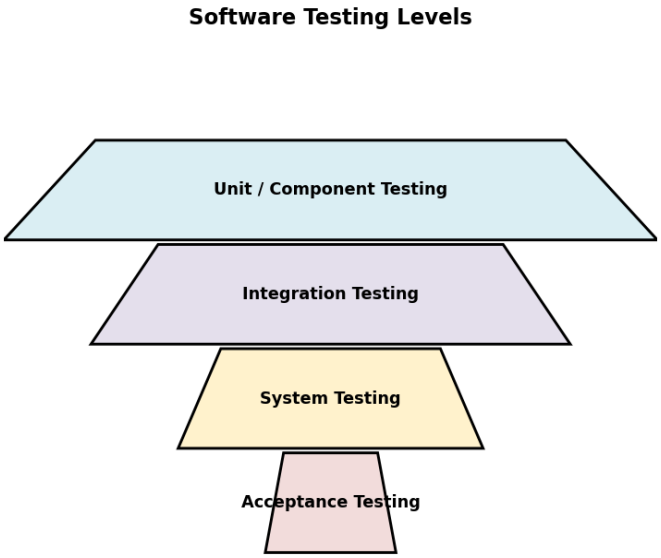
✓Answer: b) Meeting strict timing/deadline constraints

Why: Real-time systems must guarantee responses within specific deadlines, often more important than peak throughput.

EASY EXPLANATION

8.3 Testing, Cost Estimation, Quality & Configuration Management

Levels of Software Testing



Level	Focus
Unit/Component Testing	Tests a single function/module in isolation
Integration Testing	Tests interactions BETWEEN modules once combined
System Testing	Tests the COMPLETE, integrated system against requirements
Acceptance Testing	Validates the system meets business needs — often performed by the CLIENT

- **Test case design techniques:** `Black-box (equivalence partitioning, boundary value analysis)` vs `White-box (statement/branch/path coverage)`.
- **Test automation:** using tools to **automatically execute test cases** and compare actual vs expected results — saves time for regression testing.

Cost Estimation

COCOMO (Constructive Cost Model): Effort = a × (KLOC)^b person-months
(a, b are constants depending on project type: Organic, Semi-detached, Embedded)

- Project duration and staffing are derived from the estimated total effort — more people ≠ proportionally faster (see **Brooks's Law**: 'adding manpower to a late software project makes it later').

Software Quality Assurance (SQA)

- **SQA:** umbrella activity ensuring the **process AND product** meet defined quality standards — includes reviews, audits, and standards compliance.
- **Formal Technical Reviews (FTR):** structured **peer review** of a work product (design, code) to find defects early — cheaper to fix than after release.
- **Statistical SQA:** uses **defect data** (categorized by cause) to identify and prioritize the most common/costly sources of errors (similar to Pareto analysis).

Standard/Model	Purpose
ISO 9001	General quality management standard, applicable to software organizations

Standard/Model	Purpose
CMMI (Capability Maturity Model Integration)	Assesses & improves an organization's software process maturity (5 levels: Initial → Optimizing)

Software Configuration Management (SCM)

- **SCM:** manages **change** throughout the software lifecycle — tracks versions of code, documents, and other artifacts.
- **Configuration items:** anything placed under version control (source code, requirements docs, test plans).
- **Change management:** formal process to **request, evaluate, approve, and track** changes to a baseline.
- **Version control vs Release management:** Version control tracks **incremental changes** during development; release management coordinates **packaging and delivering** a stable version to users.
- **CASE tools for CM:** automate version control, baselining, and change tracking (e.g. Git, SVN).

Easy Explanation

CMMI's 5 maturity levels (memorize in order): 1-Initial (chaotic/ad-hoc) → 2-Managed → 3-Defined → 4-Quantitatively Managed → 5-Optimizing (continuous process improvement).

Probable MCQs: Testing, Cost Estimation & Quality

1. Which testing level is typically performed by the CLIENT to validate business needs?

a) Unit testing
b) Integration testing
c) System testing
d) Acceptance testing
✓Answer: d) Acceptance testing
Why: Acceptance testing confirms the system satisfies the client's actual business requirements.

2. COCOMO is a model used primarily for:

a) Bug tracking
b) Software cost/effort estimation
c) Version control
d) UML diagramming
✓Answer: b) Software cost/effort estimation
Why: COCOMO estimates development effort (person-months) based on project size (KLOC) and type.

3. Brooks's Law states that:

a) More people always finish a project faster
b) Adding manpower to a late project makes it later
c) Testing should be skipped to save time
d) Requirements never change
✓Answer: b) Adding manpower to a late project makes it later
Why: Brooks's Law highlights the overhead (communication, training) of adding people to an already-delayed project.

