Object-Oriented Analysis & Design Notes:
Principles, Patterns, and Practice

Zack Polaski

Part I
Reference Table of GOF Design Patterns

Category	Pattern	Intent (One–liner)	Typical Use Cases
Creational	Factory Method	Define an interface for creating an object, letting subclasses decide the concrete class.	UI widgets per platform; loggers; storage drivers
Creational	Abstract Factory	Create families of related objects without specifying their concrete classes.	Cross-platform GUI kits; theming; DB provider stacks
Creational	Builder	Separate construction of a complex object from its representation.	Document/report generators; test data builders; query builders
Creational	Prototype	Create objects by cloning a prototypical instance.	Game entities; graphical shapes; preconfigured templates
Creational	Singleton	Ensure a class has one instance and provide a global access point.	Config registries; caches; connection pools; schedulers
Structural	Adapter	Convert one interface into another clients expect.	Legacy system integration; API bridges; driver shims
Structural	Bridge	Decouple abstraction from implementation so both can vary independently.	Rendering backends; device drivers; cross-platform libraries
Structural	Composite	Treat part–whole hierarchies uniformly (objects and compositions the same).	GUI component trees; file systems; scene graphs
Structural	Decorator	Attach responsibilities to objects dynamically without subclassing.	I/O streams; middleware; auth/logging layers
Structural	Facade	Provide a unified, simplified interface to a complex subsystem.	SDK entrypoints; subsystem gateways; init/orchestration
Structural	Flyweight	Share intrinsic state across many fine-grained objects to save memory.	Text glyphs; sprite sharing; caching immutable data
Structural	Proxy	Provide a surrogate to control access to a real subject.	Lazy loading; virtual proxies; protection proxies; remotes
Behavioral	Chain of Responsibility	Pass a request along handlers until one processes it.	Web middleware; event pipelines; auth chains
Behavioral	Command	Encapsulate a request as an object (supports queueing, logging, undo).	GUI actions; job queues; macro/undo systems
Behavioral	Interpreter	Define a grammar and interpret expressions built from it.	DSLs; rule engines; expression evaluators; filters
Behavioral	Iterator	Provide a standard way to traverse elements without exposing representation.	Collection iteration; cursors; tree traversals

Category	Pattern	Intent (One–liner)	Typical Use Cases
Behavioral	Mediator	Centralize complex communication among collaborators to reduce coupling.	Chat rooms; GUI dialogs; message hubs/brokers
Behavioral	Memento	Capture and restore an object's internal state without breaking encapsulation.	Undo/redo; checkpoints; state snapshots
Behavioral	Observer	Define one-to-many dependency; observers auto-update on subject changes.	Event buses; GUIs; reactive UIs; pub/sub
Behavioral	State	Let an object change behavior when its internal state changes.	Vending machines; protocol states; game AI
Behavioral	Strategy	Define a family of algorithms, encapsulate, and make them interchangeable.	Pluggable policies; sorting; pricing/tax rules
Behavioral	Template Method	Define algorithm skeleton; defer varying steps to subclasses.	Test runners; ETL pipelines; request lifecycles
Behavioral	Visitor	Add new operations to structured objects without modifying their classes.	AST passes; reporting; document renderers

Part II
Principles

1 Software Design Perspectives

1.1 Learning Objectives

- Contrast functional decomposition with object-oriented analysis and design (OOAD).
- Understand why multiple design perspectives beyond implementation matter.
- Define and review the core OOAD concepts:
  - Coupling
  - Cohesion
  - Abstraction
  - Encapsulation
  - Modularity

1.2 Problem Analysis Approaches

- Humans naturally solve problems by devising a **plan**.
- For computational tasks, this often results in **functional decomposition**: breaking a problem into solvable steps.
- Example: displaying shapes
  1. Connect to database.
  2. Locate and retrieve shapes.
  3. Sort shapes (e.g., z-order).
  4. Loop through list and display each.
  5. Identify type of shape (circle, square, triangle).
  6. Compute location and draw.

1.3 Functional Decomposition: Strengths and Weaknesses

Strengths

- Natural approach — humans and first-time programmers gravitate to it.
- Effective for small, well-defined problems.

Weaknesses

- Designs center around a controlling "main program."
- Poor modularity — changes ripple through the system.
- Difficult to extend abstractions.
- Problems often due to:
  - Poor abstraction
  - Weak encapsulation (lack of information hiding)
  - Weak modularity

1.4 Key OOAD Concepts

Abstraction

Identifying essential concepts that support problem solving.

- Example: public methods of Java's `String` class.

Encapsulation

Mechanisms that control access to implementation details.

- Example: private instance variables with getters/setters.

Modularity
Grouping related elements with clear interaction boundaries.

Cohesion
"How closely related are the operations?"

- We want methods, classes, and subsystems to do one thing well.

Coupling
"How strongly connected are two routines or modules?"

- Goal: loose coupling, strong cohesion.

1.5 Breakdown of Functional Decomposition

- Lack of modularity → weak cohesion and tight coupling.
- Example (bad design):

```
void process_records(records: record_list) {
 sort records;
 update values;
 print records;
 archive records;
 log operations;
}
```

1.6 Object-Oriented Design Approach

- Systems become **networks of objects** collaborating.
- Objects = { data (attributes) + behavior (methods) }.
- Each object:
  - Knows its type.
  - Maintains its own state.
  - Performs tasks through methods.

1.7 Responsibilities of an Object

- Think of an object as "something with responsibilities."
- Responsibilities guide discovery of required objects.
- Domain concepts often suggest candidate objects.

1.8 Design Perspectives

- **Conceptual**: a set of responsibilities.
- **Specification**: a set of methods.
- **Implementation**: code and data.
- OOAD is often taught at implementation only, but broader perspectives bring major benefits.

1.9 Benefits of Conceptual Thinking

- Encourages broad exploration in early design stages.
- Prevents premature optimization.
- Helps define mental models and system responsibilities.

1.10 Conceptual Design as Abstraction

- Abstraction answers:
  1. How will users think about using it?
  2. What mental model should be conveyed?
  3. What responsibilities and connections exist?
- Encapsulation supports abstraction, but abstraction is broader (generalization, interfaces, responsibility assignment).

1.11 Transitioning from Functional to OO Paradigm

Scenario: Conference Navigation

- **Functional decomposition:** instructor manages every detail (find sessions, compute routes, instruct each attendee).
- **Object-oriented:** each attendee has knowledge of their next session and navigates independently.

Key Insight

- OO shifts responsibility from a central program to self-sufficient objects.
- Easier to extend and handle variation.

1.12 Where We're Heading

- Entities = objects, self-sufficient.
- General instructions → **code to an interface**.
- Flexibility → **polymorphism, subclasses**.
- Goal: designs with strong cohesion, loose coupling, robust abstractions.

1.13 Summary

- Functional decomposition is natural but fragile.
- OOAD emphasizes conceptual design, responsibility assignment, and modularity.
- Our guiding principles: **abstraction, encapsulation, cohesion, coupling, modularity**.

2 Programming Language Paradigms

2.1 Learning Objectives

- Define a programming paradigm.
- Consider alternatives to the object-oriented (OO) paradigm.
- Understand why OO and the OOAD process will be our primary focus.

2.2 What is a Paradigm?

- In general usage: a philosophical or theoretical framework for theories, laws, and patterns of thought.
- In programming: a style of structuring and reasoning about programs, supported by languages, libraries, and practices.
- Examples: object-oriented, procedural, functional, declarative.

2.3 Programming Paradigms Overview

- This course emphasizes OO, but other paradigms exist:
  - **Imperative**: procedural, OO languages.
  - **Declarative**: logic, functional, and specification languages.

2.4 Taxonomy of Languages

- **Object-Oriented**: C#, Java, C++, Ruby.
- **Procedural**: Fortran, C.
- **Functional**: Lisp, Haskell, Clojure, F#.
- **Composites**: Python, Scala.
- **Logic**: Prolog, Datalog, ASP.
- **Graphical/Visual**: Scratch, LabView.
- **Domain-Specific**: LaTeX, R, Matlab, SQL.
- **Specifications**: HTML, CSS, SQL DDL.

2.5 Procedural Language Characteristics

- Examples: Fortran, C, and certain uses of Python.
- Core features:
  - Procedure/function calls for modularity.
  - Top-down design and sequential execution.
  - Scoped variables (global, local).
  - Emphasis on data with limited abstraction.
- Often used for low-level system access (e.g., firmware) or scientific applications.

2.6 Functional Language Characteristics

- Examples: Lisp, Haskell, Clojure, F#, elements of Scala.
- Core features:
  - First-class and higher-order functions.
  - Immutable variables, pure functions (no side effects).
  - Recursion and referential transparency.
  - Lazy evaluation and pattern matching.
  - Strong concurrency and parallelism support.
- Often used in data science, distributed systems, and communications.

2.7 Logic-Based Language Characteristics

- Example: Prolog.
- Core features:
  - Declarative rule-based programming.
  - Predicate logic, unification, and inference.
  - Backtracking and searching.
- Often used in AI, natural language processing, and expert systems.

2.8 Domain-Specific Languages (DSLs)

- Examples: LaTeX, R, Matlab, SQL.
- Designed for a specific class of problems rather than general-purpose use.
- **Internal DSLs**: embedded within a host language (fluent interfaces).
- **External DSLs**: custom syntax with their own parser (or encoded in XML, YAML).

2.9 Graphical or Visual Programming

- Example: Scratch.
- Promotes computational thinking, problem solving, creativity, and collaboration.
- Widely used in education.

2.10 Visual Programming and UML

- Early OOAD methods in the 80s/90s: OMT (Rumbaugh), OOSE (Jacobson), Booch method.
- Unified in the mid-90s by Rational: **UML (Unified Modeling Language)**.
- Current standard: UML 2.5.1 (2017).

2.11 Challenges of Visual Programming

- Advantages: directness, immediacy, and simplicity for small or educational tasks.
- Limitations:
  - Syntax and readability issues.
  - Navigating multiple levels of detail.
  - Extensibility challenges.
  - Poor scalability for complex systems.

2.12 The OOAD Paradigm: Our Focus

- Designing systems as **networks of collaborating objects**.
- Objects:
  - Combine state (attributes) and behavior (methods).
  - Maintain identity and type clarity.
  - Interact through class-defined relationships.
- Widely used in industry — most popular languages (Java, Python, C#, etc.) are OO or partially OO.

2.13 A Look Ahead: OO Concepts

- Abstraction.
- Encapsulation, information hiding, accessibility.
- Inheritance and delegation.
- Polymorphism (override, overload).
- Modularity, coupling, cohesion.
- Classes, objects, interfaces, generics.
- Aggregation, composition, UML modeling.

2.14 A Look Ahead: OO Design Principles

General Guidelines

- Program to interfaces, not implementations.
- Encapsulate what varies.
- A class should have only one reason to change.
- Favor delegation over inheritance.
- Strive for loosely coupled designs.
- Law of Demeter (Principle of Least Knowledge).
- Hollywood Principle: “Don’t call us, we’ll call you.”

SOLID Principles

- Single Responsibility Principle (SRP).
- Open/Closed Principle.
- Liskov Substitution Principle (LSP).
- Interface Segregation Principle (ISP).
- Dependency Inversion Principle (DIP).

Other Principles

- DRY (Don't Repeat Yourself).
- YAGNI (You Aren't Going to Need It).
- Principle of Healthy Skepticism.

2.15 A Look Ahead: OO Design Patterns

- Standard, reusable solutions for common design problems.
- From the "Gang of Four" catalog:
  - Creational: Builder, Factory, Abstract Factory, Prototype, Singleton.
  - Structural: Adapter, Decorator, Proxy, Bridge, Composite, Flyweight, Façade.
  - Behavioral: Strategy, Observer, Iterator, Command, State, Template, Visitor, Chain of Responsibility, Mediator, Memento, Interpreter.

2.16 Summary

- OOAD reflects the OO paradigm, but alternatives exist (procedural, functional, logic, DSLs, visual).
- OOAD is not the only approach but remains the industry standard.
- Our goal: learn OOAD effectively by grounding it in principles, guidelines, and patterns.

3 Object-Oriented Analysis and Design Concepts

3.1 Learning Objectives

- Review and define the foundational concepts of object-oriented systems.
- Understand objects, classes, inheritance, encapsulation, and accessibility.
- Examine object lifecycle: construction, destruction, and resource management.
- Explore polymorphism, abstract vs. concrete classes.
- Assess the role of delegation, association, aggregation, and composition.
- Distinguish override vs. overload and the principle of Design by Contract.
- Study abstract classes, interfaces, and generic components.
- Understand the notions of object identity and equality in OO design.

3.2 Objects and Classes

Objects as Fundamental Units.

- An object combines both **data** (attributes, properties, fields) and **behavior** (methods).
- Objects track their state through attributes and act on that state through methods.
- Objects inherently know their type and interact accordingly.

Classes as Blueprints.

- Objects are **instances of classes**.
- Classes define:
  - The complete behavior of their instances.
  - Data members (attributes) and method definitions.
  - Accessibility (public, private, protected).
- Example: Two `Student` objects may have different attribute values but share the same set of methods, since both are defined by the `Student` class.

3.3 Inheritance and Type Hierarchies

Subclasses and Superclasses.

- Classes can be arranged in hierarchies, where a **subclass** inherits behavior and data from its **superclass**.
- Subclasses:
  - Add new behaviors and attributes.
  - Modify or override inherited behaviors.
- Relationships are often described as **IS-A**:
  - Undergraduate IS-A Student.
  - Natural numbers IS-A subset of integers.

Benefits of Superclasses.

- Collections of mixed subclasses can be treated as collections of the superclass type.
- Example: `Undergraduate`, `MastersStudent`, and `PhDStudent` objects can all be stored in a single list of `Student`.
- Methods defined in the superclass apply uniformly across subclasses.

3.4 Encapsulation and Accessibility

Encapsulation.

- Encapsulation hides implementation details and restricts access to class internals.
- Controlled through access modifiers:
  - **Public**: visible everywhere.
  - **Protected**: visible to the class and its subclasses.
  - **Private**: visible only to the defining class.

3.5 Constructors and Destructors

- **Constructors** ensure objects are properly initialized.
- **Destructors** (or finalizers) ensure resources are released when objects are destroyed.
- In garbage-collected languages, destructors may not execute immediately after an object is no longer used.

3.6 Polymorphism

Definition.

- **Polymorphism** means "many forms."
- It allows objects of different subclasses to be treated as instances of their superclass while still invoking behavior appropriate to their specific type.

Practical Implications.

- Code written for the superclass can seamlessly operate on subclasses.
- New subclasses can be added without changing existing code.
- Example:

```
for (Student s : students) {
 s.saySomething();
}
```

  Each subclass overrides `saySomething()`, and the loop adapts automatically.

3.7 Abstract and Concrete Classes

- **Abstract classes**: define generic behavior and data but cannot be instantiated directly.
- **Concrete classes**: provide full implementations and can be instantiated.
- Abstract classes:
  - Define method signatures (contracts).
  - May provide shared data or implemented methods.

3.8 Delegation and Composition

Delegation.

- A class may delegate responsibility to another class.
- Relationship: **HAS-A**.
- Example: A `Car` HAS-A `Engine`.
- Benefits:
  - Better abstraction and separation of concerns.
  - Less code duplication.
  - Flexibility: delegation can be changed at run-time (unlike inheritance).

Association, Aggregation, and Composition.

- **Association**: one class references another.
- **Aggregation**: whole-part relationship where parts may exist independently.
- **Composition**: whole-part relationship with dependency (parts cannot exist independently).

3.9 Inheritance: Best Practices and Pitfalls

- Proper IS-A relationships should reflect domain logic.
- Misuse (e.g., Dog IS-A Window) leads to poor designs.
- Inheritance provides:
  - Code reuse (shared functionality across subclasses).
  - Polymorphism (substituting subclasses for superclasses).

3.10 Override vs. Overload

Override.

- Run-time polymorphism.
- Subclass provides new behavior for a method with the same signature as its superclass.

Overload.

- Compile-time polymorphism.
- Multiple methods share the same name but differ in parameter lists.
- Common in constructors.
- Some languages allow operator overloading (e.g., redefining `+` in Python via ___add___).

3.11 Design by Contract

Principle.

- Public methods of a class define a **contract** between the class and its clients.
- Subclasses must honor this contract when overriding methods.
- Breaking contracts leads to fragile systems.

Explicit vs. Implicit Contracts.

- **Implicit**: enforced by convention (e.g., in Java or Python).
- **Explicit**: supported by languages like Eiffel, which allow preconditions, postconditions, and invariants to be stated directly in code.

3.12 Interfaces and Generic Components

Interfaces.

- Define a set of abstract methods without implementation.
- Useful to represent roles in a system.
- Example: A `Pet` interface with `play()` or `takeForWalk()` methods can be implemented by multiple classes.

Generic Components.

- Abstract definitions of data structures or algorithms independent of their contents.
- Allow reuse of logic across many types.
- Example: Java's `List<T>` can hold any type while maintaining consistent semantics.

3.13 Object Identity and Equality

Identity.

- Each object has a unique identity, often tied to memory location or runtime-assigned IDs.
- Identity is distinct from equality of content.

Design Implications.

- Not all problem-domain entities require unique identity.
- Example:
  - A flight: identity may be defined by flight number and route.
  - A crate of carrots: individual carrots do not require unique identity in most systems.

3.14 Summary

- Core OOAD concepts:
  - Classes, Objects, Attributes, Methods.
  - Subclasses, Superclasses, Inheritance.
  - Encapsulation and Accessibility.
  - Constructors and Destructors.
  - Polymorphism and Abstraction.
  - Delegation, Association, Aggregation, Composition.
  - Override vs. Overload.
  - Design by Contract.
  - Abstract Classes, Interfaces, Generics.
  - Object Identity and Equality.
- Together, these principles enable robust, modular, and extensible designs.

4 Testing and Test-Driven Development in OOAD

4.1 Learning Objectives

- Understand the unique challenges of testing software systems.
- Examine testing across multiple levels of requirements.
- Recognize why early discovery of defects is critical for cost and schedule.
- Compare different testing approaches and explore the benefits of Test-Driven Development (TDD).

4.2 Introduction to Software Testing

- Software testing is a vast discipline — entire certifications exist (e.g., ASTQB).
- Testing is as essential as design: reliable code cannot exist without structured testing.
- Exhaustive testing is almost always infeasible due to the huge number of possible inputs.

4.3 Key Definitions

Verification
Ensuring code meets engineering requirements ("Did we build the system right?").

Validation
Ensuring code meets user expectations ("Did we build the right system?").

Testing
Running code with selected inputs and checking outputs.

Formal Verification
Using mathematical proof or specification (e.g., Z notation) to ensure correctness.

4.4 Verification vs. Validation

- **Verification:** Does the system conform to its technical specification?
- **Validation:** Does the system deliver value for the intended user?
- Both are essential — software can meet specifications but still fail to satisfy end-users.

4.5 Software Quality Expectations

- Typical industry: 1–10 defects per KLOC.
- High-quality libraries (e.g., Java core): 0.1–1 defects/KLOC.
- Safety-critical systems (NASA): 0.01–0.1 defects/KLOC.
- Scale: For 100,000 LOC of industry code, even at 1 defect/KLOC, 100 bugs may be missed.

4.6 Challenges in Testing

Exhaustive Testing.

- Impossible in practice (e.g., 2^{64} possible input combinations for a floating-point multiply).

Haphazard Testing.

- Informal "try it and see" testing is biased and rarely uncovers hidden errors.

Software vs. Physical Systems.

- Software failures are **discontinuous**: small input changes can trigger catastrophic failures.
- Example: 1994 Intel Pentium division bug (1 in 9 billion cases).
- Unlike physical systems, software rarely provides "early warning" of failure.

4.7 Testing Perspectives and Levels

- Testing must happen at multiple levels:
  - **Unit tests**: verify small modules.
  - **Integration tests**: verify interactions between modules.
  - **System tests**: verify the full application.
  - **Acceptance tests**: verify satisfaction of user requirements.
- The **V-model** emphasizes testing activities at every level of design.

4.8 Developer vs. Tester Mindset

- Developers aim to make code work.
- Testers aim to make code fail.
- Healthy tension ensures robustness.

4.9 Test-After Development (Problems)

- Common practice: build code first, test later.
- Testing becomes deprioritized compared to delivery.
- Defects discovered late are expensive to fix.
- Can derail project timelines due to unexpected issues.

4.10 The Cost of Late Defect Discovery

- "Surprises only get more expensive if discovered later."
- The **Cone of Uncertainty**: project scope and schedule are highly uncertain early on.
- Testing early reduces uncertainty and lowers cost of rework.

4.11 Test-With Development

- Improves on test-after: write code and test cases simultaneously.
- Benefits:
  - Forces consideration of how code will be tested.
  - Enables automation for regression testing.

4.12 Test-Driven Development (TDD)

Principles.

1. Write test cases before writing the code.
2. Write the minimum code needed to pass the test.
3. Refactor to improve structure while keeping tests green.

Benefits.

- Forces precise requirement specification.
- Discovers specification defects early.
- Encourages clean, modular design.

4.13 Blending TDD and OOAD

- OOAD: conceptual, high-level design and responsibility assignment.
- TDD: details-first, code-focused.
- Blended approach:
  1. Identify problem, draw entities and relationships.
  2. Define class responsibilities.
  3. Write failing tests for these responsibilities.
  4. Implement code to make tests pass.
  5. Iterate, refine, and refactor.
- OOAD = conceptual foundation; TDD = specification/implementation method.

4.14 Testing Strategies

Black Box Testing.

- Tests system behavior against specifications.
- Ignores internal structure.
- Good for acceptance and regression testing.

White Box Testing.

- Examines internal structure and code paths.
- Finds design-level defects and unreachable paths.
- Requires updates if implementation changes.

4.15 Test Coverage

- **Statement coverage**: every statement is executed at least once.
- **Branch coverage**: every conditional branch tested.
- **Path coverage**: every possible path (usually infeasible).
- Code coverage tools measure how thoroughly tests exercise code.
- Goal: maximize coverage while balancing feasibility.

4.16 Summary

- Testing ensures delivery of software that meets both requirements and expectations.
- Early defect discovery is essential for cost and schedule control.
- Test-after is risky; test-with is better; test-first (TDD) is best.
- OOAD and TDD can complement each other when blended carefully.
- Strong testing strategies include black-box, white-box, and coverage measurement.

5 Serialization and Messaging

5.1 Learning Objectives

In this section we aim to:

- Understand the relationship between messaging and integrating object-oriented system elements.
- Identify tools and their characteristics for messaging implementations.
- Learn the processes of serialization and deserialization in communications.
- Explore alternative approaches to serialization across languages and formats.

5.2 Why Serialization and Messaging?

Messaging is a fundamental mechanism for integrating distributed system components. It reliably delivers data and commands from one element to another. Since most messaging involves streams of bytes, a process is required to convert objects, data, and commands into a byte stream: this is known as **serialization**. The reverse process is **deserialization**, which restores the original object representation from its byte stream.

Serialization and messaging are key enablers for object-based systems, and are critical to design patterns such as Proxy and Memento, which depend on reliable data conversion and interconnection:contentReference[oaicite:0]index=0

5.3 Messaging Concepts

Messaging refers to the communication of data packets that require attention by receiving systems. It is central to systems ranging from Java-based distributed objects to IoT systems with cloud APIs. Alternatives to messaging include:

- Shared memory areas.
- Databases or persistent storage.
- File systems.

Nevertheless, most object-oriented systems rely on sending messages between objects to ensure modularity and extensibility. As Grady Booch observed, "At a certain level of abstraction, every system is a message passing system.":contentReference[oaicite:1]index=1

5.4 Message Channels and Processing

Messages typically pass over channels, and require:

- Construction and parsing.
- Methods and architectures for establishing connections.
- Routing mechanisms to ensure delivery.
- Transformation for compatibility across protocols.
- Observation and monitoring for quality of service.

Messages can encode data updates, events, commands, requests, or other forms of transactional information:contentReference[oaicite:2]index=2.

5.5 Message Architecture and Patterns

Messages often require addressing, headers, and control metadata for proper routing. Some messages may need to be fragmented, queued, or assembled. Architectural choices for messaging include:

- Publish–Subscribe
- Request–Response
- Fan-Out
- Point-to-Point

- Databus
- Survey messaging

Beyond these, the *Enterprise Integration Patterns* catalog documents over 65 recurring messaging design patterns, covering construction, routing, transformation, and monitoring:contentReference[oaicite:3]index=3.

5.6 Messaging Tools

Several tools and frameworks provide practical implementations of messaging:

ZeroMQ: A lightweight library offering brokerless messaging with sockets. Supports pub-sub, fan-out, and request-response patterns. Minimal overhead, cross-platform, LGPL licensed.

RabbitMQ: A widely used open-source message broker implementing AMQP 0.9.1 (and others). Supports pub/sub, request/reply, work queues, and monitoring via a management interface.

AWS SQS: A managed cloud-based queuing service. Offers standard queues (high throughput, at-least-once delivery) and FIFO queues (guaranteed ordering and exactly-once processing). Easily integrates with AWS Lambda and IoT services.

5.7 Serialization and Deserialization

Serialization converts an object's state into a byte stream, while **deserialization** reconstructs the object. This process is required for transmitting objects over a network or storing them in databases. Only primitive types or serialized objects can cross network proxy boundaries:contentReference[oaicite:4]index=4.

5.7.1 Serialization in Java

In Java, a class must implement the `Serializable` interface to be eligible for serialization:

```
public class Person implements Serializable {
 private static final long serialVersionUID = 1L;
 static String country = "ITALY";
 private int age;
 private String name;
 transient int height; // ignored during serialization
}
```

Key points:

- Static fields are not serialized since they belong to the class, not the object.
- The `transient` keyword prevents certain fields from being serialized.
- A unique `serialVersionUID` should be declared for version control.

5.7.2 Serialization in Python

Python provides the `pickle` module for serialization:

- `pickle.dumps()` serializes an object hierarchy into a byte stream.
- `pickle.loads()` deserializes the byte stream back into objects.

```
import pickle
data = {'a':1, 'b':2}
s = pickle.dumps(data)
restored = pickle.loads(s)
```

5.7.3 Other Formats

Serialization formats also include:

- **CSV:** Comma-separated values for tabular data.
- **JSON:** Lightweight text-based format with nested structures.
- **XML:** Markup-based representation with attributes and hierarchy.
- **YAML:** Human-readable structured representation.

5.8 Summary

Serialization and messaging form the backbone of object-oriented systems that integrate distributed components. Whether via brokered queues or lightweight channels, objects communicate through messages encoded as byte streams. Serialization ensures that objects can cross system boundaries, enabling interoperability and scalability:contentReference[oaicite:5]index=5.

Part III
Intro to Design Patterns

6 Introduction and Learning Objectives

Design patterns are reusable solutions to common software design problems that arise in object-oriented development. The objectives of studying patterns include:

- Introducing the concept of object-oriented (OO) design patterns.
- Tracing their origins in architecture and anthropology.
- Explaining why design patterns are important and the advantages they provide.
- Preparing developers to recognize, discuss, and apply patterns in their own design work.

6.1 Origins of Design Patterns

The concept of design patterns in software draws inspiration from two distinct fields:

1. **Architecture (Christopher Alexander)** In the 1970s, Alexander studied what makes architectural designs "good." He observed that high-quality designs for structures such as towns, buildings, and community centers often shared recurring solutions. These recurring solutions became known as *patterns*. Each pattern describes:

   - A problem that recurs frequently in a given environment.
   - A proven solution that balances the necessary constraints.
   - An approach that can be reused infinitely without producing identical outcomes.

   Alexander identified four key elements:

   (a) **Name**: A concise identifier.
   (b) **Purpose**: The problem it solves.
   (c) **Solution**: The method or structure that addresses the problem.
   (d) **Constraints**: Considerations or trade-offs that affect the solution.

2. **Cultural Anthropology** Patterns also have roots in anthropology, where cultures agree on what constitutes good design. Patterns capture these culturally-shared judgments by identifying structures and relationships that recur across well-designed artifacts. This gives an *objective basis* for evaluating design quality.

6.2 The Gang of Four and the Rise of OO Patterns

In 1995, the book *Design Patterns: Elements of Reusable Object-Oriented Software* was published by the "Gang of Four" (Gamma, Helm, Johnson, Vlissides). It cataloged 23 recurring OO design patterns, chosen because they had been independently identified in at least three real-world systems. Importantly:

- The authors did not *invent* these patterns; they documented existing practices.
- Subsequent works have expanded the catalog, often domain- or language-specific.
- Learning patterns early can accelerate mastery of OO design, rather than being a topic reserved only for experts.

6.3 Design Patterns in Software Engineering

Design patterns in software aim to answer key questions:

- Are there recurring problems in software design that can be solved in consistent, reusable ways?
- Can the success of prior designs be captured in a form that allows future reuse?

The consensus answer is "yes," and this gave rise to the systematic study of software design patterns.

The Gang of Four book organizes its patterns into categories such as *creational*, *structural*, and *behavioral*, each addressing different recurring challenges in OO systems.

6.4 Key Elements of a Software Design Pattern

Software patterns are generally described with the following components:

- **Name**: A concise and evocative identifier.
- **Intent**: The purpose of the pattern.
- **Problem**: The context and recurring issue the pattern addresses.
- **Solution**: The structural and interaction-based approach taken to solve the problem.
- **Participants**: The entities involved and their roles.
- **Consequences**: The trade-offs, benefits, and drawbacks of applying the pattern.
- **Implementation**: Guidance and examples of how the pattern can be applied.
- **Structure**: A UML class or interaction diagram capturing its form.

6.5 Why Patterns Matter

Patterns offer a range of benefits to developers:

1. **Experience reuse**: Instead of reinventing solutions, developers reuse proven approaches.
2. **Shared vocabulary**: Patterns provide a concise way to discuss designs. For example, "I used an Observer pattern" instantly communicates the solution without restating its mechanics.
3. **Improved design quality**: Patterns guide developers toward flexible, maintainable, and adaptable systems.
4. **Learning motivation**: Junior developers see the value of patterns and become motivated to deepen their design knowledge.
5. **Reinforcement of OO principles**: Patterns highlight and encourage principles such as coding to interfaces, favoring composition over inheritance, and encapsulating what varies.

6.6 Patterns as Perspective

Design patterns provide a higher-level lens for thinking about software:

- They abstract away from low-level implementation details.
- They let designers focus on recurring structures and relationships.
- They place individual design choices within a larger body of shared knowledge.

This conceptual shift allows developers to avoid "getting bogged down" too early in coding details, instead working at the level of design reasoning.

6.7 Analogy: Carpentry and Software

Just as carpenters have a shared language for discussing wood joints (e.g., dovetail vs. miter), software developers can use patterns to concisely describe design options. Mastery of the vocabulary enables efficient communication and evaluation of alternatives.

6.8 Design Patterns and Software Quality

Design patterns embody the idea that quality in software can be judged objectively. By comparing high-quality and low-quality designs, one can identify:

- What positive attributes (*X's*) are present in successful designs.
- What negative attributes (*Y's*) appear in poor designs.

The design process then aims to maximize the X's while minimizing the Y's.

6.9 Advantages of Using Patterns

1. **Maintainability**: Many patterns yield systems that are easier to extend and modify.
2. **Flexibility**: Patterns allow designs to better accommodate change.
3. **Communication**: Patterns enable teams to discuss designs at a higher level of abstraction.
4. **Consistency**: They provide standardized solutions to recurring issues.

6.10 Categories of Design Patterns

Design patterns are commonly grouped into three major categories based on the nature of the problems they solve and how they achieve reusability: **Creational**, **Structural**, and **Behavioral** patterns.

6.10.1 Creational Patterns

Creational patterns focus on **object instantiation** mechanisms. They abstract the process of object creation to make a system independent of how its objects are constructed, composed, and represented. The key idea is to provide greater flexibility and reduce coupling between the client code and the concrete classes it uses.

- **Intent:** Control how objects are created to achieve flexibility and reuse.
- **Benefits:** Promote loose coupling, hide instantiation logic, and improve maintainability.
- **Common Patterns:**
  - **Factory Method** – Defines an interface for creating objects but lets subclasses decide which class to instantiate.
  - **Abstract Factory** – Creates families of related objects without specifying their concrete classes.
  - **Builder** – Separates the construction of a complex object from its representation.
  - **Prototype** – Creates new objects by cloning an existing instance.
  - **Singleton** – Ensures only one instance of a class exists and provides global access to it.
- **Examples:** Cross-platform GUI libraries, database driver factories, configuration managers.

6.10.2 Structural Patterns

Structural patterns deal with the **composition of classes and objects** to form larger structures while keeping them flexible and efficient. They help ensure that changes in one part of the system minimally affect others.

- **Intent:** Define simple ways to compose objects or classes to form complex structures.
- **Benefits:** Improve code maintainability, reusability, and adaptability to change.
- **Common Patterns:**
  - **Adapter** – Converts the interface of one class into another expected by the client.
  - **Bridge** – Decouples abstraction from implementation so that both can evolve independently.

  - **Composite** – Treats individual objects and groups of objects uniformly.
  - **Decorator** – Dynamically attaches responsibilities to objects without altering their code.
  - **Facade** – Provides a simplified interface to a complex subsystem.
  - **Flyweight** – Shares objects to support large numbers of fine-grained instances efficiently.
  - **Proxy** – Controls access to another object, adding functionality like lazy loading or security.
- **Examples:** GUI component hierarchies, API bridges, caching layers.

6.10.3 Behavioral Patterns

Behavioral patterns focus on the **interaction and communication between objects**, distributing responsibilities and improving flexibility. They define how objects collaborate to perform tasks without tightly coupling their implementations.

- **Intent:** Simplify communication between objects and make the system more dynamic.
- **Benefits:** Increase flexibility, promote delegation, and reduce dependencies between components.
- **Common Patterns:**
  - **Strategy** – Encapsulates interchangeable algorithms and lets clients choose them at runtime.
  - **Observer** – Defines a one-to-many dependency so that observers are automatically notified of state changes.
  - **Command** – Encapsulates a request as an object, enabling undo/redo, queuing, and logging.
  - **Iterator** – Provides a standard way to traverse elements in a collection without exposing its representation.
  - **Mediator** – Centralizes complex communication between objects, reducing direct dependencies.
  - **Memento** – Captures and restores an object's internal state without violating encapsulation.
  - **Template Method** – Defines the skeleton of an algorithm, deferring some steps to subclasses.
  - **Visitor** – Adds new operations to existing object structures without modifying them.
  - **Chain of Responsibility** – Passes a request along a chain of handlers until one processes it.
  - **Interpreter** – Defines a grammar and an interpreter for processing structured expressions.
- **Examples:** Event-driven GUIs, logging frameworks, workflow engines.

6.11 Summary and Forward Look

Design patterns are recurring solutions to recurring problems. They:

- Solve common OO design problems.
- Reinforce fundamental OO principles.
- Provide a shared design language.
- Help teams build higher-quality, more maintainable software.

From here, we transition into specific patterns, beginning with **Strategy** and **Observer**, which illustrate how patterns solve real design challenges while reinforcing core principles.

7 The Strategy Pattern

7.1 Learning Objectives

- Review our first design pattern: the **Strategy Pattern**.
- Understand the motivation through a guided design exercise (the duck simulator).
- Learn how OO principles (encapsulation, delegation, programming to interfaces) lead to the Strategy solution.
- Analyze trade-offs of inheritance, interfaces, and composition.
- Apply Strategy to eliminate conditionals, promote flexibility, and encapsulate algorithms.

7.2 The Duck Simulator Example

The Strategy Pattern is introduced through the development of a duck-based video game (adapted from *Head First Design Patterns*). Initially, the design begins with a simple `Duck` class:

- `name` attribute (optional).
- `makeNoise()` for quacking.
- `swim()` for movement.
- `render()` for graphics.

At this conceptual stage, the emphasis is on responsibilities and behaviors, not yet on implementation details.

7.2.1 Differentiating Ducks

As new duck species are introduced, inheritance is used. Subclasses override `render()`, while `makeNoise()` and `swim()` remain in the base class. Later, a new requirement—`fly()`—is added at the root `Duck` class, allowing all ducks to fly.

7.2.2 Problems with Inheritance

Special cases create difficulties:

- **RubberDuck:** squeaks instead of quacks, cannot fly.
- **WoodenDecoyDuck:** silent and cannot fly.

Maintaining consistent behavior through inheritance becomes error-prone and rigid.

7.3 Interfaces vs. Inheritance

An alternative is to use interfaces such as `Flight` and `Noise`, but this leads to duplicated code across subclasses. Abstract base classes could help, but languages with single inheritance struggle to combine multiple hierarchies cleanly.

7.4 OO Principles to the Rescue

Guiding principles (from *Head First*):

- Encapsulate what varies.
- Favor composition over inheritance.
- Program to an interface, not an implementation.
- Strive for loosely coupled designs.
- Open/Closed Principle: classes should be open for extension but closed for modification.

- Depend on abstractions, not concrete classes.
- A class should have only one reason to change.

For ducks, the varying behaviors are `fly()` and `makeNoise()`. These are pulled out of `Duck` and encapsulated as separate behavior classes.

7.5 Refactoring Ducks with Behaviors

Two new hierarchies are introduced:

- `FlightBehavior`: `WingFlier`, `NoFly`, `FlyIfThrown`.
- `NoiseBehavior`: `Quack`, `Squeak`, `Silent`.

The `Duck` class delegates behavior to these strategies:

```
public abstract class Duck {
 FlightBehavior flightBehavior;
 NoiseBehavior noiseBehavior;

 public void performFlight() {
  flightBehavior.fly();
 }
 public void setFlightBehavior(FlightBehavior fb) {
  flightBehavior = fb;
 }

 public void performNoise() {
  noiseBehavior.makeNoise();
 }
 public void setNoiseBehavior(NoiseBehavior nb) {
  noiseBehavior = nb;
 }
}
```

FlightBehavior hierarchy (interface + concrete strategies).

```
/** Strategy interface for flight-related behavior. */
public interface FlightBehavior {
 void fly();
}

/** Concrete strategy: normal wing-powered flight. */
public final class WingFlier implements FlightBehavior {
 @Override public void fly() {
  System.out.println("Flapping wings: flying normally.");
 }
}

/** Concrete strategy: cannot fly (e.g., rubber or wooden decoys). */
public final class NoFly implements FlightBehavior {
 @Override public void fly() {
  // intentionally does nothing
 }
}

/** Concrete strategy: brief "flight" only when thrown (no propulsion). */
```

```
public final class FlyIfThrown implements FlightBehavior {
 @Override public void fly() {
  System.out.println("Glides briefly when thrown.");
 }
}
```

NoiseBehavior hierarchy (interface + concrete strategies).

```
/** Strategy interface for vocalization / sound behavior. */
public interface NoiseBehavior {
 void makeNoise();
}

/** Concrete strategy: standard duck quack. */
public final class Quack implements NoiseBehavior {
 @Override public void makeNoise() {
  System.out.println("Quack!");
 }
}

/** Concrete strategy: squeak (e.g., rubber duck). */
public final class Squeak implements NoiseBehavior {
 @Override public void makeNoise() {
  System.out.println("Squeak!");
 }
}

/** Concrete strategy: silence (e.g., wooden decoy). */
public final class Silent implements NoiseBehavior {
 @Override public void makeNoise() {
  // intentionally silent
 }
}
```

7.6 Code to Interfaces

The duck holds references of type `FlightBehavior` and `NoiseBehavior`, not concrete implementations. This makes the design loosely coupled: the duck can be configured with any behavior implementation without changing its class.

7.7 Delegation in Action

By delegating to behavior objects:

- Subclasses no longer need to override `fly()` or `makeNoise()` directly.
- Wooden ducks and rubber ducks simply plug in `NoFly` or `Silent`.
- Behaviors can be changed at runtime:

```
Duck d = new RubberDuck();
d.performFlight(); // NoFly
d.setFlightBehavior(new WingFlier());
d.performFlight(); // Now flies!
```

Clarifying Note - Concrete RubberDuck Class.

```
/** A rubber duck cannot fly and squeaks. */
public final class RubberDuck extends Duck {

 public RubberDuck() {
  /* Set default behaviors */
  this.flightBehavior = new NoFly();   // cannot fly
  this.noiseBehavior  = new Squeak();  // squeaks
 }
 }
}
```

7.8 The Strategy Pattern Defined

The Strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

7.9 When to Use Strategy

- When related classes differ only in behavior.
- To configure a class instance with a chosen behavior at instantiation.
- To encapsulate algorithm-specific data structures.
- To eliminate complex conditionals.

Trade-offs:

- Increased number of classes and objects.
- More flexible and extensible design.
- Cleaner, more maintainable code base.

7.10 Delegation Structure

A Strategy-based design involves:

- **Client:** uses the host class.
- **Host:** delegates behavior to strategy objects.
- **Strategy Interface:** defines the family of algorithms.
- **Concrete Strategies:** encapsulate specific algorithms.

7.11 UML Example for Strategy: Car Handling

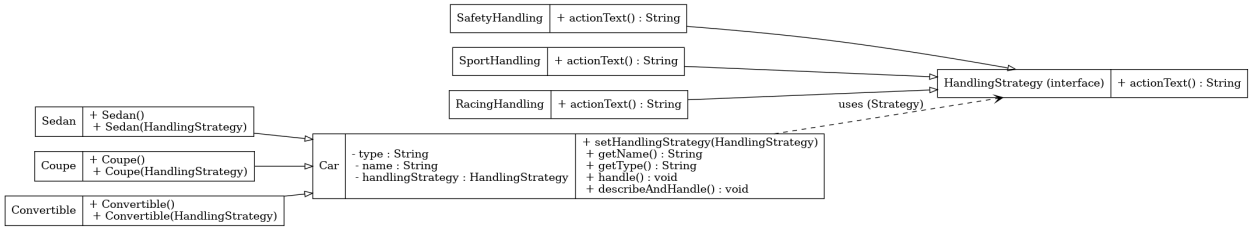


Figure 1: Strategy pattern applied to car handling.