4. CMMI Level 1 (Initial) is characterized by:

a) Fully optimized, continuously improving processes
b) Ad-hoc, chaotic processes with unpredictable outcomes
c) Quantitatively managed processes
d) Standardized, defined processes org-wide

EASY EXPLANATION

✓**Answer: b) Ad-hoc, chaotic processes with unpredictable outcomes**

Why: CMMI Level 1 represents an immature, unpredictable process — the starting point before improvement.

5. Software Configuration Management is primarily concerned with:

a) Writing new features

b) Managing and controlling change to software artifacts

c) UI design

d) Marketing the software

✓**Answer: b) Managing and controlling change to software artifacts**

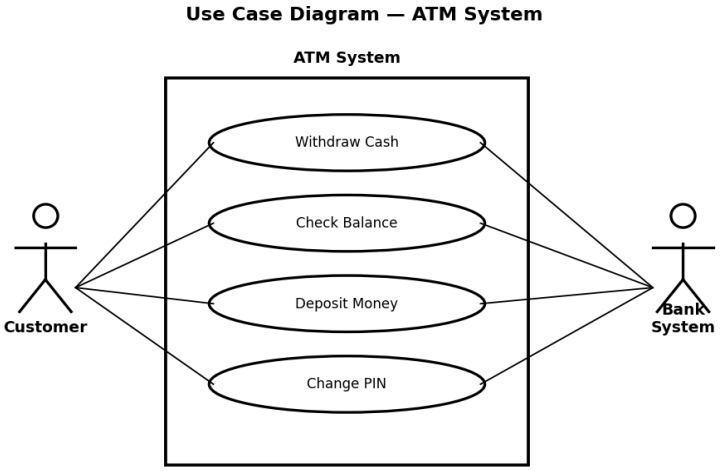
Why: SCM tracks and controls changes to code, documents, and other configuration items over the project lifecycle.

EASY EXPLANATION

8.4 Object-Oriented Fundamentals and Analysis

OO Development Cycle & Use Cases

- **OO Development Cycle:** `Requirements (Use Cases) → OO Analysis (conceptual model) → OO Design (class/sequence diagrams) → OO Implementation (code) → Testing`.
- **Use Case:** describes a **specific interaction** between an **Actor** (user/external system) and the system, to achieve a goal.



- **Actor:** anyone/anything **external** to the system that interacts with it (a person, another system, or hardware device).

UML (Unified Modeling Language)

Diagram Type	Purpose
Use Case Diagram	Shows WHAT the system does, from the user's perspective
Class Diagram	Shows the STATIC structure — classes, attributes, methods, relationships
Sequence Diagram	Shows message flow between objects OVER TIME
Collaboration Diagram	Shows the same interactions as sequence diagram, but focused on OBJECT LINKS/structure
Activity Diagram	Shows workflow/business process logic (like a flowchart)
State Diagram	Shows the different STATES an object can be in and transitions between them

Building the Conceptual Model

- **Conceptual model:** identifies real-world **concepts/classes** and their **associations** and **attributes** — built early in analysis, independent of implementation details.
- **Association:** a **structural relationship** between two classes (e.g. Customer 'owns' Account) — described with **multiplicity** (1, 0..1, 1..*, *, etc.).
- **Representation of system behavior:** done via **use cases (functional view)** combined with **sequence/state diagrams (dynamic view)**.

> **Easy Explanation**
>
> *Multiplicity notation cheat sheet: '1' = exactly one, '0..1' = zero or one (optional), '*' or '0..*' = zero or many, '1..*' = one or many (at least one required).*

Probable MCQs: OO Fundamentals & Analysis

1. A Use Case primarily describes:

a) The internal code structure

b) An interaction between an actor and the system to achieve a goal

c) The database schema

d) The physical hardware layout

✓Answer: b) An interaction between an actor and the system to achieve a goal

Why: Use cases capture functional interactions from the perspective of external actors.

2. Which UML diagram shows the static structure of classes and their relationships?

a) Sequence diagram

b) Class diagram

c) Activity diagram

d) Use case diagram

✓Answer: b) Class diagram

Why: Class diagrams show classes, attributes, methods, and relationships — the static view of the system.