7.12 Summary

The Strategy pattern is the first major OO pattern introduced. It enables:

- Assigning behavior dynamically.
- Avoiding subclass explosion and rigid inheritance hierarchies.
- Cleaner designs via delegation and encapsulation.
- Robust, extensible code for evolving systems.

Strategy replaces inheritance with flexible composition, providing a reusable solution to families of related behaviors.

Strategy — Interview Blurb

The Strategy pattern encapsulates what varies by moving algorithmic behavior into interchangeable objects, allowing clients to delegate rather than inherit. This avoids brittle hierarchies where subclasses are forced to implement or override unwanted base behaviors, promoting flexibility and cleaner composition.

8 Observer Pattern

8.1 Learning Objectives

- Consider categories of object-oriented patterns (creational, structural, behavioral).
- Review the Observer pattern, its principles, and its applications.
- Explore different approaches to using Observer in Java.

8.2 Categories of Patterns

OO design patterns are commonly grouped into three categories:

- **Creational patterns:** Focused on object instantiation, e.g., Factory, Abstract Factory, Singleton, Builder.
- **Structural patterns:** Concerned with composition of classes/objects into larger structures, e.g., Decorator, Adapter, Proxy, Composite.
- **Behavioral patterns:** Capture interactions and responsibilities of objects, e.g., Strategy, Observer, Command, Iterator, Mediator, Visitor.

8.3 Introduction to Observer

The **Observer pattern** enables objects to stay informed about events in a software system. It establishes a one-to-many dependency: when one object (the subject) changes, all dependent objects (observers) are notified automatically. Key properties:

- Dynamic: observers may subscribe/unsubscribe at run-time.
- Widely used: heavily integrated into the Java Development Kit and many frameworks.
- Promotes loose coupling: subjects only know that observers implement an interface, not their details.

8.4 Example: A Weather Monitor

Imagine a Raspberry Pi weather station with sensors for temperature, humidity, pressure, wind, and light:

- Data from sensors is collected by a `SensorData` class.
- Three display tabs in a GUI show: current conditions, statistics, and forecast.
- Each display should update when new sensor data arrives.

A naïve implementation tightly couples displays to sensors, making it hard to extend or modify. The Observer pattern decouples these parts, letting displays subscribe dynamically.

8.5 Observer Analogy

Observer works much like subscribing to a newspaper:

- A subject (publisher) produces updates.
- Observers (subscribers) register to receive notifications.
- Subscribers may join or leave at any time.
- Notifications are broadcasted without the subject knowing observer details.

8.6 Observer Interactions

- The **Subject** maintains a list of observers.
- When its state changes, it `notify()`s observers.
- Observers implement an `update()` method, deciding whether to pull more information from the subject.

8.7 UML Structure

The UML for Observer shows:

- A `Subject` interface with `attach()`, `detach()`, and `notify()`.
- An `Observer` interface with `update()`.
- Concrete subjects and observers implementing these interfaces.

8.8 Implementation Example (Java)

Subject interface:

```
public interface Subject {
 void registerObserver(Observer o);
 void removeObserver(Observer o);
 void notifyObservers();
}
```

Observer interface:

```
public interface Observer {
 void update(float temp, float humidity, float pressure);
}
```

ConcreteSubject (SensorData):

```
public class SensorData implements Subject {
 private List<Observer> observers = new ArrayList<>();
 private float temperature, humidity, pressure;

 public void setMeasurements(float t, float h, float p) {
  this.temperature = t;
  this.humidity = h;
  this.pressure = p;
  notifyObservers();
 }

 public void registerObserver(Observer o) {
  observers.Add(o);
 }

 public void removeObserver(Observer o) {
  observers.Remove(o);
 }

 public void notifyObservers() {
  for (Observer o : observers) {
   o.update(temperature, humidity, pressure);
  }
 }
}
```

ConcreteObserver (ConditionsDisplay):

```
public class ConditionsDisplay implements Observer {
 public void update(float temp, float humidity, float pressure) {
  System.out.println("Current: " + temp +
  "F, " + humidity + "% humidity");
 }
}
```

8.9 Benefits

- Loose coupling: subjects and observers interact only through interfaces.
- Flexibility: observers can be added/removed without changing subjects.
- Reusability: observers and subjects can be reused across contexts.

8.10 Concerns

- **Unexpected updates:** poorly defined dependencies may cause unintended cascades.
- **Overly simple protocol:** without detail in updates, observers may need to query subjects repeatedly.
- **Push vs. Pull:**
  - Pull: observers query subject for details.
  - Push: subject provides data directly in the update.

8.11 Variations

Observer implementations may vary in:

- Event objects (e.g., classes, data types, generics).
- Subscription management: by subject, broker, or manager.
- Complex flows: queued, prioritized, or addressed events.
- Hybrid roles: components that are both publishers and subscribers.

8.12 Observer in Java

Options for Java developers:

- Create custom Subject/Observer interfaces (as above).
- Use `PropertyChangeListener` and `PropertyChangeSupport`.
- Use Java's `Flow` API (`Publisher`, `Subscriber`, `Processor`, `Subscription`).

Note: Java's built-in `Observer/Observable` was deprecated in Java 9 due to issues with inheritance, ordering, serialization, and thread safety.

8.13 Pub/Sub Models

Observer is conceptually similar to publish/subscribe messaging. External tools like RabbitMQ, ZeroMQ, or MQTT can extend this model to distributed systems.

8.14 Summary

The Observer pattern is one of the most widely used behavioral patterns:

- Defines one-to-many dependency for automatic updates.
- Promotes loose coupling between interacting objects.
- Useful in GUIs, event-driven systems, and distributed messaging.
- Java developers can choose between custom interfaces, `PropertyChangeListener`, or the `Flow` API.

Observer — Interview Blurb

Defines a one-to-many dependency between objects so that when a subject changes state, all its observers are notified automatically. This decouples event producers from consumers, avoiding tight coupling and enabling dynamic registration of listeners or subscribers.

9 Foundational Patterns Quiz with Answers & Explanations

This quiz reviews key concepts related to the origins of design patterns, the Strategy pattern, the Observer pattern, and foundational OOAD ideas.

Q1. Origins of Design Patterns

Regarding the origins of design patterns, which of the following are correct?

- ✗ Christopher Alexander is a renowned software developer who first presented the first software patterns
- ✓ The four elements of Alexander's patterns include – a **name**, a **purpose**, **how to solve the problem**, and **what constraints we have to consider**
- ✗ The Gang of Four initially developed a description of architectural patterns in the book *Timeless Way of Building*
- ✓ The Gang of Four book lists **23 software patterns for OOAD**, but there are others in practice

Explanation: Christopher Alexander was an **architect**, not a software developer, who introduced the concept of patterns in architecture. The GoF applied his ideas to software in their book *Design Patterns: Elements of Reusable Object-Oriented Software* (1994), which documents 23 OO patterns. *The Timeless Way of Building* is Alexander's book, not the GoF's.

Q2. Patterns in OOAD

Which of the following statements regarding patterns in OOAD is **NOT correct**?

- ✗ Study of patterns provides perspective on typical OOAD problems
- ✗ Patterns provide experienced developers concise terms to discuss possible design solutions
- ✗ Patterns provide both code and experience reuse
- ✓ **Patterns help in the conceptual level of design prior to implementation**

Explanation: Patterns are discovered, not invented. They describe proven solutions to recurring design problems and give developers a shared vocabulary. However, they are generally applied during **design and implementation**, not strictly at the high-level conceptual phase.

Q3. Strategy Pattern

Which of the following statements is **NOT true** about the Strategy pattern?

- ✓ **Strategy provides delegation in the Duck class example by overriding methods in Duck subclasses**
- ✗ In lecture, the Duck class diagram based on Strategy and delegation shows that `Duck` **HAS-A** reference to a `FlightBehavior` and a `MallardDuck` **IS-A** `Duck`
- ✗ The Strategy pattern supports the OO principle of favoring **delegation over inheritance**
- ✗ The Strategy pattern supports the OO principle of **encapsulating what varies**

Explanation: Strategy delegates behavior using composition (`Duck HAS-A FlightBehavior`). The incorrect statement assumes Strategy works by overriding methods in subclasses, but it actually delegates behavior to interchangeable strategy objects.

Q4. Observer Pattern Intent

The intent of the Observer pattern is to…

- ✗ Be a method used to instantiate other objects
- ✗ Create tight coupling through abstraction
- ✓ **Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified automatically**
- ✗ Allow for chaining multiple functionality handlers

Explanation: The Observer pattern establishes a **publish-subscribe relationship** between objects. Observers register interest in a subject, and when the subject changes, all observers are notified automatically — maintaining loose coupling.

Q5. Observer Pattern Characteristics

The Observer pattern… (Select all correct)

- ✓ **Supports publish-subscribe modeling in application designs**
- ✓ **Aids flexibility and supports decoupling class connectivity**
- ✓ **Can be implemented by hand or by using Java-based or external messaging frameworks**
- ✗ Supports push, but not pull, for moving information between objects

Explanation: Observer can be implemented manually or via frameworks like Java's `java.util.Observable`, or external event buses. It supports both the **push** model (subject pushes data to observers) and the **pull** model (observers request data on notification).

Part IV
Structural Design Patterns

10 Decorator Pattern

10.1 Learning Objectives

- Understand the purpose and motivation behind the **Decorator** pattern.
- Learn how to dynamically add responsibilities to objects without modifying their code.
- Compare Decorator to alternatives like subclassing and composition.
- Understand the structure via UML and practical implementation in Java.
- Explore advantages, trade-offs, and common pitfalls.

10.2 Introduction

The **Decorator Pattern** is a **structural** OO design pattern that lets you attach additional responsibilities to objects *dynamically* without changing their class. Decorators provide a flexible alternative to subclassing for extending functionality.

> *Definition (GoF):* "Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality."

10.3 Motivation

Consider designing a coffee shop ordering system:

- A `Beverage` base class has subclasses like `Espresso`, `Latte`, `DarkRoast`.
- Customers can add condiments like milk, soy, mocha, or whipped cream.
- A naïve implementation would require creating subclasses for every combination: `EspressoWithMilkAndMocha`, `DarkRoastWithSoy`, etc.
- This leads to **subclass explosion** and makes adding new condiments painful.

Instead, decorators allow you to "wrap" a base object with any number of new features without modifying its class.

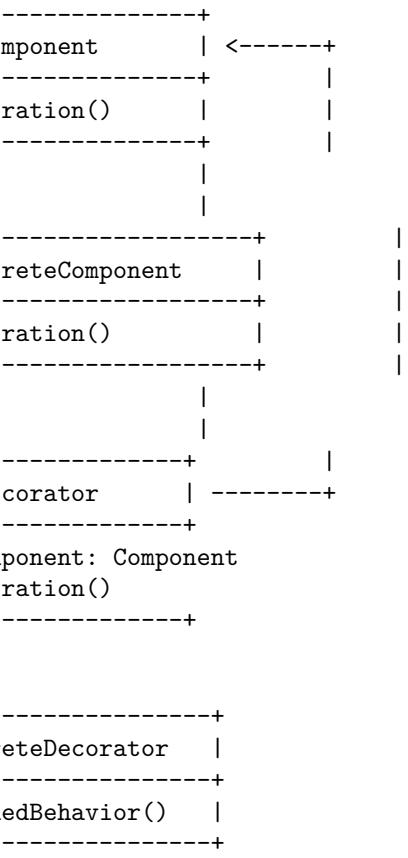
10.4 Key Concepts

- **Component:** The original interface or abstract class (e.g., `Beverage`).
- **Concrete Component:** A class implementing the component interface (e.g., `Espresso`).
- **Decorator:** An abstract class that implements the same interface but contains a reference to a `Component`.
- **Concrete Decorators:** Subclasses of `Decorator` that add responsibilities (e.g., `Mocha`, `Whip`).

10.5 UML Structure

The UML relationships:

- Both `Component` and `Decorator` define the same interface.
- The `Decorator` contains a reference to a `Component`.
- `ConcreteDecorator` overrides behavior while delegating base operations to the wrapped component.



10.6 Java Implementation

Step 1: Define the component interface.

```
public interface Beverage {
 String getDescription();
 double cost();
}
```

Step 2: Implement a concrete component.

```
public class Espresso implements Beverage {
 public String getDescription() { return "Espresso"; }
 public double cost() { return 1.99; }
}
```

Step 3: Create an abstract decorator. Note the implementation of the Beverage interface, just like the concrete components! Note also that as an abstract class, it doesn't have to have default implementations of the interface. The children will do this.

```
public abstract class CondimentDecorator implements Beverage {
 protected Beverage beverage;
 public CondimentDecorator(Beverage beverage) {
  this.beverage = beverage;
 }
}
```

Step 4: Implement concrete decorators.

```
public class Mocha extends CondimentDecorator {
 public Mocha(Beverage beverage) { super(beverage); }
 public String getDescription() {
  return beverage.getDescription() + ", Mocha";
 }
 public double cost() { return beverage.cost() + 0.20; }
}
```

Step 5: Use the decorator.

```
Beverage beverage = new Espresso();
beverage = new Mocha(beverage);
beverage = new Mocha(beverage);
System.out.println(beverage.getDescription() + " $" + beverage.cost());
// Output: Espresso, Mocha, Mocha $2.39
```

10.7 Python Implementation

Python uses duck typing, making decorators even more flexible:

```
class Beverage:
def cost(self):
return 0

class Espresso(Beverage):
def cost(self):
return 1.99

class Mocha:
def __init__(self, beverage):
self.beverage = beverage
def cost(self):
return self.beverage.cost() + 0.20

drink = Mocha(Mocha(Espresso()))
print(drink.cost())  # 2.39
```

10.8 Advantages

- **Open/Closed Principle:** New functionality is added without modifying existing classes.
- **Flexible composition:** Combine decorators in any order.
- Avoids subclass explosion.
- Enhances functionality at runtime.

10.9 Drawbacks

- Creates many small classes, which can increase complexity.
- Debugging and tracing execution flow can become harder.
- Clients must understand the composition of decorators to interpret behavior.

10.10 Common Uses

- GUI toolkits (e.g., scrollbars, borders, themes).
- Input/output streams (e.g., `java.io.BufferedInputStream`).
- Logging frameworks.

10.11 Summary

- The **Decorator Pattern** dynamically attaches additional behavior to objects.
- Uses composition and delegation instead of inheritance.
- Promotes loose coupling and follows the Open/Closed Principle.
- Widely used in GUI frameworks, IO pipelines, and middleware.

Decorator — Interview Blurb

Attaches new responsibilities to an object dynamically by wrapping it with a decorator that shares the same interface. This allows behavior to be extended without modifying or subclassing the original class, preventing deep inheritance trees and promoting flexible composition of features.

11 Facade Pattern

11.1 Learning Objectives

- Understand the purpose of the **Facade** pattern.
- Learn how Facade simplifies complex subsystems by providing a unified interface.
- Compare Facade to related patterns like Adapter and Mediator.
- Explore its UML structure and Java/Python implementations.
- Analyze advantages, trade-offs, and common real-world applications.

11.2 Introduction

The **Facade Pattern** is a **structural** design pattern that provides a **simplified, unified interface** to a complex subsystem. It hides internal implementation details while exposing only what clients need.

> *Definition (GoF):* "Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use."

11.3 Motivation

Consider designing a home theater automation system:

- Components: DVD player, projector, amplifier, lights, and speakers.
- Without Facade, a client must manually control each subsystem:
  - Turn on the projector.
  - Lower the screen.
  - Set the amplifier's input and volume.
  - Power up the DVD player.
- This requires understanding every subsystem API and sequencing the calls correctly.

With a Facade, a single call like `homeTheater.watchMovie()` hides the complexity by internally coordinating subsystems.

11.4 Key Concepts

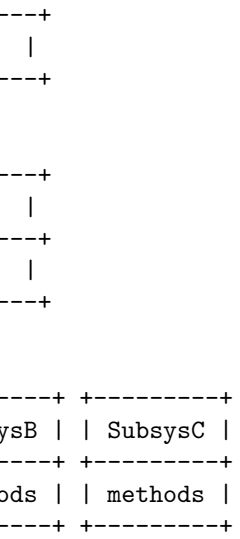
- **Subsystem Components:** A collection of related, lower-level APIs.
- **Facade:** A higher-level interface that simplifies usage of the subsystem.
- **Client:** Uses the facade without interacting directly with subsystem details.

11.5 When to Use Facade

- When working with a complex library or framework.
- To reduce coupling between clients and subsystems.
- When creating layered architectures, where each layer exposes a simplified interface to the next.

11.6 UML Structure

The UML shows how the Facade interacts with subsystems:



- The `Client` interacts with the `Facade`.
- The `Facade` coordinates subsystem calls internally.
- Subsystems remain unchanged but hidden from the client.

11.7 Java Implementation

Step 1: Subsystem classes.

```
public class DVDPlayer {
 public void on() { System.out.println("DVD Player ON"); }
 public void play(String movie) { System.out.println("Playing " + movie); }
 public void off() { System.out.println("DVD Player OFF"); }
}

public class Amplifier {
 public void on() { System.out.println("Amp ON"); }
 public void setVolume(int level) { System.out.println("Volume: " + level); }
 public void off() { System.out.println("Amp OFF"); }
}
```

Step 2: The Facade class.

```
public class HomeTheaterFacade {
 private DVDPlayer dvd;
 private Amplifier amp;

 public HomeTheaterFacade(DVDPlayer dvd, Amplifier amp) {
  this.dvd = dvd;
  this.amp = amp;
 }

 public void watchMovie(String movie) {
  System.out.println("Get ready to watch a movie...");
  amp.on();
```

```
  amp.setVolume(5);
  dvd.on();
  dvd.play(movie);
 }

 public void endMovie() {
  System.out.println("Shutting down theater...");
  dvd.off();
  amp.off();
 }
}
```

Step 3: Using the facade.

```
DVDPlayer dvd = new DVDPlayer();
Amplifier amp = new Amplifier();
HomeTheaterFacade theater = new HomeTheaterFacade(dvd, amp);

theater.watchMovie("Inception");
theater.endMovie();
```

11.8 Python Implementation

Python can implement the same idea more simply:

```
class DVDPlayer:
def on(self): print("DVD Player ON")
def play(self, movie): print(f"Playing {movie}")
def off(self): print("DVD Player OFF")

class Amplifier:
def on(self): print("Amp ON")
def set_volume(self, level): print(f"Volume: {level}")
def off(self): print("Amp OFF")

class HomeTheaterFacade:
def __init__(self, dvd, amp):
self.dvd, self.amp = dvd, amp
def watch_movie(self, movie):
print("Get ready to watch a movie...")
self.amp.on()
self.amp.set_volume(5)
self.dvd.on()
self.dvd.play(movie)
def end_movie(self):
print("Shutting down theater...")
self.dvd.off()
self.amp.off()

dvd = DVDPlayer()
amp = Amplifier()
theater = HomeTheaterFacade(dvd, amp)
theater.watch_movie("Inception")
theater.end_movie()
```

11.9 Advantages

- **Simplifies complexity:** Provides a single entry point to multiple subsystems.

- **Reduces coupling:** Clients depend only on the Facade, not the subsystems.
- **Improves maintainability:** Subsystem changes don't affect clients.
- **Supports layered architectures:** Commonly used in frameworks.

11.10 Drawbacks

- Risk of creating a **god object** if the Facade becomes too bloated.
- Limited flexibility — exposing fewer details may constrain advanced clients.
- Over-reliance on the Facade can hide powerful subsystem features.

11.11 Real-World Use Cases

- **Java APIs:** `javax.faces.context.FacesContext` provides a Facade for JSF subsystems.
- **Spring Framework:** `JdbcTemplate` simplifies database access.
- **Web Development:** Facades hide multiple backend service calls behind a single API.
- **Middleware Systems:** Unified APIs for network protocols, logging, or authentication.

11.12 Summary

- The **Facade Pattern** provides a single, unified interface to complex subsystems.
- Simplifies client interaction and reduces coupling.
- Frequently used in frameworks, layered architectures, and middleware.
- Complements other structural patterns such as Adapter and Decorator.

Facade — Interview Blurb

Provides a simplified, high-level interface to a complex subsystem, hiding implementation details and reducing client dependencies. This promotes loose coupling, clearer usage, and easier maintenance as subsystem components can change behind the facade without affecting external code.

12 Adapter Pattern

12.1 Learning Objectives

- Understand the intent of the **Adapter** pattern.
- Learn how to make incompatible interfaces work together.
- Compare Adapter with related structural patterns like **Facade** and **Decorator**.
- Study its UML structure and see practical Java and Python implementations.
- Explore advantages, trade-offs, and real-world applications.

12.2 Introduction

The **Adapter Pattern** is a **structural** design pattern that allows objects with incompatible interfaces to work together.

> *Definition (GoF):* "Convert the interface of a class into another interface the client expects. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces."

The Adapter pattern acts like a **translator** between two components: one producing a certain API and another expecting a different one.

12.3 Motivation

Imagine you are integrating a new third-party payment gateway into your existing system:

- Your app expects a method `processPayment(amount)`.
- The new gateway's SDK exposes a completely different API: `makeTransaction(total)`.
- You can't change the SDK, and you don't want to refactor all your existing code.

The solution: wrap the new SDK with an **Adapter** that translates your app's `processPayment()` calls into the gateway's `makeTransaction()` calls.

12.4 Key Concepts

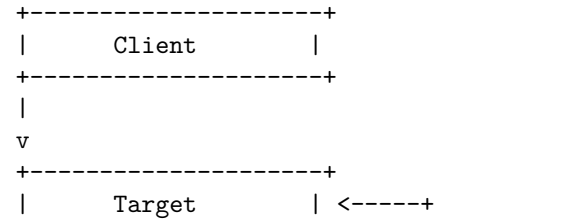
- **Target:** The expected interface used by the client.
- **Adaptee:** The existing class or system with an incompatible interface.
- **Adapter:** Bridges the gap by translating calls from the Target interface to the Adaptee.
- **Client:** Uses the Target interface without knowing about the underlying Adaptee.

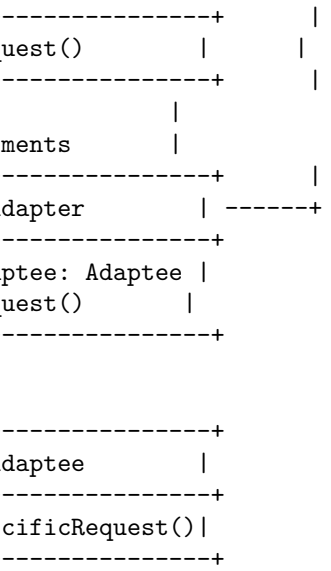
12.5 Types of Adapters

- **Object Adapter (Composition):** Uses delegation; the adapter holds a reference to the adaptee.
- **Class Adapter (Inheritance):** Uses multiple inheritance (less common; works better in languages like C++ than Java).

12.6 UML Structure

Below is the UML for the Object Adapter approach:





12.7 Java Implementation

Step 1: Define the Target interface.

```
public interface PaymentProcessor {
 void processPayment(double amount);
}
```

Step 2: Implement the Adaptee (incompatible SDK).

```
public class ThirdPartyGateway {
 public void makeTransaction(double total) {
  System.out.println("Processing payment of $" + total);
 }
}
```

Step 3: Create the Adapter.

```
public class PaymentAdapter implements PaymentProcessor {
 private ThirdPartyGateway gateway;

 public PaymentAdapter(ThirdPartyGateway gateway) {
  this.gateway = gateway;
 }

 @Override
 public void processPayment(double amount) {
  gateway.makeTransaction(amount);
 }
}
```

Step 4: Use the Adapter in the client.

```
ThirdPartyGateway gateway = new ThirdPartyGateway();
PaymentProcessor processor = new PaymentAdapter(gateway);

processor.processPayment(150.00);
```

12.8 Improving Java Implementation with Ports

Why ports? Classic *Adapter* wraps a single incompatible class to match a target interface. That's fine when you only have one SDK. But as soon as you want to swap providers (e.g., third-party today, in-house tomorrow), hard-coupling the adapter to a specific SDK forces changes in your app layer. A **Port** (aka Hexagonal / Ports & Adapters) defines a *stable boundary interface* that the application depends on; concrete providers sit *outside* that boundary behind *Adapters*. This applies the **Dependency Inversion Principle (DIP)**: high-level policy depends only on abstractions, not on volatile SDKs.

Dependency Inversion Principle (DIP)

High-level modules should not depend on low-level modules; both should depend on abstractions. DIP inverts the usual dependency direction by introducing stable interfaces (abstractions) between policy and detail, allowing implementation details (SDKs, databases, frameworks) to vary independently without breaking business logic.

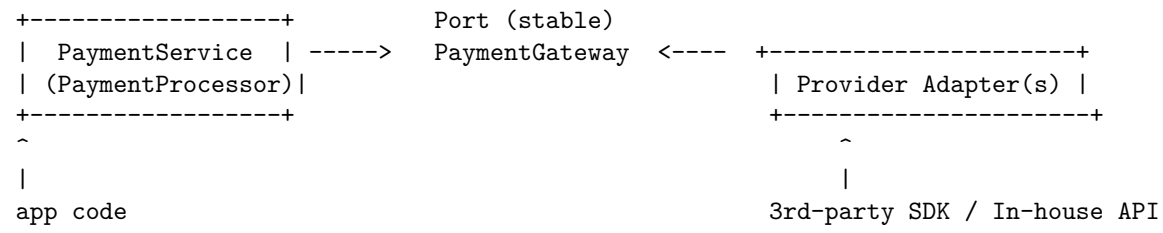
Interview Blurb — Adapter via Ports

Stabilize what varies by introducing a port (boundary interface) the app depends on, and write one adapter per provider to implement that port. Your business logic stays unchanged as providers change; you gain cleaner tests, lower coupling, and Open/Closed extensibility.

Structure (Intent → Mechanism → Benefit).

- **Intent:** Swap payment providers without touching business logic; isolate SDK quirks.
- **Mechanism:** Define a port `PaymentGateway`; implement provider-specific adapters; inject the port into an application service `PaymentService` that satisfies the app-facing `PaymentProcessor`.
- **Benefit:** Open/Closed for new providers, clear anti-corruption layer, easy mocking, simpler onboarding of legacy/third-party APIs.

Mental model (ASCII).



Java Implementation (Port + Adapters + Service)

App-facing interface (unchanged contract for clients):

```
public interface PaymentProcessor {
        void processPayment(double amount);
}
```

Port at the boundary (stable abstraction the app depends on):

```
public interface PaymentGateway {
        void makeTransaction(double total);
}
```

Application service depends on the port, not any SDK:

```
public final class PaymentService implements PaymentProcessor {
        private final PaymentGateway gateway;

        public PaymentService(PaymentGateway gateway) {
                this.gateway = gateway;
        }

        @Override
        public void processPayment(double amount) {
                gateway.makeTransaction(amount);
        }
}
```

Existing third-party SDK (adaptee, unchanged):

```
public class ThirdPartyGateway {
        public void makeTransaction(double total) {
                System.out.println("Third-party␣processing␣$" + total);
        }
}
```

Adapter: third-party SDK → Port:

```
public final class ThirdPartyGatewayAdapter implements PaymentGateway {
        private final ThirdPartyGateway sdk;

        public ThirdPartyGatewayAdapter(ThirdPartyGateway sdk) {
                this.sdk = sdk;
        }

        @Override
        public void makeTransaction(double total) {
                // map parameters, handle SDK quirks, translate exceptions, etc.
                sdk.makeTransaction(total);
        }
}
```

Tomorrow's in-house API (different adaptee):

```
public final class InHousePayments {
        public void makeTransaction(double total) {
                System.out.println("In-house␣processing␣$" + total);
        }
}
```

Adapter: in-house API → Port:

```
public final class InHouseGatewayAdapter implements PaymentGateway {
        private final InHousePayments api;

        public InHouseGatewayAdapter(InHousePayments api) {
                this.api = api;
        }

        @Override
        public void makeTransaction(double total) {
                api.makeTransaction(total);
        }
}
```

Client wiring (easy swap without touching business code):

```
// Choose a provider at composition time (DI container or manual wiring)
```

```
ThirdPartyGateway thirdParty = new ThirdPartyGateway();
PaymentGateway gateway = new ThirdPartyGatewayAdapter(thirdParty);
// or: PaymentGateway gateway = new InHouseGatewayAdapter(new InHousePayments());

PaymentProcessor processor = new PaymentService(gateway);
processor.processPayment(150.00);
```

Testing: With a port, you can unit test the application service without any SDK:

```
class FakeGateway implements PaymentGateway {
        double lastTotal = -1.0;
        @Override public void makeTransaction(double total) { lastTotal = total; }
}

// Test
FakeGateway fake = new FakeGateway();
PaymentProcessor svc = new PaymentService(fake);
svc.processPayment(42.50);
assert fake.lastTotal == 42.50;
```

Key takeaways.

- **DIP:** High-level policy (`PaymentService`) depends on a stable interface (`PaymentGateway`), not volatile SDKs.
- **OCP:** Adding a new provider is additive (new adapter) with zero changes to business code.
- **Testing & SLAs:** Mock the port for unit tests; enforce timeouts/retries/metrics centrally in the service or a cross-cutting adapter.
- **Anti-corruption:** SDK quirks (exceptions, currencies, async models) are localized in adapters.

12.9 Python Implementation

Python makes Adapter implementation more concise:

```
class ThirdPartyGateway:
def make_transaction(self, total):
print(f"Processing payment of ${total}")

class PaymentAdapter:
def __init__(self, gateway):
self.gateway = gateway
def process_payment(self, amount):
self.gateway.make_transaction(amount)

gateway = ThirdPartyGateway()
processor = PaymentAdapter(gateway)
processor.process_payment(150.00)
```

12.10 Comparison: Adapter vs. Facade

While both patterns simplify interactions, their intent differs:

- **Adapter:** Resolves interface incompatibilities — changes the *interface* to match expectations.
- **Facade:** Simplifies usage of a complex subsystem but does *not* change interfaces.

12.11 Advantages

- Promotes reuse of existing classes even with incompatible APIs.
- Decouples clients from third-party libraries or legacy systems.
- Works well with composition to maintain loose coupling.

12.12 Drawbacks

- Adds an extra layer of abstraction, which may slightly reduce performance.
- Multiple adapters may increase complexity when many subsystems are involved.

12.13 Real-World Use Cases

- **Java I/O Streams:** `InputStreamReader` adapts a byte-based `InputStream` to a character-based `Reader`.
- **Database Drivers:** JDBC adapters unify database access under a consistent API.
- **GUI Frameworks:** Adapters allow bridging between different widget libraries.
- **Third-Party Integrations:** Payment gateways, APIs, SDKs, etc.

12.14 Summary

- The **Adapter Pattern** converts one interface into another the client expects.
- Enables reuse of existing, incompatible components.
- Often used for API integration, I/O handling, database connections, and cross-library compatibility.
- Complements patterns like Facade and Decorator in structural design.

13 Proxy Pattern

13.1 Learning Objectives

- Understand the intent and purpose of the **Proxy** pattern.
- Learn how a proxy controls access to another object.
- Explore different types of proxies (virtual, protection, remote, caching).
- Examine UML structure and study practical Java and Python implementations.
- Identify real-world use cases and analyze advantages and trade-offs.

13.2 Introduction

The **Proxy Pattern** is a **structural** design pattern that provides a surrogate or placeholder for another object to control access to it.

> *Definition (GoF):* "Provide a surrogate or placeholder for another object to control access to it."

In other words, a proxy acts as an intermediary between the client and the real object. It presents the same interface as the underlying object but adds control or optimization logic before delegating requests.

13.3 Motivation

Imagine a large image gallery application:

- Images are stored remotely and are expensive to load.
- Without optimization, opening the gallery would fetch every image upfront.
- A better solution: load image objects immediately, but fetch image data *on demand*.

The proxy can represent the images initially, and when an image is actually needed, it fetches and caches the data.

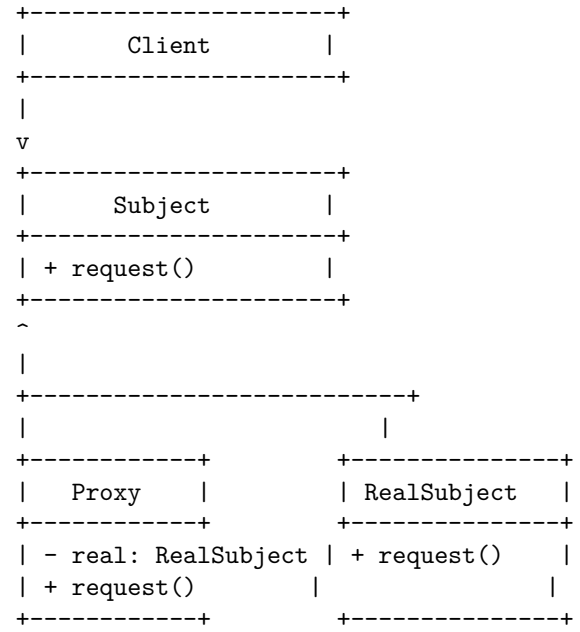
13.4 Key Concepts

- **Subject:** Defines the interface used by both the real object and the proxy.
- **RealSubject:** The actual object doing the work.
- **Proxy:** Controls access to the `RealSubject` and may add logic such as security checks, caching, or lazy initialization.
- **Client:** Interacts with the proxy as if it were the real object.

13.5 Types of Proxies

- **Virtual Proxy:** Delays expensive object creation until necessary (lazy loading).
- **Protection Proxy:** Controls access based on permissions (security layers).
- **Remote Proxy:** Represents an object in a different address space or server (e.g., RMI).
- **Caching Proxy:** Stores results of expensive operations for reuse.
- **Smart Proxy:** Adds logging, monitoring, or additional behaviors around the real object.

13.6 UML Structure



13.7 Java Implementation

Step 1: Define the Subject interface.

```
public interface Image {
 void display();
}
```

Step 2: Implement the RealSubject.

```
public class RealImage implements Image {
 private String filename;

 public RealImage(String filename) {
  this.filename = filename;
  loadFromDisk();
 }

 private void loadFromDisk() {
  System.out.println("Loading " + filename + " from disk...");
 }

 @Override
 public void display() {
  System.out.println("Displaying " + filename);
 }
}
```

Step 3: Implement the Proxy.

```
public class ProxyImage implements Image {
 private RealImage realImage;
 private String filename;

 public ProxyImage(String filename) {
```

```
  this.filename = filename;
 }

 @Override
 public void display() {
  if (realImage == null) {
   realImage = new RealImage(filename); // lazy initialization
  }
  realImage.display();
 }
}
```

Step 4: Using the Proxy.

```
Image img = new ProxyImage("highres_photo.jpg");

// Image loaded only when actually displayed:
System.out.println("Application started...");
img.display();
img.display(); // Uses cached image
```

13.8 Python Implementation

Python supports a concise implementation:

```
class RealImage:
def __init__(self, filename):
self.filename = filename
self._load_from_disk()

def _load_from_disk(self):
print(f"Loading {self.filename} from disk...")

def display(self):
print(f"Displaying {self.filename}")

class ProxyImage:
def __init__(self, filename):
self.filename = filename
self._real_image = None

def display(self):
if self._real_image is None:
self._real_image = RealImage(self.filename)
self._real_image.display()

img = ProxyImage("highres_photo.jpg")
print("Application started...")
img.display()
img.display()  # Uses cached image
```

13.9 Advantages

- **Performance optimization:** Supports lazy loading and caching.
- **Security control:** Useful for access restrictions.
- **Transparency:** Clients use the same interface whether interacting with the proxy or real object.
- **Extensibility:** Additional behaviors can be introduced without modifying the real object.

13.10 Drawbacks

- Adds an extra layer of indirection, potentially impacting performance for lightweight operations.
- Increased complexity in systems with many proxy types.
- Care must be taken to ensure synchronization when proxies are used in multi-threaded environments.

13.11 Real-World Use Cases

- **Remote Proxies:** Java RMI, gRPC, SOAP, RESTful clients.
- **Virtual Proxies:** Image rendering in GUI applications.
- **Protection Proxies:** Role-based access control in enterprise systems.
- **Caching Proxies:** Content delivery networks (e.g., Cloudflare, Akamai).
- **Smart Proxies:** Logging, monitoring, authentication.

13.12 Proxy vs. Decorator vs. Facade

- **Proxy:** Controls access to an object, possibly delaying, securing, or optimizing requests.
- **Decorator:** Dynamically adds new behavior to an object without altering its interface.
- **Facade:** Simplifies a complex subsystem by exposing a unified interface.

13.13 Summary

- The **Proxy Pattern** provides controlled access to an object by acting as an intermediary.
- Supports multiple use cases: lazy initialization, caching, security, logging, and remote access.
- Shares structural similarities with Decorator but focuses on *access control* rather than extending behavior.
- Common in enterprise applications, distributed systems, and performance-sensitive designs.

Proxy — Interview Blurb

Provides a surrogate or placeholder object that controls access to another object. The proxy implements the same interface as the real subject but can add cross-cutting behavior such as lazy instantiation, caching, access control, logging, or remote communication, all while keeping clients unaware of the indirection.

14 Composite Pattern

14.1 Learning Objectives

- Understand the intent and purpose of the **Composite** pattern.
- Learn how to represent hierarchical tree-like structures of objects.
- Explore its UML structure and analyze Java and Python implementations.
- Compare Composite with related structural patterns like Decorator and Proxy.
- Examine real-world applications and identify advantages and trade-offs.

14.2 Introduction

The **Composite Pattern** is a **structural** design pattern that allows you to treat **individual objects** and **compositions of objects** uniformly. It is ideal for modeling **part-whole hierarchies**.

> *Definition (GoF):* "Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly."

14.3 Motivation

Consider designing a file explorer:

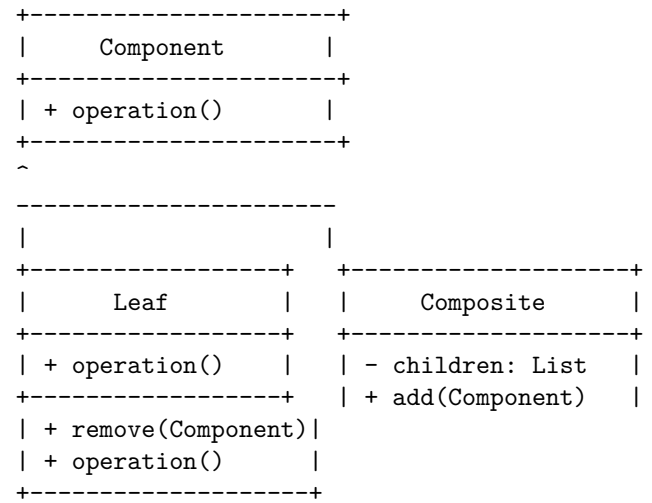
- Files and folders share common operations (e.g., rename, delete, getSize).
- A folder can contain files *and* other folders.
- Without Composite, you'd have to treat files and folders differently, leading to repetitive, complex client code.

The Composite pattern abstracts this: both files and folders implement the same interface, and folders delegate operations to their children recursively.

14.4 Key Concepts

- **Component:** The common interface for both simple and composite objects.
- **Leaf:** Represents individual objects (e.g., files).
- **Composite:** A container object holding child components (e.g., folders).
- **Client:** Works uniformly with both Leafs and Composites via the `Component` interface.

14.5 UML Structure



- `Component`: declares the common interface.
- `Leaf`: implements base behavior.
- `Composite`: stores child components and delegates operations recursively.
- `Client`: interacts with `Component` without knowing whether it's a `Leaf` or `Composite`.

14.6 Java Implementation

Step 1: Define the Component interface.

```
public interface FileSystemComponent {
 void showDetails();
}
```

Step 2: Implement the Leaf.

```
public class File implements FileSystemComponent {
 private String name;
 public File(String name) { this.name = name; }
 @Override
 public void showDetails() {
  System.out.println("File: " + name);
 }
}
```

Step 3: Implement the Composite.

```
import java.util.ArrayList;
import java.util.List;

public class Directory implements FileSystemComponent {
 private String name;
 private List<FileSystemComponent> children = new ArrayList<>();

 public Directory(String name) { this.name = name; }

 public void add(FileSystemComponent component) {
  children.add(component);
 }

 public void remove(FileSystemComponent component) {
  children.remove(component);
 }

 @Override
 public void showDetails() {
  System.out.println("Directory: " + name);
  for (FileSystemComponent c : children) {
   c.showDetails();
  }
 }
}
```

Step 4: Using the Composite.

```
public class Main {
 public static void main(String[] args) {
  FileSystemComponent file1 = new File("resume.pdf");
  FileSystemComponent file2 = new File("notes.txt");
  Directory folder = new Directory("Documents");
  folder.add(file1);
  folder.add(file2);

  Directory root = new Directory("Root");
  root.add(folder);
  root.showDetails();
 }
}
```

14.7 Python Implementation

Python allows a simpler approach due to duck typing:

```
class FileSystemComponent:
def show_details(self):
pass

class File(FileSystemComponent):
def __init__(self, name):
self.name = name
def show_details(self):
print(f"File: {self.name}")

class Directory(FileSystemComponent):
def __init__(self, name):
self.name = name
self.children = []
def add(self, component):
self.children.append(component)
def show_details(self):
print(f"Directory: {self.name}")
for child in self.children:
child.show_details()

file1 = File("resume.pdf")
file2 = File("notes.txt")
folder = Directory("Documents")
folder.add(file1)
folder.add(file2)

root = Directory("Root")
root.add(folder)
root.show_details()
```

14.8 Advantages

- **Uniformity:** Treats individual objects and groups of objects the same way.
- **Extensibility:** Adding new components (leaf or composite) requires no client-side changes.
- **Recursive simplicity:** Great for tree structures like file systems, GUIs, and game objects.