3. In UML, the multiplicity '1..*' means:

a) Exactly zero

b) Zero or one

c) One or many (at least one)

d) Exactly one only

✓Answer: c) One or many (at least one)

Why: '1..' indicates at least one instance is required, with no upper limit.*

4. A Sequence diagram is primarily used to show:

a) Class inheritance only

b) Message flow between objects over time

c) Database tables

d) Hardware wiring

✓Answer: b) Message flow between objects over time

Why: Sequence diagrams capture the time-ordered exchange of messages between objects.

5. An Actor in a use case diagram represents:

a) A class attribute

b) Anything external that interacts with the system

c) A database table

d) A design pattern

✓Answer: b) Anything external that interacts with the system

Why: Actors are external entities (users, systems, devices) that interact with the system under design.

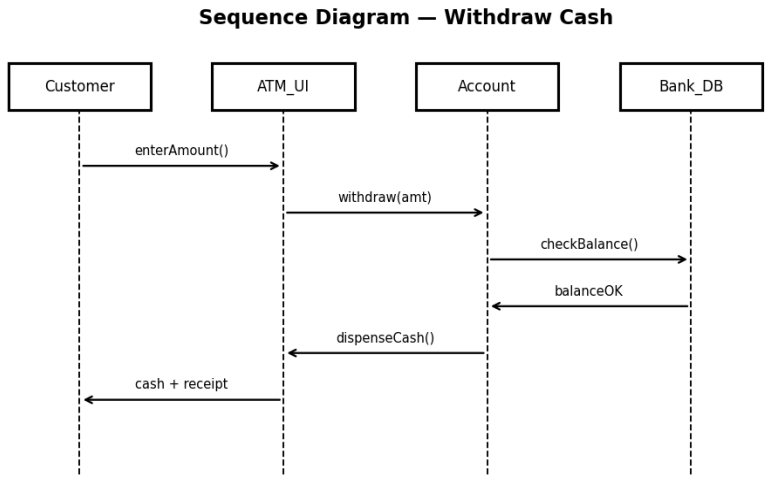
EASY EXPLANATION

8.5 Object-Oriented Design

From Analysis to Design

- OO Design **elaborates** the conceptual model from analysis into a **detailed, implementable design** — adds design-specific classes (e.g. UI classes, database access classes) not present in the pure conceptual model.
- **Elaborating use cases:** expanding a brief use case description into detailed **step-by-step interaction flows** (main success scenario + alternative/exception flows).

Collaboration Diagrams

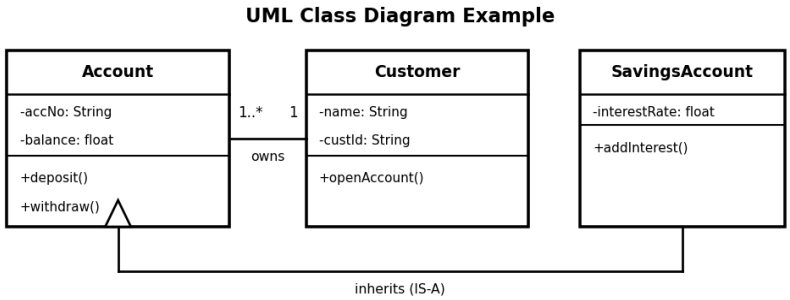


- (Shown above as a Sequence Diagram — a Collaboration Diagram conveys the **SAME message exchanges**, but is drawn around **object links** rather than a vertical timeline; both are '**interaction diagrams**'.)

Objects, Patterns & Visibility

- **Design patterns:** reusable, proven solutions to **recurring design problems** — e.g. `Singleton, Factory, Observer, Strategy, MVC`.
- **Determining visibility:** deciding **which classes need references to which other classes** to fulfill their responsibilities/collaborations (this determines the attributes/associations needed in the class diagram).

Class Diagram (Design-Level)



Association: Customer 'owns' 1..* Account(s)
Inheritance: SavingsAccount IS-A Account (hollow triangle points to the parent class)

- At design time, a class diagram is refined to show **visibility markers** — `+ public, - private, # protected` — along with full attribute types and method signatures, ready for coding.

> **Easy Explanation**
> *Visibility symbol cheat sheet: '+' = public (accessible from anywhere), '−' = private (only within the same class), '#' = protected (class + its subclasses) — these map directly to access specifiers in C++/Java.*

Probable MCQs: Object-Oriented Design

1. OO Design differs from OO Analysis mainly because design:

a) Ignores the conceptual model entirely
b) Adds implementation-specific classes and detail to the conceptual model
c) Is done before requirements gathering
d) Never uses UML
✓Answer: b) Adds implementation-specific classes and detail to the conceptual model
Why: Design elaborates the analysis model with concrete, implementable detail (UI classes, DB access, etc.).