14.9 Drawbacks

- Makes designs more general than necessary, increasing complexity when hierarchies are shallow.
- Can make type-checking harder: distinguishing leaves from composites sometimes requires runtime checks.
- Overuse can result in large, deeply nested structures that are hard to manage.

14.10 Real-World Use Cases

- **File Systems:** Directories containing files and subdirectories.
- **GUI Frameworks:** Windows, panels, and buttons all share a common interface.
- **Graphics Engines:** Scenes, groups, and shapes in rendering trees.
- **DOM Trees:** HTML elements represented as nested composites.

14.11 Composite vs. Decorator vs. Proxy

- **Composite:** Models part-whole hierarchies and treats objects uniformly.
- **Decorator:** Dynamically adds functionality to individual objects without affecting others.
- **Proxy:** Controls access to an object but doesn't represent hierarchies.

14.12 Summary

- The **Composite Pattern** unifies the treatment of individual objects and groups of objects.
- Well-suited for hierarchical or recursive structures.
- Commonly used in file explorers, GUI frameworks, DOM representations, and rendering engines.
- Complements patterns like **Decorator** and **Proxy** within structural design.

Composite — Interview Blurb

Lets clients treat individual objects and groups of objects uniformly by organizing them into tree structures. Each node implements a common interface, enabling recursive composition where operations on a composite delegate to their children. This simplifies client code and supports scalable, hierarchical object models.

15 Flyweight Pattern

15.1 Learning Objectives

- Understand the intent and purpose of the **Flyweight** pattern.
- Learn how to optimize memory by sharing immutable, reusable objects.
- Examine the distinction between **intrinsic** and **extrinsic** state.
- Study UML structure and walk through Java and Python implementations.
- Explore advantages, trade-offs, and real-world applications.

15.2 Introduction

The **Flyweight Pattern** is a **structural** design pattern that improves efficiency when a large number of similar objects are needed. It achieves this by:

- Storing **shared, immutable state** in a common object (the flyweight).
- Delegating **unique, external state** to the client.

Definition (GoF): "Use sharing to support large numbers of fine-grained objects efficiently."

15.3 Motivation

Consider a text editor rendering thousands of characters:

- Each character has immutable properties like font, size, and weight.
- Storing these properties in every `Character` object wastes memory.
- Instead, we share one `CharacterFlyweight` per unique character style and store position or formatting externally.

This pattern is widely used when **many objects share common, immutable attributes**.

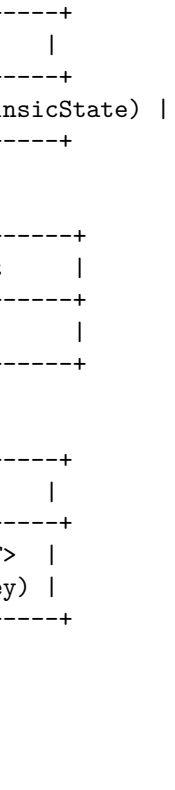
15.4 Key Concepts

- **Flyweight:** A shared, reusable object containing intrinsic, immutable state.
- **Intrinsic State:** Information stored inside the flyweight; independent of context (e.g., font style, image sprite).
- **Extrinsic State:** Context-specific data provided by the client at runtime (e.g., screen position, current health).
- **Flyweight Factory:** Manages the creation and sharing of flyweight objects.
- **Client:** Supplies extrinsic state when using flyweights.

15.5 When to Use Flyweight

- When an application uses a large number of objects.
- When object storage is costly in terms of memory.
- When most object attributes are shared and immutable.
- When performance demands require minimizing memory usage.

15.6 UML Structure



15.7 Java Implementation

Step 1: Define the Flyweight interface.

```
public interface Shape {
 void draw(int x, int y);
}
```

Step 2: Implement the ConcreteFlyweight.

```
public class Circle implements Shape {
 private final String color; // intrinsic state, shared

 public Circle(String color) {
  this.color = color;
 }

 @Override
 public void draw(int x, int y) {
  System.out.println("Drawing " + color +
  " circle at (" + x + ", " + y + ")");
 }
}
```

Step 3: Create the Flyweight Factory.

```
import java.util.HashMap;
import java.util.Map;

public class ShapeFactory {
```

```
 private static final Map<String, Shape> shapes = new HashMap<>();

 public static Shape getCircle(String color) {
  Shape circle = shapes.get(color);
  if (circle == null) {
   circle = new Circle(color);
   shapes.put(color, circle);
   System.out.println("Created new circle of color " + color);
  }
  return circle;
 }
}
```

Step 4: Use Flyweights in the Client.

```
public class Main {
 public static void main(String[] args) {
  for (int i = 0; i < 5; i++) {
   Shape red = ShapeFactory.getCircle("red");
   red.draw(i, i);

   Shape blue = ShapeFactory.getCircle("blue");
   blue.draw(i, i + 1);
  }
 }
}
```

15.8 Python Implementation

Python supports Flyweight naturally with dictionaries:

```
class Circle:
def __init__(self, color):
self.color = color  # intrinsic state

def draw(self, x, y):
print(f"Drawing {self.color} circle at ({x},{y})")

class ShapeFactory:
_circles = {}

@staticmethod
def get_circle(color):
if color not in ShapeFactory._circles:
ShapeFactory._circles[color] = Circle(color)
print(f"Created new circle of color {color}")
return ShapeFactory._circles[color]

for i in range(5):
red = ShapeFactory.get_circle("red")
red.draw(i, i)

blue = ShapeFactory.get_circle("blue")
blue.draw(i, i + 1)
```

15.9 Advantages

- Significant **memory savings** when many objects share state.

- Flyweights are immutable, making them naturally **thread-safe**.
- Centralized control via a **Flyweight Factory**.
- Improves performance in scenarios with large-scale object creation.

15.10 Drawbacks

- Clients must manage **extrinsic state**, increasing complexity.
- Sharing flyweights may reduce flexibility for objects requiring unique states.
- Overuse can lead to confusing designs when object distinctions blur.

15.11 Real-World Use Cases

- **Text Rendering:** Each character glyph is shared; extrinsic data includes position, size.
- **Game Engines:** Thousands of sprites or trees in a forest share models/textures.
- **GUI Widgets:** Shared look-and-feel styles across buttons, menus, and labels.
- **Caching Systems:** Avoid recreating identical immutable objects repeatedly.

15.12 Flyweight vs. Singleton vs. Prototype

- **Flyweight:** Shares objects to minimize memory; supports many logical instances.
- **Singleton:** Only one global instance exists.
- **Prototype:** Creates copies of an object rather than sharing a single one.

15.13 Summary

- The **Flyweight Pattern** reduces memory usage by sharing immutable objects.
- Separates **intrinsic state** (shared) from **extrinsic state** (context-dependent).
- Useful when working with large numbers of fine-grained objects.
- Widely used in graphics rendering, GUIs, caching systems, and large-scale simulations.

Flyweight — Interview Blurb

Reduces memory footprint by sharing immutable, intrinsic state among many fine-grained objects. Extrinsic (context-specific) data is supplied externally when needed, allowing thousands of lightweight objects to be represented efficiently without duplicating shared state.

16 Bridge Pattern

16.1 Learning Objectives

- Understand the intent and purpose of the **Bridge** pattern.
- Learn how to decouple an abstraction from its implementation.
- Explore the UML structure and analyze Java and Python implementations.
- Compare Bridge with related patterns like Adapter and Strategy.
- Examine real-world use cases, advantages, and trade-offs.

16.2 Introduction

The **Bridge Pattern** is a **structural** design pattern that separates an abstraction from its implementation, enabling them to evolve independently.

> *Definition (GoF):* "Decouple an abstraction from its implementation so that the two can vary independently."

It is especially useful in systems where:

- We want to avoid a rigid inheritance hierarchy.
- We need to combine multiple dimensions of variation (e.g., shapes + rendering APIs).

16.3 Motivation

Consider designing a graphics library:

- We have multiple **shapes** (circle, rectangle, triangle).
- We also have multiple **rendering backends** (OpenGL, DirectX, Vulkan).
- A naïve inheritance model would require one class for every combination: `OpenGLCircle`, `DirectXCircle`, `VulkanCircle`, etc.
- Adding a new shape or backend multiplies the number of required classes.

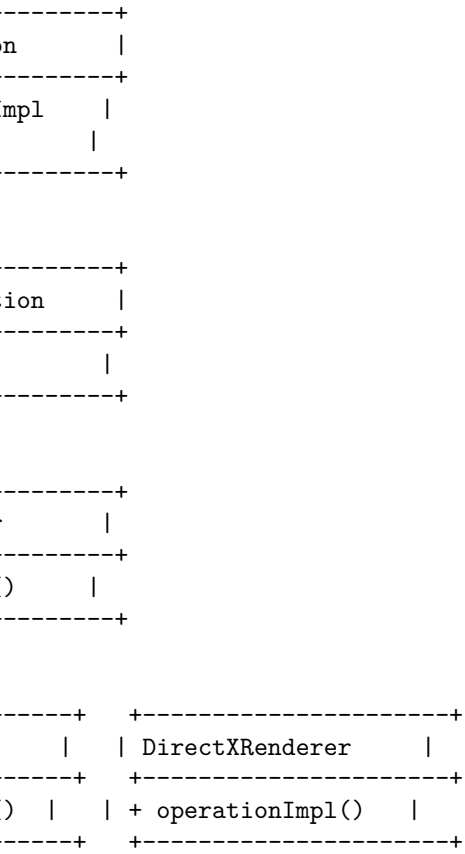
The Bridge Pattern solves this by:

- Keeping **shapes** as one abstraction hierarchy.
- Keeping **rendering implementations** separate.
- Connecting them via composition rather than inheritance.

16.4 Key Concepts

- **Abstraction:** Defines the high-level interface for clients (e.g., `Shape`).
- **Refined Abstraction:** Extends the abstraction with more specific behaviors (e.g., `Circle`).
- **Implementor:** Defines the lower-level interface for implementations (e.g., `Renderer`).
- **Concrete Implementor:** Provides concrete implementation logic (e.g., `OpenGLRenderer`).
- **Client:** Interacts only with the abstraction, unaware of the underlying implementation.

16.5 UML Structure



16.6 Java Implementation

Step 1: Define the Implementor interface.

```
public interface Renderer {
 void renderShape(String shape);
}
```

Step 2: Provide Concrete Implementations.

```
public class OpenGLRenderer implements Renderer {
 public void renderShape(String shape) {
  System.out.println("Drawing " + shape + " using OpenGL.");
 }
}

public class DirectXRenderer implements Renderer {
 public void renderShape(String shape) {
  System.out.println("Drawing " + shape + " using DirectX.");
 }
}
```

Step 3: Define the Abstraction.

```
public abstract class Shape {
 protected Renderer renderer;
 protected Shape(Renderer renderer) {
  this.renderer = renderer;
```

```
 }
 public abstract void draw();
}
```

Step 4: Create Refined Abstractions.

```
public class Circle extends Shape {
 public Circle(Renderer renderer) { super(renderer); }
 public void draw() {
  renderer.renderShape("Circle");
 }
}

public class Square extends Shape {
 public Square(Renderer renderer) { super(renderer); }
 public void draw() {
  renderer.renderShape("Square");
 }
}
```

Step 5: Using the Bridge.

```
public class Main {
 public static void main(String[] args) {
  Renderer opengl = new OpenGLRenderer();
  Renderer directx = new DirectXRenderer();

  Shape circle = new Circle(opengl);
  Shape square = new Square(directx);

  circle.draw();   // OpenGL rendering
  square.draw();   // DirectX rendering
 }
}
```

16.7 Python Implementation

Python makes the composition model very concise:

```
class Renderer:
def render_shape(self, shape):
pass

class OpenGLRenderer(Renderer):
def render_shape(self, shape):
print(f"Drawing {shape} using OpenGL.")

class DirectXRenderer(Renderer):
def render_shape(self, shape):
print(f"Drawing {shape} using DirectX.")

class Shape:
def __init__(self, renderer):
self.renderer = renderer
def draw(self):
pass
```

```
class Circle(Shape):
def draw(self):
self.renderer.render_shape("Circle")

class Square(Shape):
def draw(self):
self.renderer.render_shape("Square")

opengl = OpenGLRenderer()
directx = DirectXRenderer()

circle = Circle(opengl)
square = Square(directx)

circle.draw()   # OpenGL rendering
square.draw()   # DirectX rendering
```

16.8 Advantages

- Decouples abstractions from implementations, improving flexibility.
- Prevents **class explosion** caused by combining multiple dimensions of variation.
- Supports Open/Closed Principle: add new abstractions or implementations independently.
- Encourages composition over inheritance.

16.9 Drawbacks

- Introduces more classes and increases initial design complexity.
- Requires careful understanding of intrinsic and extrinsic responsibilities.
- Slightly more boilerplate code compared to tightly coupled inheritance.

16.10 Real-World Use Cases

- **GUI Frameworks:** Separating widgets from rendering APIs.
- **Cross-Platform Libraries:** Decoupling business logic from OS-specific implementations.
- **Device Drivers:** Unified abstractions for hardware controlled by different backends.
- **Reporting Systems:** Separating report generation from data formatting and rendering.

16.11 Bridge vs. Adapter vs. Strategy

- **Bridge:** Decouples abstractions from implementations so both evolve independently.
- **Adapter:** Makes an existing implementation work with a different expected interface.
- **Strategy:** Selects an algorithm dynamically; Bridge separates the implementation itself.

16.12 Summary

- The **Bridge Pattern** separates abstractions from implementations, avoiding deep inheritance hierarchies.
- Uses composition to connect higher-level abstractions to interchangeable implementation details.
- Useful when multiple dimensions of variation exist.
- Commonly applied in GUIs, rendering engines, device control, and cross-platform architectures.

Bridge — Interview Blurb

Decouples an abstraction from its implementation so that each can vary independently. Instead of binding an interface to one hierarchy of implementations, the Bridge pattern introduces a stable abstraction layer that delegates to a separate implementation interface, avoiding class explosion and enabling flexible composition of features and platforms.

Bridge vs. Adapter

Bridge and **Adapter** both decouple abstractions from implementations, but they do so at different times and for different reasons:

- **Intent:** Bridge is designed *up front* to separate abstraction and implementation so both can evolve independently. Adapter is applied *afterward* to make existing, incompatible interfaces work together.
- **Direction:** Bridge unifies two evolving hierarchies (abstraction $\leftrightarrow$ implementation). Adapter converts one existing interface to another expected by clients.
- **Analogy:** Bridge = *designed flexibility*, Adapter = *retrofitted compatibility.*

17 Structural Patterns Quiz with Answers & Explanations

This quiz reviews key concepts related to structural design patterns, principles, and heuristics. For each question, the correct answers are highlighted and explained.

Q1. Properties of the Decorator Pattern

Which of the following statements are TRUE? (Select all correct)

- ✗ Decorator supports the Open-Closed principle: **open for modification but closed to extension**
- ✓ Decorators are a subclass of the component being decorated **and** the decorator has a reference back to the decorated component
- ✓ Decorator lets you assign extra behaviors to objects at **runtime** without changing the base code for those objects
- ✓ Decorators should be considered whenever there are a variety of optional methods that can extend an object's existing operations

Explanation: Decorators subclass the same interface as the component, wrap an instance of it, and allow adding behaviors dynamically at runtime. The first statement is wrong because the Open-Closed Principle actually means *open for extension, closed for modification.*

Q2. Façade Pattern

Which of the following statements is **NOT correct** regarding the Façade pattern?

- ✓ **Facades are particularly effective when you need access to all the methods from a subsystem**
- ✗ Façade patterns often encapsulate entire subsystems, not just a single class
- ✗ Facades generally support loose coupling by reducing a client's connections to both methods and classes
- ✗ The Gang of Four definition for the Façade pattern highlights making a subsystem easier to use

Explanation: Facades simplify subsystem usage by exposing only a few high-level operations. If you need **all** subsystem methods, you typically bypass the facade entirely.

Q3. Principle of Least Knowledge

Which of the following statements is **NOT correct**?

- ✗ The principle supports reducing coupling in designs, preventing change cascades
- ✗ The principle is also known as "talk only to your friends"
- ✓ **The principle leads to reduced dependencies, making it harder to maintain code**
- ✗ This principle is analogous to the Law of Demeter

Explanation: The principle **reduces dependencies** and **improves** maintainability, not the opposite. It encourages objects to communicate only with closely related components, preventing ripple effects.

Q4. Adapter Pattern

Which of the following statements is **NOT correct**?

- ✗ Versions of the adapter pattern can be created using interfaces or multiple class inheritance
- ✓ **Façades focus on conversion, adapters are more focused on simplification**
- ✗ Adapters prevent changes to existing code that is already dependent on an external resource like an API or library
- ✗ Adapters could be used when existing APIs change connectivity, method structures, or method execution order

Explanation: Adapters focus on **interface conversion**, not simplification. Facades simplify complex subsystems; adapters make otherwise incompatible interfaces work together.

Q5. Multiple Inheritance Heuristics

Which of the following are TRUE heuristics? (Select all correct)

- ✓ If the thing I'm inheriting from is **part of me**, multiple inheritance might be OK
- ✓ Assume multiple inheritance is a **mistake unless proven otherwise**
- ✗ If the base class is really a derived class of another base class, multiple inheritance might be OK
- ✓ If I am a **special type** of the thing I'm inheriting from, multiple inheritance might be OK

Explanation: Multiple inheritance can introduce complexity. Use it cautiously and only when there's a clear "is-a" or "has-a" relationship.

Q6. Proxy Pattern Definitions

Which of the following are correct proxy definitions? (Select all correct)

- ✗ Protection Proxy: A proxy that includes additional actions when subjects are referenced, such as reference counting
- ✗ Smart Reference Proxy: A proxy that controls access to subjects or subject methods
- ✓ **Virtual Proxy: A proxy that only creates an expensive object or operation when it is needed**
- ✓ **Remote Proxy: A proxy that acts as a local interface to a networked subject via a messaging interface**

Explanation: - A **Virtual Proxy** defers object creation until necessary. - A **Remote Proxy** represents objects located elsewhere (e.g., networked services). - A **Protection Proxy** manages permissions, not reference counting. - A **Smart Reference Proxy** manages additional actions (like logging or caching), not access control.

Q7. Composite Pattern

Which principle was in question given the structure and functionality of the Composite pattern?

- ✗ Principle of Least Knowledge
- ✗ Single Responsibility
- ✓ **Open-Closed**
- ✗ Prefer Delegation over Inheritance

Explanation: The Composite pattern can violate OCP because adding new component types often requires modifying existing code.

Q8. Flyweight Pattern

In the Flyweight pattern example, the `TreeType` class is designed to handle:

- ✗ Extrinsic data – it is unique to individual objects and accessed for changes by clients
- ✗ A client interface for applying methods to aggregates of trees
- ✓ **Intrinsic data – it is used by multiple small objects, and it is not duplicated once defined**
- ✗ A factory for creating tree objects

Explanation: The `TreeType` stores shared, immutable, intrinsic data (e.g., texture, color, model). The client provides extrinsic data like position or size.

Q9. Bridge Pattern

Which of the following statements are TRUE? (Select all correct)

- ✗ Bridge does not support the OO principle of favoring delegation/aggregation over inheritance
- ✓ **Bridge allows abstract objects to decouple implementation of operations, allowing both the abstract objects and the operation implementations to vary in a scalable way**
- ✓ **Bridge supports the OO principle of encapsulating what varies**
- ✗ The Strategy and Bridge patterns have no significant differences

Explanation: Bridge **promotes composition over inheritance**, separates abstraction from implementation, and supports encapsulating variability. It differs from Strategy: Bridge focuses on structural decoupling, while Strategy focuses on dynamic behavioral changes.

Part V
Creational Design Patterns

18 Factory Pattern

18.1 Learning Objectives

- Understand the intent and purpose of the **Factory** pattern.
- Learn how factories decouple object creation from client code.
- Explore variations: **Simple Factory** vs. **Factory Method**.
- Examine UML structures and analyze Java/Python implementations.
- Compare Factory with other creational patterns like Abstract Factory, Builder, and Prototype.

18.2 Introduction

The **Factory Pattern** is a **creational** design pattern that deals with object creation. Rather than instantiating classes directly with the `new` keyword, we delegate the responsibility to a factory, which decides which subclass to instantiate.

> *Definition (GoF - Factory Method):* "Define an interface for creating an object but let subclasses decide which class to instantiate."

By encapsulating object creation logic, factories:

- Reduce coupling between client code and concrete implementations.
- Make the code more extensible and maintainable.
- Follow the Open/Closed Principle by allowing new product types without modifying existing code.

18.3 Motivation

Consider a pizza ordering system:

- A naïve approach would instantiate specific pizza classes directly in client code:

```
Pizza pizza = new CheesePizza();
```

- If we later introduce new pizza types, we must modify the client code.
- This violates the Open/Closed Principle and leads to tight coupling.

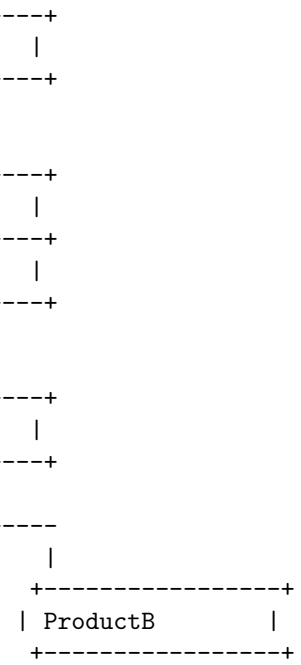
The solution: delegate object creation to a factory, which centralizes and abstracts the logic.

18.4 Types of Factory Patterns

- **Simple Factory:** Not technically a GoF pattern; uses a single factory class to return objects based on input parameters.
- **Factory Method:** Defines an interface for creating objects but lets subclasses decide which concrete class to return.

18.5 Simple Factory

18.5.1 UML Structure



18.5.2 Java Implementation

Step 1: Define the product interface.

```
public interface Pizza {
 void prepare();
}
```

Step 2: Create concrete products.

```
public class CheesePizza implements Pizza {
 public void prepare() {
  System.out.println("Preparing Cheese Pizza");
 }
}

public class VeggiePizza implements Pizza {
 public void prepare() {
  System.out.println("Preparing Veggie Pizza");
 }
}
```

Step 3: Implement the Simple Factory.

```
public class SimplePizzaFactory {
 public Pizza createPizza(String type) {
  if (type.equals("cheese")) return new CheesePizza();
  else if (type.equals("veggie")) return new VeggiePizza();
  else return null;
 }
}
```

Step 4: Use the factory in the client.

```
SimplePizzaFactory factory = new SimplePizzaFactory();
Pizza pizza = factory.createPizza("cheese");
pizza.prepare();
```

18.5.3 Python Implementation

```
class CheesePizza:
def prepare(self):
print("Preparing Cheese Pizza")

class VeggiePizza:
def prepare(self):
print("Preparing Veggie Pizza")

class SimplePizzaFactory:
def create_pizza(self, pizza_type):
if pizza_type == "cheese":
return CheesePizza()
elif pizza_type == "veggie":
return VeggiePizza()
return None

factory = SimplePizzaFactory()
pizza = factory.create_pizza("cheese")
pizza.prepare()
```

18.6 Factory Method

18.6.1 UML Structure

```
+---------------------+
|    Creator          |
+---------------------+
| + factoryMethod()   |
| + operation()       |
+---------------------+
^
|
+---------------------+
| ConcreteCreator     |
+---------------------+
| + factoryMethod()   |
+---------------------+
|
v
+---------------------+
|    Product          |
+---------------------+
^
-----------------------
|                      |
+----------------+   +-----------------+
| ProductA       |   | ProductB        |
+----------------+   +-----------------+
```

18.6.2 Java Implementation

Step 1: Define the product interface.

```
public interface Button {
 void render();
}
```

Step 2: Create concrete products.

```
public class WindowsButton implements Button {
 public void render() {
  System.out.println("Rendering a Windows button");
 }
}

public class MacOSButton implements Button {
 public void render() {
  System.out.println("Rendering a MacOS button");
 }
}
```

Step 3: Define the abstract creator.

```
public abstract class Dialog {
 public void renderWindow() {
  Button okButton = createButton();
  okButton.render();
 }
 public abstract Button createButton();
}
```

Step 4: Implement concrete creators.

```
public class WindowsDialog extends Dialog {
 public Button createButton() {
  return new WindowsButton();
 }
}

public class MacOSDialog extends Dialog {
 public Button createButton() {
  return new MacOSButton();
 }
}
```

Step 5: Use the factory in the client.

```
Dialog dialog = new WindowsDialog();
dialog.renderWindow();
```

18.6.3 Python Implementation

```
class Button:
def render(self):
pass

class WindowsButton(Button):
```

```
def render(self):
print("Rendering a Windows button")

class MacOSButton(Button):
def render(self):
print("Rendering a MacOS button")

class Dialog:
def render_window(self):
button = self.create_button()
button.render()

def create_button(self):
raise NotImplementedError

class WindowsDialog(Dialog):
def create_button(self):
return WindowsButton()

class MacOSDialog(Dialog):
def create_button(self):
return MacOSButton()

dialog = WindowsDialog()
dialog.render_window()
```

18.7 Advantages

- Centralizes and encapsulates object creation logic.
- Promotes loose coupling between clients and concrete classes.
- Makes it easier to add new products without modifying client code.
- Encourages adherence to the Open/Closed Principle.

18.8 Drawbacks

- Adds extra classes and abstractions, increasing complexity.
- May introduce slight performance overhead from delegation.

18.9 Real-World Use Cases

- **GUI Frameworks:** Creating platform-specific buttons, menus, and widgets.
- **Database Connections:** Factories for different database drivers (e.g., MySQL vs PostgreSQL).
- **Logging Systems:** Factories for console, file, or cloud-based loggers.
- **Document Processing:** Creating different parsers (e.g., XML vs JSON).

18.10 Factory vs. Abstract Factory vs. Builder

- **Factory Method:** Focuses on creating one type of product at a time via subclassing.
- **Abstract Factory:** Creates *families* of related products without specifying concrete classes.
- **Builder:** Focuses on step-by-step construction of complex objects.

18.11 Summary

- The **Factory Pattern** abstracts object creation, promoting flexibility and scalability.
- Reduces client coupling to specific implementations.
- Common in frameworks, libraries, and applications needing extensible product hierarchies.
- Serves as the foundation for more advanced creational patterns like **Abstract Factory**.

Factory — Interview Blurb

Encapsulates object creation logic in a dedicated method or class, separating instantiation from use. Clients depend only on abstract types while the factory decides which concrete implementation to create. This reduces coupling, simplifies testing, and supports the Open/Closed Principle by allowing new products to be introduced without modifying client code.

19 Abstract Factory Pattern

19.1 Learning Objectives

- Understand the intent and purpose of the **Abstract Factory** pattern.
- Learn how to create families of related objects without coupling client code to concrete implementations.
- Examine the UML structure and work through Java and Python implementations.
- Compare Abstract Factory with Factory Method and Builder patterns.
- Explore real-world use cases, advantages, and trade-offs.

19.2 Introduction

The **Abstract Factory Pattern** is a **creational** design pattern used to create **families of related objects** without specifying their concrete classes.

> *Definition (GoF):* "Provide an interface for creating families of related or dependent objects without specifying their concrete classes."

This pattern is particularly useful when:

- A system must be independent of how its objects are created.
- Products from the same family are designed to work together.
- We want to enforce consistency among related objects.

19.3 Motivation

Imagine designing a cross-platform GUI framework:

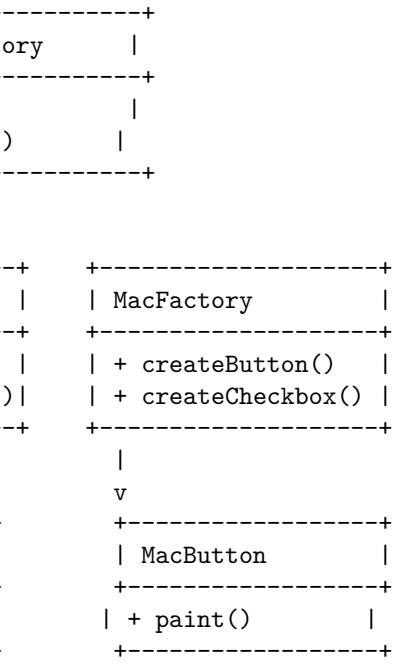
- We support multiple platforms: **Windows** and **MacOS**.
- Each platform has its own implementation of `Button` and `Checkbox`.
- Without Abstract Factory, client code would need `if-else` checks everywhere to choose the right concrete classes.
- This creates tight coupling and makes adding new platforms difficult.

The Abstract Factory encapsulates platform-specific creation logic in one place, allowing clients to work with high-level interfaces instead of concrete classes.

19.4 Key Concepts

- **Abstract Factory:** Declares creation methods for each product in the family.
- **Concrete Factory:** Implements creation methods to produce specific platform/product variants.
- **Abstract Product:** Defines a common interface for a product type.
- **Concrete Product:** Implements the abstract product interface for a specific variant.
- **Client:** Uses the abstract factory to create products without knowing their concrete classes.

19.5 UML Structure



19.6 Java Implementation

Step 1: Define Abstract Products

```
public interface Button {
 void paint();
}

public interface Checkbox {
 void paint();
}
```

Step 2: Implement Concrete Products

```
public class WindowsButton implements Button {
 public void paint() {
  System.out.println("Rendering a Windows-style button");
 }
}

public class MacButton implements Button {
 public void paint() {
  System.out.println("Rendering a Mac-style button");
 }
}

public class WindowsCheckbox implements Checkbox {
 public void paint() {
  System.out.println("Rendering a Windows-style checkbox");
 }
}

public class MacCheckbox implements Checkbox {
```

```
 public void paint() {
  System.out.println("Rendering a Mac-style checkbox");
 }
}
```

Step 3: Define Abstract Factory

```
public interface GUIFactory {
 Button createButton();
 Checkbox createCheckbox();
}
```

Step 4: Implement Concrete Factories

```
public class WindowsFactory implements GUIFactory {
 public Button createButton() { return new WindowsButton(); }
 public Checkbox createCheckbox() { return new WindowsCheckbox(); }
}

public class MacFactory implements GUIFactory {
 public Button createButton() { return new MacButton(); }
 public Checkbox createCheckbox() { return new MacCheckbox(); }
}
```

Step 5: Use Abstract Factory in the Client

```
public class Application {
 private Button button;
 private Checkbox checkbox;

 public Application(GUIFactory factory) {
  button = factory.createButton();
  checkbox = factory.createCheckbox();
 }

 public void render() {
  button.paint();
  checkbox.paint();
 }

 public static void main(String[] args) {
  GUIFactory factory = new WindowsFactory();
  Application app = new Application(factory);
  app.render();
 }
}
```

19.7 Python Implementation

```
class Button:
def paint(self): pass

class Checkbox:
def paint(self): pass

class WindowsButton(Button):
def paint(self):
print("Rendering a Windows-style button")
```

```
class MacButton(Button):
def paint(self):
print("Rendering a Mac-style button")

class WindowsCheckbox(Checkbox):
def paint(self):
print("Rendering a Windows-style checkbox")

class MacCheckbox(Checkbox):
def paint(self):
print("Rendering a Mac-style checkbox")

class GUIFactory:
def create_button(self): pass
def create_checkbox(self): pass

class WindowsFactory(GUIFactory):
def create_button(self): return WindowsButton()
def create_checkbox(self): return WindowsCheckbox()

class MacFactory(GUIFactory):
def create_button(self): return MacButton()
def create_checkbox(self): return MacCheckbox()

# Client code
factory = WindowsFactory()
button = factory.create_button()
checkbox = factory.create_checkbox()
button.paint()
checkbox.paint()
```

19.8 Advantages

- Ensures consistency among families of related objects.
- Decouples client code from concrete product classes.
- Makes it easy to introduce new product families without modifying client code.
- Encourages scalability and flexibility in large applications.

19.9 Drawbacks

- Introduces additional complexity with multiple interfaces and classes.
- Adding a new product type (e.g., `Slider`) requires updating all factories.
- Can be overkill for small systems with few variations.

19.10 Real-World Use Cases

- **Cross-Platform GUI Frameworks:** Qt, Java Swing, Android View Factories.
- **Database Drivers:** Abstract factories for database connections across vendors.
- **Testing Environments:** Switching between mock and production implementations.
- **Cloud SDKs:** Managing service-specific implementations for multiple cloud providers.

19.11 Abstract Factory vs. Factory Method vs. Builder

- **Factory Method:** Focuses on creating one product at a time; subclasses choose which implementation.
- **Abstract Factory:** Produces *families* of related products; enforces consistency.
- **Builder:** Focuses on assembling complex objects step-by-step rather than producing interchangeable families.

19.12 Summary

- The **Abstract Factory Pattern** provides an interface for creating families of related objects.
- It promotes consistency, decoupling, and scalability in applications.
- Especially valuable in frameworks and systems where multiple implementations must coexist.
- Complements other creational patterns such as Factory Method and Builder.

Abstract Factory — Interview Blurb

Provides an interface for creating families of related or dependent objects without specifying their concrete classes. Each concrete factory produces a consistent set of products that work together, enabling configuration-level flexibility and maintaining encapsulation of creation logic.

Factory vs. Abstract Factory

- **Factory Method:** Focuses on creating one product at a time. Subclasses decide which concrete implementation to instantiate, letting the client depend only on the abstract product type.
- **Abstract Factory:** Creates *families of related products* that are designed to work together (e.g., matching UI components or themed widgets). Provides a higher-level interface that returns factories rather than single instances.
- **Analogy:** Factory Method = *one product at a time*; Abstract Factory = *an entire product line or kit.*
- **Design Intent:** Both hide creation details, but Abstract Factory emphasizes *consistency across product families*, while Factory Method emphasizes *polymorphic creation of a single type.*

20 Singleton Pattern

20.1 Learning Objectives

- Understand the intent and purpose of the **Singleton** pattern.
- Learn how to ensure that a class has only one instance.
- Study the UML structure and analyze Java and Python implementations.
- Explore advantages, trade-offs, and real-world applications.
- Compare Singleton with related creational patterns such as Factory and Flyweight.

20.2 Introduction

The **Singleton Pattern** is a **creational** design pattern that ensures a class has **only one instance** and provides a global point of access to that instance.

> *Definition (GoF):* "Ensure a class has only one instance, and provide a global point of access to it."

Singletons are useful when:

- Exactly one object is required to coordinate actions across the system.
- Centralized control is necessary for managing shared resources.
- Multiple objects would cause conflicts or unnecessary resource usage.

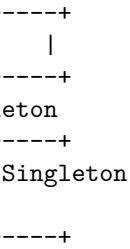
20.3 Motivation

Consider a logging framework:

- All parts of the application need to write logs.
- If each module instantiated its own logger, it could lead to inconsistent file handling.
- A better solution: a **Singleton Logger** that ensures one shared instance across the application.

Other typical examples include database connections, thread pools, configuration managers, and caches.

20.4 UML Structure



Key points:

- The Singleton class has a `private static` instance.
- A `public static` method returns the single instance.
- The constructor is private to prevent external instantiation.

20.5 Java Implementation

20.5.1 Basic Singleton

```
public class Singleton {
 private static Singleton instance;

 private Singleton() { }  // private constructor

 public static Singleton getInstance() {
  if (instance == null) {
   instance = new Singleton();
  }
  return instance;
 }

 public void doSomething() {
  System.out.println("Singleton in action!");
 }
}
```

Usage:

```
Singleton singleton = Singleton.getInstance();
singleton.doSomething();
```

20.5.2 Thread-Safe Singleton

The above implementation is **not thread-safe**. In multithreaded environments, use **synchronized**:

```
public class Singleton {
 private static Singleton instance;

 private Singleton() { }

 public static synchronized Singleton getInstance() {
  if (instance == null) {
   instance = new Singleton();
  }
  return instance;
 }
}
```

20.5.3 Double-Checked Locking (Efficient Singleton)

```
public class Singleton {
 private static volatile Singleton instance;

 private Singleton() { }

 public static Singleton getInstance() {
  if (instance == null) {
   synchronized (Singleton.class) {
    if (instance == null) {
     instance = new Singleton();
    }
   }
```

```
  }
  return instance;
 }
}
```

20.5.4 Bill Pugh Singleton (Recommended)

This is the most efficient and recommended approach in Java:

```
public class Singleton {
 private Singleton() { }

 private static class SingletonHelper {
  private static final Singleton INSTANCE = new Singleton();
 }

 public static Singleton getInstance() {
  return SingletonHelper.INSTANCE;
 }
}
```

This approach is thread-safe, lazy-loaded, and avoids unnecessary synchronization overhead.

20.6 Python Implementation

Python naturally supports Singleton through module-level objects, but here's a class-based approach:

20.6.1 Simple Singleton

```
class Singleton:
_instance = None

def __new__(cls):
if cls._instance is None:
cls._instance = super().__new__(cls)
return cls._instance

def do_something(self):
print("Singleton in action!")

s1 = Singleton()
s2 = Singleton()
print(s1 is s2)  # True
```

20.6.2 Using a Decorator

```
def singleton(cls):
instances = {}
def get_instance(*args, **kwargs):
if cls not in instances:
instances[cls] = cls(*args, **kwargs)
return instances[cls]
return get_instance

@singleton
class Logger:
pass
```

```
logger1 = Logger()
logger2 = Logger()
print(logger1 is logger2)  # True
```

20.7 Advantages

- Ensures a single, consistent instance across the entire application.
- Provides controlled global access to shared resources.
- Can improve performance when expensive objects are reused.
- Lazy initialization ensures objects are created only when needed.

20.8 Drawbacks

- Can introduce **hidden dependencies** between classes.
- Difficult to test since global state is shared.
- May lead to violations of the Single Responsibility Principle (SRP).
- In multithreaded environments, extra care is required for synchronization.

20.9 Real-World Use Cases

- **Logging Systems:** Centralized loggers across modules.
- **Database Connections:** Single shared connection pool.
- **Configuration Managers:** Shared system-wide settings.
- **Caching Systems:** Centralized, reusable cache objects.
- **Thread Pools:** Managed, globally accessible worker pools.

20.10 Singleton vs. Factory vs. Flyweight

- **Singleton:** Ensures a single, globally accessible instance.
- **Factory:** Focuses on object creation but can return new or shared instances.
- **Flyweight:** Shares many lightweight objects, unlike Singleton which enforces one.

20.11 Summary

- The **Singleton Pattern** ensures a class has only one instance and provides a global point of access.
- Widely used for resource sharing, logging, caching, and configuration management.
- Must be implemented carefully in multithreaded environments.
- While powerful, Singleton should be used sparingly to avoid introducing global state issues.

Singleton — Interview Blurb

Ensures a class has only one instance and provides a global access point to it. Commonly used for shared resources such as configuration managers, caches, or loggers. The Singleton encapsulates its own lifecycle, preventing uncontrolled instantiation and ensuring consistent, centralized state across the application.

21 Prototype Pattern

21.1 Learning Objectives

- Understand the intent and purpose of the **Prototype** pattern.
- Learn how to create new objects by copying existing instances.
- Explore UML structure and Java/Python implementations.
- Compare Prototype with Factory, Builder, and Singleton.
- Examine real-world applications, advantages, and trade-offs.

21.2 Introduction

The **Prototype Pattern** is a **creational** design pattern that creates new objects by **cloning existing ones** rather than instantiating classes directly.

> *Definition (GoF):* "Specify the kinds of objects to create using a prototypical instance and create new objects by copying this prototype."

Instead of constructing an object from scratch, we reuse an existing, preconfigured object, which reduces cost and simplifies creation.

21.3 Motivation

Object creation can be:

- **Expensive:** Large initialization overhead (e.g., parsing XML configs or loading textures).
- **Complex:** Involving many interdependent components.
- **Dynamic:** The system must create objects at runtime based on state rather than compile-time class knowledge.

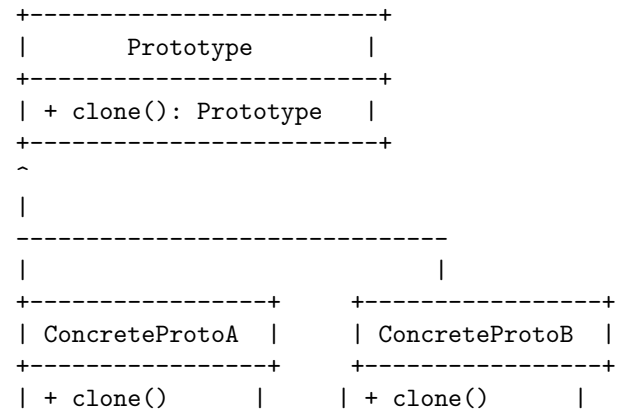
For example, in a video game:

- We may have a complex `Enemy` class with hundreds of properties.
- Instead of re-creating enemies from scratch, we store prototype instances and clone them efficiently.

21.4 Key Concepts

- **Prototype:** Declares an interface for cloning itself.
- **Concrete Prototype:** Implements the cloning logic.
- **Client:** Uses the prototype to create new objects without knowing their exact types.