2. A Collaboration diagram differs from a Sequence diagram mainly in that it:

a) Shows completely different information
b) Emphasizes object links/structure rather than a timeline
c) Cannot show messages at all
d) Is used only for database design
✓Answer: b) Emphasizes object links/structure rather than a timeline
Why: Both are interaction diagrams showing the same messages; collaboration diagrams organize this around object relationships instead of time.

3. The '+' symbol before a class attribute/method in UML indicates:

a) Private visibility
b) Protected visibility
c) Public visibility
d) Static visibility
✓Answer: c) Public visibility
Why: '+' denotes public visibility — accessible from any other class.

4. A design pattern is best described as:

EASY EXPLANATION

a) A specific piece of code copied everywhere

b) A reusable solution to a recurring design problem

c) A type of database schema

d) A testing framework

✓Answer: b) A reusable solution to a recurring design problem

Why: Design patterns are general, proven templates for solving common design challenges (e.g. Singleton, Observer).

5. 'Determining visibility' between classes in OO design refers to deciding:

a) Which classes are visible on screen

b) Which classes need references to collaborate with each other

c) The color scheme of the UI

d) Database table visibility

✓Answer: b) Which classes need references to collaborate with each other

Why: Visibility here means which objects need to 'know about' (hold a reference to) other objects to fulfill their responsibilities.

EASY EXPLANATION

8.6 Object-Oriented Design Implementation

Mapping Design to Code

- **Programming and development process:** translates the **design model (class diagrams, collaboration diagrams)** into actual **source code** in a chosen OO language (Java, C++, Python, etc.).
- **Creating class definitions from Class Diagrams:** each UML class becomes a **class declaration**; attributes become **member variables**; associations become **object references** (as fields).
- **Creating methods from Collaboration Diagrams:** each **message** sent between objects in the interaction diagram becomes a **method call**; the sequence of messages defines the **method body's logic**.

From UML to Code — Example Mapping

UML Element	Code Equivalent
Class	class ClassName { ... }
Attribute (− type)	private DataType attributeName;
Method (+ signature)	public ReturnType methodName(params) { ... }
Association (1..*)	A collection/list field (e.g. List<Account> accounts;)
Inheritance (▷hollow triangle)	class Child extends Parent { ... } (or : public Parent in C++)
Message in sequence diagram	A method call: object.method(args);

- **Updating class definitions:** as design evolves (e.g. new associations discovered), class code is **iteratively refined** to stay consistent with the (possibly updated) design model.

Exception and Error Handling

- Implementation must **account for error conditions** not always obvious in the design model — e.g. invalid input, failed network calls, resource unavailability.
- Standard OO approach: `try / catch / throw` (or `try/except` in Python) — keeps normal logic separate from error-handling logic, improving readability and robustness.

> **Easy Explanation**
>
> *A simple rule of thumb when moving from design to code: EVERY class in the class diagram → a class in code; EVERY method signature (with visibility +/−/#) → a method with matching access modifier; EVERY message arrow in a sequence/collaboration diagram → an actual method call in the code.*

Probable MCQs: OO Implementation

1. In mapping UML to code, an attribute marked with '−' (private) in the class diagram becomes:

a) A public field
b) A private member variable
c) A separate class
d) A comment only
✓Answer: b) A private member variable
Why: The '−' visibility marker directly maps to a private access modifier in the implementation.

2. Messages shown in a sequence/collaboration diagram typically become, in code, a:

a) New class definition
b) Method call between objects
c) Database table
d) Compiler directive
✓Answer: b) Method call between objects

EASY EXPLANATION

Why: Each message exchanged between objects in the interaction diagram corresponds to a method invocation in the implementation.

3. An association with multiplicity '1..*' in a class diagram is typically implemented in code as:

a) A single primitive variable

b) A collection/list of references

c) A boolean flag

d) A static constant

✓Answer: b) A collection/list of references

Why: A 'one-to-many' association needs a collection (list/array) to hold multiple related object references.

4. Separating normal program logic from error-handling logic in OO code is typically done using:

a) Global variables

b) try/catch/throw constructs

c) Inline comments only

d) Preprocessor macros only

✓Answer: b) try/catch/throw constructs

Why: Structured exception handling (try/catch/throw) cleanly separates error handling from the main logic flow.

5. Inheritance shown with a hollow triangle arrow in a UML class diagram is implemented in Java using the keyword:

a) implements

b) extends

c) inherits

d) uses

✓Answer: b) extends

Why: Java uses 'extends' for class inheritance (implements is used for interfaces).

EASY EXPLANATION