21.5 UML Structure



```
+-----------------+     +-----------------+
^
|
+---------------+
|    Client     |
+---------------+
| uses clone()  |
+---------------+
```

21.6 Java Implementation

Step 1: Define the Prototype Interface

```
public interface Prototype {
 Prototype clone();
}
```

Step 2: Implement Concrete Prototypes

```
public class Circle implements Prototype {
 private int radius;
 private String color;

 public Circle(int radius, String color) {
  this.radius = radius;
  this.color = color;
 }

 @Override
 public Prototype clone() {
  return new Circle(this.radius, this.color);
 }

 public void draw() {
  System.out.println("Drawing " + color + " circle of radius " + radius);
 }
}
```

Step 3: Use the Prototype in the Client

```
public class Main {
 public static void main(String[] args) {
  Circle original = new Circle(5, "red");
  Circle copy = (Circle) original.clone();

  original.draw();
  copy.draw();
  System.out.println(original == copy); // false
 }
}
```

21.6.1 Alternative: Deep vs. Shallow Cloning

Java's built-in `clone()` performs a shallow copy by default:

- **Shallow Copy:** Copies primitive fields but shares object references.
- **Deep Copy:** Creates independent copies of referenced objects.

For deep cloning, manually clone nested objects or use libraries like Jackson, Gson, or Apache Commons.

21.7 Python Implementation

21.7.1 Using the copy Module

```
import copy

class Circle:
def __init__(self, radius, color):
self.radius = radius
self.color = color

def draw(self):
print(f"Drawing {self.color} circle of radius {self.radius}")

original = Circle(5, "red")
copy_circle = copy.deepcopy(original)

original.draw()
copy_circle.draw()
print(original is copy_circle)  # False
```

21.7.2 Custom clone() Method

```
class Circle:
def __init__(self, radius, color):
self.radius = radius
self.color = color

def clone(self):
return Circle(self.radius, self.color)

circle1 = Circle(10, "blue")
circle2 = circle1.clone()
print(circle1 is circle2)  # False
```

21.8 Advantages

- **Performance:** Efficient creation of costly objects.
- **Decoupling:** Clients don't need to know concrete class details.
- **Dynamic Instantiation:** Objects can be cloned at runtime without class awareness.
- **Reduces Subclassing:** Avoids large, rigid factories by reusing existing instances.

21.9 Drawbacks

- Cloning complex objects with nested references requires careful handling.
- Deep vs. shallow copying can lead to subtle bugs.
- Can be less intuitive than factories when object construction is simple.

21.10 Real-World Use Cases

- **Document Editors:** Copying styled components like text blocks or tables.
- **Game Engines:** Cloning entities (e.g., enemies, projectiles) for performance.
- **Graphics Systems:** Reusing preconfigured templates for shapes or sprites.
- **Machine Learning Models:** Copying trained models without reinitialization.

21.11 Prototype vs. Factory vs. Builder

- **Prototype:** Creates new objects by cloning existing instances.
- **Factory:** Creates new instances from scratch based on type.
- **Builder:** Constructs complex objects step by step.

21.12 Summary

- The **Prototype Pattern** creates new objects by cloning prototypes.
- Reduces overhead when instantiating objects with complex initialization.
- Supports shallow and deep copying based on requirements.
- Useful when runtime flexibility is needed and classes are expensive to initialize.

Prototype — Interview Blurb

Creates new objects by cloning existing prototypes instead of instantiating new classes. This allows flexible creation of objects with pre-set configurations or expensive initialization, avoiding tight coupling to concrete constructors and enabling runtime customization of prototypes.

22 Builder Pattern

22.1 Learning Objectives

- Understand the intent and purpose of the **Builder** pattern.
- Learn how to construct complex objects step by step.
- Explore the UML structure and analyze Java/Python implementations.
- Compare Builder with Factory, Abstract Factory, and Prototype.
- Examine real-world applications, advantages, and trade-offs.

22.2 Introduction

The **Builder Pattern** is a **creational** design pattern used to construct complex objects incrementally. Instead of requiring a large constructor with many parameters, the Builder separates the construction process from the representation.

> *Definition (GoF):* "Separate the construction of a complex object from its representation so that the same construction process can create different representations."

The pattern provides fine-grained control over the creation process and is especially useful when:

- The object has many optional or configurable parts.
- Different representations of the same type of object are needed.
- The construction process involves multiple steps.

22.3 Motivation

Imagine designing a pizza ordering system:

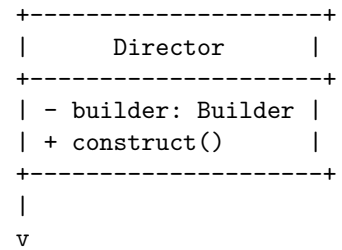
- A pizza has optional toppings, crust styles, sauces, and sizes.
- Using a constructor with 10–15 parameters would make the code unreadable and error-prone.
- The Builder lets you set only the desired attributes and construct the final object when ready.

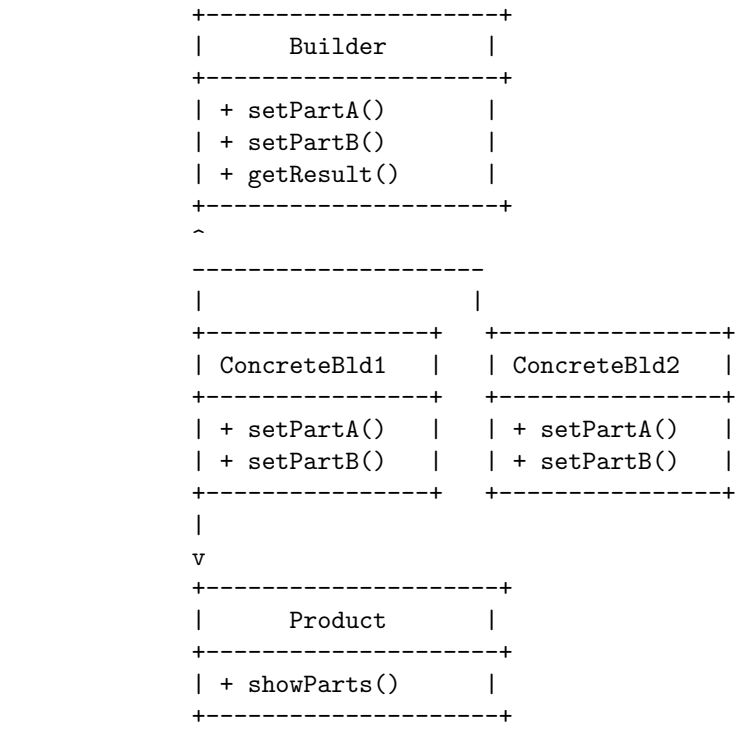
Similarly, Builder is used for constructing documents, GUI layouts, reports, and objects with nested dependencies.

22.4 Key Concepts

- **Builder:** Abstract interface defining the steps to build a product.
- **Concrete Builder:** Implements the steps defined in `Builder`.
- **Product:** The complex object being constructed.
- **Director (optional):** Orchestrates the construction steps in a specific order.
- **Client:** Configures the builder and retrieves the product.

22.5 UML Structure





22.6 Java Implementation

Step 1: Define the Product

```
public class Computer {
 private String CPU;
 private String GPU;
 private int RAM;
 private int storage;

 public Computer(String CPU, String GPU, int RAM, int storage) {
  this.CPU = CPU;
  this.GPU = GPU;
  this.RAM = RAM;
  this.storage = storage;
 }

 public String toString() {
  return "Computer [CPU=" + CPU + ", GPU=" + GPU +
  ", RAM=" + RAM + "GB, Storage=" + storage + "GB]";
 }
}
```

Step 2: Create the Builder Interface

```
public interface ComputerBuilder {
 ComputerBuilder setCPU(String cpu);
 ComputerBuilder setGPU(String gpu);
 ComputerBuilder setRAM(int ram);
 ComputerBuilder setStorage(int storage);
 Computer build();
}
```

22.6.1 Step 3: Implement the Concrete Builder

```
public class GamingComputerBuilder implements ComputerBuilder {
 private String CPU;
 private String GPU;
 private int RAM;
 private int storage;

 GamingComputerBuilder() {}

 @Override
 public ComputerBuilder setCPU(String cpu) {
  this.CPU = cpu;
  return this;
 }

 @Override
 public ComputerBuilder setGPU(String gpu) {
  this.GPU = gpu;
  return this;
 }

 @Override
 public ComputerBuilder setRAM(int ram) {
  this.RAM = ram;
  return this;
 }

 @Override
 public ComputerBuilder setStorage(int storage) {
  this.storage = storage;
  return this;
 }

 @Override
 public Computer build() {
  return new Computer(CPU, GPU, RAM, storage);
 }
}
```

Step 4: Use the Builder

```
public class Main {
 public static void main(String[] args) {
  Computer gamingPC = new GamingComputerBuilder()
  .setCPU("Intel i9")
  .setGPU("NVIDIA RTX 4080")
  .setRAM(32)
  .setStorage(2000)
  .build();

  System.out.println(gamingPC);
 }
}
```

22.7 Python Implementation

```
class Computer:
def __init__(self, cpu=None, gpu=None, ram=0, storage=0):
self.cpu = cpu
self.gpu = gpu
self.ram = ram
self.storage = storage

def __str__(self):
return f"Computer [CPU={self.cpu}, GPU={self.gpu}, RAM={self.ram}GB, Storage={self.storage}GB]"

class ComputerBuilder:
def __init__(self):
self.cpu = None
self.gpu = None
self.ram = 0
self.storage = 0

def set_cpu(self, cpu):
self.cpu = cpu
return self

def set_gpu(self, gpu):
self.gpu = gpu
return self

def set_ram(self, ram):
self.ram = ram
return self

def set_storage(self, storage):
self.storage = storage
return self

def build(self):
return Computer(self.cpu, self.gpu, self.ram, self.storage)

# Client code
gaming_pc = (ComputerBuilder()
.set_cpu("Intel i9")
.set_gpu("NVIDIA RTX 4080")
.set_ram(32)
.set_storage(2000)
.build())

print(gaming_pc)
```

22.8 Advantages

- Simplifies the creation of complex objects.
- Reduces the need for large constructors with many parameters.
- Improves readability and flexibility via method chaining.
- Supports different representations of the same object.

22.9 Drawbacks

- Introduces extra classes (builder, director, etc.), increasing complexity.
- Less efficient than directly constructing objects when they are simple.
- Not always necessary for small, lightweight products.

22.10 Real-World Use Cases

- **UI Layout Managers:** Constructing GUIs dynamically.
- **Document Generation:** Building reports, resumes, or PDFs step by step.
- **Game Development:** Creating characters or worlds with configurable attributes.
- **Test Data Builders:** Simplifying object creation in unit tests.

22.11 Builder vs. Factory vs. Prototype

- **Builder:** Constructs complex objects step by step, supporting many configurations.
- **Factory Method / Abstract Factory:** Focus on instantiating objects but don't manage stepwise construction.
- **Prototype:** Creates new objects by cloning existing ones.

22.12 Summary

- The **Builder Pattern** separates the construction of a complex object from its representation.
- Supports controlled, stepwise object creation with optional configurations.
- Useful when objects have many parameters or representations.
- Complements other creational patterns, especially when combined with Abstract Factory.

Builder — Interview Blurb

Separates the construction of a complex object from its representation, so the same process can create different forms or configurations. Encapsulates step-by-step assembly logic, improving readability, testability, and flexibility in object creation—especially for objects with many optional or interdependent parameters.

Builder vs. Factory

- **Intent:** Both encapsulate object creation, but they target different problems. **Factory** decides *which* concrete object to create; **Builder** focuses on *how* to assemble a complex object step by step.
- **Granularity:** Factory produces a fully constructed object in one call, while Builder constructs an object incrementally through chained or staged methods.
- **Flexibility:** Builder separates construction from representation, allowing different configurations using the same creation process; Factory emphasizes polymorphic substitution.
- **Analogy:** Factory = "pick a product from the catalog." Builder = "assemble a custom product piece by piece."

23 Creational Patterns Quiz with Answers & Explanations

This quiz reviews key concepts for creational design patterns: Factory, Abstract Factory, Singleton, Builder, Prototype, and related principles.

Q1. Factory Pattern

Which of the following are TRUE regarding the Factory pattern? (Select all correct)

- ✓ **Factory isolates instantiation and makes it easier to make new classes, remove classes, or change the creation-focused code**
- ✗ The Factory pattern does not allow created products to come from a prototype, it requires the use of the `new` command for new object instances
- ✓ **Factories create objects via inheritance; the UML class diagram for Factory shows the parallel inheritance for the creators and the products created**
- ✓ **When we directly instantiate a class using a `new` command (e.g., `Duck duck = new MallardDuck();`), we break encapsulation by type**

Explanation: The Factory pattern centralizes object creation and supports extensibility by removing hardcoded `new` calls. However, the statement about prototypes is incorrect — Factories may delegate creation to prototypes if needed.

Q2. Dependency Inversion Principle

Which of the following best describes the Dependency Inversion Principle?

- ✗ High-level classes must depend on low-level classes
- ✗ Low-level classes must depend on high-level classes
- ✗ Low- and high-level classes should have mutual dependency
- ✓ **Both low-level and high-level classes should depend on abstractions**

Explanation: The Dependency Inversion Principle (DIP) encourages depending on abstractions, not concrete implementations. This decouples high-level modules from low-level details.

Q3. Abstract Factory Pattern

Which of the following are TRUE for the Abstract Factory pattern? (Select all correct)

- ✓ **Abstract Factory creates families of related objects via object composition**
- ✗ Unlike Factory, Abstract Factory does not support the Dependency Inversion principle
- ✓ **Abstract Factory provides an interface for creating composed objects without specifying concrete classes**
- ✓ **Looking at a UML Class Diagram for Abstract Factory, we can see it is composed of a set of Factory patterns**

Explanation: Abstract Factory defines an interface for creating families of related products. It supports DIP and internally uses multiple Factory methods to produce different products.

Q4. Singletons and Object Pools

Which of the following statements regarding Singletons and Object Pools is **NOT correct**?

- ✗ In the lazy instantiation version of Singleton, we may want to use the `synchronized` keyword in Java to prevent two threads from creating multiple new Singletons inadvertently
- ✗ An object pool allows for multiple instances of an object to be kept in a pool collection; users of the objects can acquire them (if available) or release them back to the pool
- ✗ Singleton can be created using eager instantiation, creating the single object before it is used and before any threading issues might occur
- ✓ **Singleton's UML class diagram is drawn with an abstract class inheriting to a concrete class; the abstract class shows the instance of the Singleton, its private constructor, and a method to get the single instance**

Explanation: This last statement is incorrect: Singleton diagrams typically **do not** require an abstract class at all. They depict a **single concrete class** with: - A private constructor - A static instance reference - A static accessor (e.g., `getInstance()`).

Q5. Object Creation/Management Rule

Regarding the Object Creation/Management Rule, which of the following is **NOT correct**?

- ✗ Classes (or objects at runtime) should either create or manage/use objects, not both
- ✗ The rule does not hold for factory objects, which need to know both how to create objects and how to use them
- ✗ The rule is really a guideline; in some cases it may not be possible or advantageous to separate the creation and use in the class structure
- ✓ **The rule supports changes to encapsulated creation functionality without impacting other code that uses the created objects**

Explanation: This last statement is incorrect — encapsulated creation allows internal factory logic to change freely without affecting dependent code.

Q6. Prototype Pattern

The Prototype pattern uses a _____ of an existing object vs. a new instantiation. In performing this operation, designers must consider whether a _____ of primitive and immutable data is sufficient, or whether a _____ including object references and possibly other copied data may be needed.

Answer:

- Clone
- Shallow copy
- Deep copy

Explanation: The Prototype pattern clones existing objects instead of instantiating them from scratch. - **Shallow copy:** Duplicates only primitive and immutable fields. - **Deep copy:** Recursively duplicates referenced objects to produce a fully independent instance.

Q7. Builder Pattern

Considering the Builder pattern, which of the following is **NOT correct**?

- ✗ Builder helps create complex objects that are independent of their parts and assembly
- ✗ Builder encapsulates construction of an object into a series of steps
- ✓ **Builder always requires all component parts to be specified via method calls, otherwise the build of the object will fail**

✗ The style of multiple chained method calls seen with Builder implementations is called a **fluent interface**

Explanation: Builder does **not** require all components to be specified; defaults or partial configurations are often supported. It's ideal for creating complex objects step by step and supports fluent interfaces.

Part VI
Behavioral Design Patterns

24 Command Pattern

24.1 Learning Objectives

- Understand the intent and purpose of the **Command** pattern.
- Learn how to encapsulate actions as objects to decouple senders from receivers.
- Explore the UML structure and study Java/Python implementations.
- Compare Command with Strategy, Mediator, and Observer patterns.
- Examine real-world applications, advantages, and trade-offs.

24.2 Introduction

The **Command Pattern** is a **behavioral** design pattern that encapsulates a request or action as an object, thereby decoupling the invoker (sender) from the receiver (executor).

> *Definition (GoF):* "Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue them, log them, and support undoable operations."

This pattern is particularly useful in applications where:

- Actions need to be parameterized or scheduled.
- Requests should be stored, queued, logged, or replayed later.
- We need undo/redo functionality.

24.3 Motivation

Consider designing a smart home automation app:

- A user presses a button to turn on a light.
- Without Command, the button directly calls the light's `turnOn()` method.
- This tightly couples the UI control to a specific device implementation.

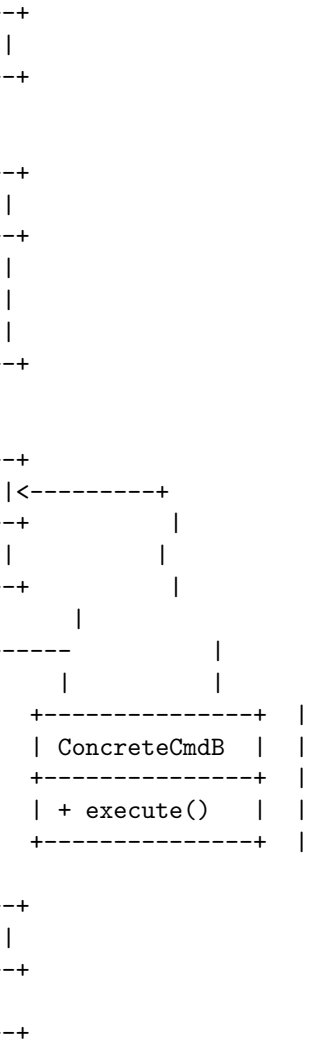
With the Command Pattern:

- The button invokes a `Command` object instead.
- The `Command` encapsulates the action and delegates to the correct receiver (e.g., Light).
- The same button can trigger any other action simply by swapping commands.

24.4 Key Concepts

- **Command:** Declares the interface for executing an action.
- **Concrete Command:** Implements the command by delegating to a receiver.
- **Receiver:** The object that performs the actual work.
- **Invoker:** Calls the command but is decoupled from its implementation.
- **Client:** Configures commands and associates them with invokers.

24.5 UML Structure



24.6 Java Implementation

Step 1: Define the Command Interface

```
public interface Command {
 void execute();
}
```

Step 2: Create the Receiver

```
public class Light {
 public void turnOn() {
  System.out.println("Light is ON");
 }

 public void turnOff() {
  System.out.println("Light is OFF");
 }
}
```

Step 3: Implement Concrete Commands

```
public class LightOnCommand implements Command {
 private Light light;

 public LightOnCommand(Light light) {
  this.light = light;
 }

 @Override
 public void execute() {
  light.turnOn();
 }
}

public class LightOffCommand implements Command {
 private Light light;

 public LightOffCommand(Light light) {
  this.light = light;
 }

 @Override
 public void execute() {
  light.turnOff();
 }
}
```

Step 4: Create the Invoker

```
public class RemoteControl {
 private Command command;

 public void setCommand(Command command) {
  this.command = command;
 }

 public void pressButton() {
  command.execute();
 }
}
```

Step 5: Use the Command in the Client

```
public class Main {
 public static void main(String[] args) {
  Light livingRoomLight = new Light();

  Command lightOn = new LightOnCommand(livingRoomLight);
  Command lightOff = new LightOffCommand(livingRoomLight);

  RemoteControl remote = new RemoteControl();
  remote.setCommand(lightOn);
  remote.pressButton();

  remote.setCommand(lightOff);
  remote.pressButton();
 }
}
```

24.7 Python Implementation

```
class Light:
def turn_on(self):
print("Light is ON")

def turn_off(self):
print("Light is OFF")

class Command:
def execute(self):
pass

class LightOnCommand(Command):
def __init__(self, light):
self.light = light

def execute(self):
self.light.turn_on()

class LightOffCommand(Command):
def __init__(self, light):
self.light = light

def execute(self):
self.light.turn_off()

class RemoteControl:
def __init__(self):
self.command = None

def set_command(self, command):
self.command = command

def press_button(self):
self.command.execute()

# Client code
light = Light()
on_command = LightOnCommand(light)
off_command = LightOffCommand(light)

remote = RemoteControl()
remote.set_command(on_command)
remote.press_button()  # Light is ON

remote.set_command(off_command)
remote.press_button()  # Light is OFF
```

24.8 Advantages

- **Decouples senders from receivers:** Invokers don't need to know how commands are executed.
- Enables easy implementation of **undo/redo** functionality.
- Supports logging, queuing, and scheduling of commands.
- Makes it easy to add new commands without modifying existing code.

24.9 Drawbacks

- Introduces additional classes, increasing code complexity.
- May cause performance overhead when dealing with very simple commands.
- Requires careful command management when many commands exist.

24.10 Real-World Use Cases

- **GUI Frameworks:** Button clicks mapped to commands.
- **Task Scheduling Systems:** Delayed job queues and event-driven systems.
- **Undo/Redo Systems:** Word processors, IDEs, and editors.
- **Macro Recording:** Record and replay sequences of actions.
- **Messaging Systems:** Asynchronous command queues in distributed architectures.

24.11 Command vs. Strategy vs. Observer

- **Command:** Encapsulates an action or request as an object.
- **Strategy:** Encapsulates an algorithm, letting clients swap implementations dynamically.
- **Observer:** Reacts to state changes; not directly responsible for invoking commands.

24.12 Summary

- The **Command Pattern** encapsulates actions as objects, decoupling senders from receivers.
- Useful for undo/redo operations, logging, scheduling, and macro recording.
- Widely used in GUI frameworks, distributed systems, and task orchestration platforms.
- A foundational behavioral pattern, often used alongside Observer, Mediator, and Strategy.

Command — Interview Blurb

Encapsulates a request as an object, decoupling the invoker from the receiver. This allows requests to be queued, logged, undone, or executed at different times without the invoker knowing the specifics of the action. Command objects package both the operation and its target, enabling flexible and reversible behaviors.

Command vs. Strategy

- **Intent: Command** turns requests into objects to enable queuing, logging, undo, scheduling. **Strategy** selects among interchangeable algorithms to vary behavior at runtime.
- **Participants: Command** introduces *Invoker*, *Command*, *Receiver*. **Strategy** introduces *Context* and *Strategy* family.
- **Axis of variability: Command** varies *which action is invoked and when*. **Strategy** varies *how an operation is performed*.
- **State / Undo: Command** often stores state to support *undo/redo*. **Strategy** is typically stateless or light-state and not about reversibility.
- **Client impact: Command** decouples the requestor from the performer and enables workflows. **Strategy** removes conditionals by delegating algorithm choice via composition.

25 State Pattern

25.1 Learning Objectives

- Understand the intent and purpose of the **State** pattern.
- Learn how to represent and manage object behavior that changes dynamically.
- Explore the UML structure and study Java/Python implementations.
- Compare State with Strategy, Command, and Observer patterns.
- Examine real-world applications, advantages, and trade-offs.

25.2 Introduction

The **State Pattern** is a **behavioral** design pattern that allows an object to alter its behavior when its **internal state** changes. From the client's perspective, the object appears to change its class.

> *Definition (GoF):* "Allow an object to alter its behavior when its internal state changes. The object will appear to change its class."

This pattern eliminates long conditional logic (e.g., `if-else` or `switch` statements) by encapsulating state-specific behaviors into separate classes.

25.3 Motivation

Imagine designing a **vending machine**:

- A vending machine has several states: `Idle`, `HasMoney`, `Dispensing`, and `OutOfStock`.
- Without the State Pattern, the `VendingMachine` class would contain large `if-else` blocks checking the current state.
- Adding new states would require modifying existing code, violating the **Open/Closed Principle**.

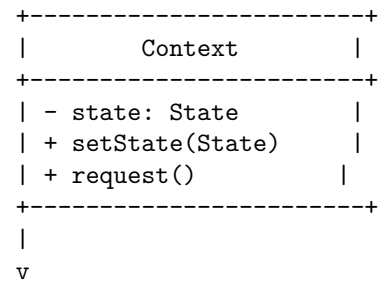
With the State Pattern:

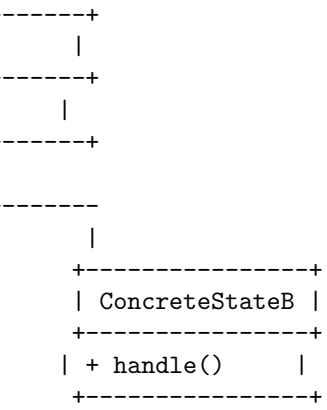
- Each state is represented by a separate class.
- The context (`VendingMachine`) delegates requests to the current state.
- Adding or modifying states no longer affects the core logic.

25.4 Key Concepts

- **Context:** Maintains a reference to the current state and delegates behavior.
- **State Interface:** Defines the common behavior for all states.
- **Concrete States:** Implement state-specific behavior.
- **Client:** Interacts with the context but remains unaware of state transitions.

25.5 UML Structure





25.6 Java Implementation

Step 1: Define the State Interface

```
public interface State {
 void handle();
}
```

Step 2: Implement Concrete States

```
public class IdleState implements State {
 @Override
 public void handle() {
  System.out.println("Machine is idle. Insert money.");
 }
}

public class HasMoneyState implements State {
 @Override
 public void handle() {
  System.out.println("Money inserted. Ready to dispense.");
 }
}

public class DispensingState implements State {
 @Override
 public void handle() {
  System.out.println("Dispensing the product...");
 }
}
```

Step 3: Create the Context

```
public class VendingMachine {
 private State state;

 public VendingMachine() {
  state = new IdleState();  // default state
 }

 public void setState(State state) {
  this.state = state;
 }
```

```
 public void request() {
  state.handle();
 }
}
```

Step 4: Use the State Pattern in the Client

```
public class Main {
 public static void main(String[] args) {
  VendingMachine machine = new VendingMachine();

  machine.request();
  machine.setState(new HasMoneyState());
  machine.request();
  machine.setState(new DispensingState());
  machine.request();
 }
}
```

25.7 Python Implementation

```
class State:
def handle(self):
pass

class IdleState(State):
def handle(self):
print("Machine is idle. Insert money.")

class HasMoneyState(State):
def handle(self):
print("Money inserted. Ready to dispense.")

class DispensingState(State):
def handle(self):
print("Dispensing the product...")

class VendingMachine:
def __init__(self):
self.state = IdleState()

def set_state(self, state):
self.state = state

def request(self):
self.state.handle()

# Client code
machine = VendingMachine()
machine.request()  # Idle
machine.set_state(HasMoneyState())
machine.request()  # Ready to dispense
machine.set_state(DispensingState())
machine.request()  # Dispensing product
```

25.8 Advantages

- Eliminates long `if-else` and `switch` statements.
- Makes code easier to extend by adding new states without modifying existing logic.
- Encapsulates state-specific behavior, improving maintainability.
- Promotes compliance with the Open/Closed and Single Responsibility Principles.

25.9 Drawbacks

- Increases the number of classes (one per state).
- Can introduce overhead when too many states exist.
- State transitions may become complex if poorly managed.

25.10 Real-World Use Cases

- **Media Players:** Different behaviors for playing, paused, or stopped states.
- **Network Protocols:** TCP connections transitioning between states like SYN, ACK, and CLOSE.
- **Game Development:** Player objects switching between idle, attack, jump, and dead states.
- **Workflow Engines:** Document approval systems with multiple review states.

25.11 State vs. Strategy vs. Command

- **State:** Behavior changes dynamically based on internal state.
- **Strategy:** Behavior changes based on client selection; strategies are interchangeable.
- **Command:** Encapsulates an action into an object but does not manage dynamic state transitions.

25.12 Summary

- The **State Pattern** encapsulates behaviors in separate classes and switches them at runtime.
- Reduces conditionals, improves maintainability, and simplifies state transitions.
- Particularly useful when an object's behavior depends heavily on its internal state.
- Common in vending machines, workflow management, game engines, and media players.

State — Interview Blurb

Allows an object to change its behavior when its internal state changes, as if it were changing its class. The pattern encapsulates state-specific logic in separate classes and delegates behavior to the current state object, eliminating large conditional blocks and making transitions clear, testable, and open to extension.

State vs. Strategy

- **Intent: State** models *mode-dependent behavior* that changes as the object's internal state evolves. **Strategy** selects among *interchangeable algorithms* to vary how a task is performed.
- **Who triggers change: State** transitions are typically driven *internally* (object rules, events) and can be automatic. **Strategy** switches are usually chosen *externally* by the client/configuration.
- **Lifecycle: State** emphasizes *transitions over time* (workflow/mode progression). **Strategy** emphasizes *selection at a point in time* (pick the best algorithm).

- **Structure:** Both delegate via a common interface; **State** objects encode transition logic and next-state selection; **Strategy** objects encode algorithm variants and avoid transition concerns.
- **Smell they cure:** **State** replaces sprawling `if/else` or `switch` on mode flags. **Strategy** replaces conditional branching over algorithm choices.

26 Template Method Pattern

26.1 Learning Objectives

- Understand the intent and purpose of the **Template Method** pattern.
- Learn how to define algorithm skeletons while deferring specific steps to subclasses.
- Explore UML structure and analyze Java/Python implementations.
- Compare Template Method with Strategy and State patterns.
- Examine real-world applications, advantages, and trade-offs.

26.2 Introduction

The **Template Method Pattern** is a **behavioral** design pattern that defines the **skeleton of an algorithm** in a base class and lets subclasses override specific steps without modifying the overall structure.

> *Definition (GoF):* "Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing its structure."

This pattern is particularly useful when:

- Multiple classes share a common workflow but differ in certain steps.
- You want to enforce consistency while allowing controlled customization.
- You need to avoid code duplication by reusing the algorithm's skeleton.

26.3 Motivation

Imagine designing a **data processing pipeline**:

- The pipeline includes steps like: `loadData()`, `processData()`, and `saveResults()`.
- Different data formats (CSV, JSON, XML) require different implementations of `loadData()` and `processData()`.
- Without Template Method, you'd duplicate the workflow logic in every subclass.

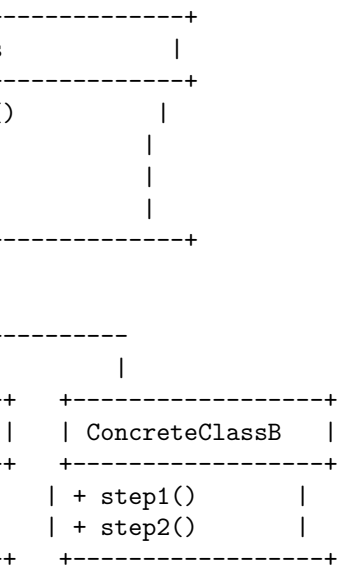
With the Template Pattern:

- The base class defines the workflow's structure.
- Subclasses override only the steps that differ.
- The overall algorithm remains consistent across implementations.

26.4 Key Concepts

- **Abstract Class (Template):** Defines the template method containing the algorithm skeleton.
- **Template Method:** Orchestrates the steps of the algorithm, some of which are implemented by subclasses.
- **Concrete Class:** Implements the steps that vary while inheriting the overall structure.
- **Hook Methods (optional):** Allow subclasses to override optional behaviors without affecting the core template.

26.5 UML Structure



26.6 Java Implementation

Step 1: Define the Abstract Class (Template)

```
public abstract class DataProcessor {
 // Template Method - final to prevent overriding
 public final void process() {
  loadData();
  processData();
  saveData();
 }

 // Steps to be implemented by subclasses
 protected abstract void loadData();
 protected abstract void processData();

 // Default implementation
 protected void saveData() {
  System.out.println("Saving data to storage.");
 }
}
```

Step 2: Implement Concrete Classes

```
public class CSVDataProcessor extends DataProcessor {
 protected void loadData() {
  System.out.println("Loading CSV data...");
 }

 protected void processData() {
  System.out.println("Processing CSV data...");
 }
}

public class JSONDataProcessor extends DataProcessor {
 protected void loadData() {
```

```
  System.out.println("Loading JSON data...");
 }

 protected void processData() {
  System.out.println("Processing JSON data...");
 }
}
```

Step 3: Use the Template Method in the Client

```
public class Main {
 public static void main(String[] args) {
  DataProcessor csv = new CSVDataProcessor();
  csv.process();

  DataProcessor json = new JSONDataProcessor();
  json.process();
 }
}
```

26.7 Python Implementation

```
from abc import ABC, abstractmethod

class DataProcessor(ABC):
def process(self):
self.load_data()
self.process_data()
self.save_data()

@abstractmethod
def load_data(self):
pass

@abstractmethod
def process_data(self):
pass

def save_data(self):
print("Saving data to storage.")

class CSVDataProcessor(DataProcessor):
def load_data(self):
print("Loading CSV data...")

def process_data(self):
print("Processing CSV data...")

class JSONDataProcessor(DataProcessor):
def load_data(self):
print("Loading JSON data...")

def process_data(self):
print("Processing JSON data...")

# Client code
```

```
csv = CSVDataProcessor()
csv.process()

json = JSONDataProcessor()
json.process()
```

26.8 Advantages

- Promotes code reuse by encapsulating shared workflows in the template.
- Ensures consistency across different implementations.
- Simplifies maintenance by centralizing the algorithm's skeleton.
- Makes adding new variations straightforward by extending the abstract class.

26.9 Drawbacks

- Increases the number of classes when there are many variations.
- Subclassing introduces inheritance coupling, reducing flexibility compared to composition.
- Overriding template steps incorrectly can break the algorithm's intended flow.

26.10 Real-World Use Cases

- **Frameworks:** Web frameworks like Spring, Django, and Rails define request/response lifecycles using templates.
- **Testing Frameworks:** JUnit, PyTest, and TestNG structure test execution via templates.
- **Data Pipelines:** ETL (Extract, Transform, Load) workflows use predefined skeletons with interchangeable steps.
- **Game Development:** Game loops often follow a template where subclasses define rendering or update logic.

26.11 Template Method vs. Strategy vs. State

- **Template Method:** The algorithm skeleton is fixed, but subclasses customize certain steps.
- **Strategy:** The entire algorithm is interchangeable; selected dynamically at runtime.
- **State:** Behavior changes automatically based on internal object state rather than subclass overrides.

26.12 Summary

- The **Template Method Pattern** defines the structure of an algorithm while allowing subclasses to customize specific steps.
- Reduces code duplication and promotes reuse across related classes.
- Widely used in frameworks, pipelines, and testing systems.
- Often combined with Strategy and State patterns for greater flexibility.

Template Method — Interview Blurb

Defines the skeleton of an algorithm in a superclass and defers selected steps to subclasses. This preserves the overall sequence (invariants) while enabling controlled customization through primitive operations and optional hooks—embodying the "don't call us, we'll call you" principle.

27 Iterator Pattern

27.1 Learning Objectives

- Understand the intent and purpose of the **Iterator** pattern.
- Learn how to traverse collections without exposing their internal structure.
- Explore the UML structure and study Java/Python implementations.
- Compare Iterator with Composite and Visitor patterns.
- Examine real-world applications, advantages, and trade-offs.

27.2 Introduction

The **Iterator Pattern** is a **behavioral** design pattern that provides a way to **sequentially access elements** of a collection without exposing its underlying representation.

> *Definition (GoF):* "Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation."

This pattern abstracts the traversal logic from the collection, allowing the client to iterate consistently across various data structures.

27.3 Motivation

Imagine designing a **library management system**:

- Books are stored in various formats: arrays, lists, trees, and hash maps.
- Without Iterator, clients must know each collection's internal structure to traverse it.
- This leads to tightly coupled, error-prone code and violates the **Open/Closed Principle**.

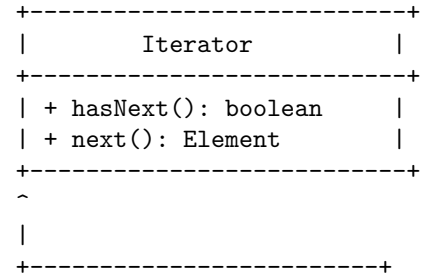
With the Iterator Pattern:

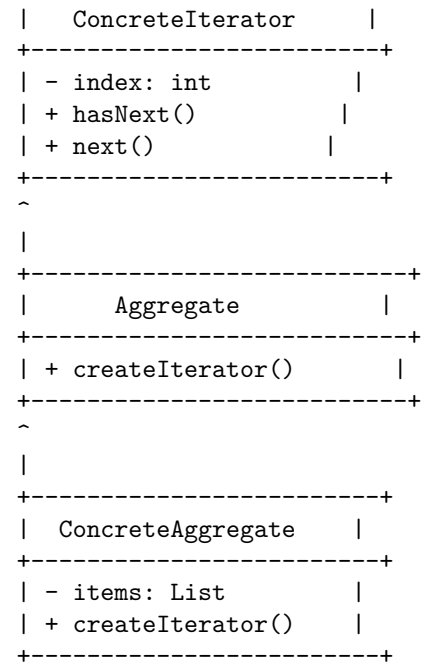
- Each collection provides its own iterator implementation.
- Clients interact with a uniform interface for traversal.
- Underlying data structures remain hidden and easily interchangeable.

27.4 Key Concepts

- **Iterator:** Defines methods to traverse elements sequentially.
- **Concrete Iterator:** Implements the traversal logic for a specific collection.
- **Aggregate (Collection):** Defines an interface to create iterators.
- **Concrete Aggregate:** Implements the collection interface and returns an iterator.
- **Client:** Uses the iterator to traverse elements without knowing the underlying representation.

27.5 UML Structure





27.6 Java Implementation

Step 1: Define the Iterator Interface

```
public interface Iterator<T> {
 boolean hasNext();
 T next();
}
```

Step 2: Implement the Concrete Iterator

```
import java.util.List;

public class BookIterator implements Iterator<String> {
 private List<String> books;
 private int index = 0;

 public BookIterator(List<String> books) {
  this.books = books;
 }

 @Override
 public boolean hasNext() {
  return index < books.size();
 }

 @Override
 public String next() {
  return books.get(index++);
 }
}
```

Step 3: Define the Aggregate Interface

```
public interface BookCollection {
```

```
 Iterator<String> createIterator();
}
```

Step 4: Implement the Concrete Aggregate

```
import java.util.ArrayList;
import java.util.List;

public class Library implements BookCollection {
 private List<String> books = new ArrayList<>();

 public void addBook(String book) {
  books.add(book);
 }

 @Override
 public Iterator<String> createIterator() {
  return new BookIterator(books);
 }
}
```

Step 5: Use the Iterator in the Client

```
public class Main {
 public static void main(String[] args) {
  Library library = new Library();
  library.addBook("Design Patterns");
  library.addBook("Clean Code");
  library.addBook("Effective Java");

  Iterator<String> iterator = library.createIterator();
  while (iterator.hasNext()) {
   System.out.println(iterator.next());
  }
 }
}
```

27.7 Python Implementation

```
class BookIterator:
def __init__(self, books):
self._books = books
self._index = 0

def __iter__(self):
return self

def __next__(self):
if self._index < len(self._books):
book = self._books[self._index]
self._index += 1
return book
raise StopIteration

class Library:
def __init__(self):
self._books = []
```

```
def add_book(self, book):
self._books.append(book)

def __iter__(self):
return BookIterator(self._books)

# Client code
library = Library()
library.add_book("Design Patterns")
library.add_book("Clean Code")
library.add_book("Effective Java")

for book in library:
print(book)
```

27.8 Advantages

- Provides a consistent traversal interface across different data structures.
- Hides internal collection details from clients.
- Simplifies switching between different collection implementations.
- Supports multiple, independent iterators on the same collection.

27.9 Drawbacks

- Introduces additional classes, increasing code size.
- May reduce performance compared to direct access in some cases.
- Simple collections may not require custom iterators.

27.10 Real-World Use Cases

- **Java Collections Framework:** Built-in iterators for Lists, Sets, and Maps.
- **Python Iterators and Generators:** The `__iter__` and `__next__` protocol.
- **Database Result Sets:** Iterating through query results.
- **Tree Traversals:** Navigating hierarchical structures like XML, JSON, or DOM.

27.11 Iterator vs. Visitor vs. Composite

- **Iterator:** Sequentially accesses elements in a collection.
- **Visitor:** Applies operations to each element without exposing structure.
- **Composite:** Represents hierarchical structures; often paired with Iterator for traversal.

27.12 Summary

- The **Iterator Pattern** standardizes how collections are traversed without exposing internal representations.
- Improves flexibility, encapsulation, and usability across varying data structures.
- Widely used in programming languages, frameworks, and enterprise systems.
- Often combined with Composite, Visitor, and Observer for working with hierarchical data.

Iterator — Interview Blurb

Provides a standardized interface for sequentially accessing elements of a collection without exposing its underlying representation. The pattern separates traversal logic from the aggregate itself, allowing different iteration strategies (forward, reverse, filtered, concurrent) while preserving encapsulation and flexibility.

28 Mediator Pattern

28.1 Learning Objectives

- Understand the intent and purpose of the **Mediator** pattern.
- Learn how to simplify communication between objects by centralizing control.
- Explore the UML structure and analyze Java/Python implementations.
- Compare Mediator with Observer, Facade, and Command patterns.
- Examine real-world applications, advantages, and trade-offs.

28.2 Introduction

The **Mediator Pattern** is a **behavioral** design pattern that defines an object (the mediator) to **centralize communication** between multiple interacting components.

> *Definition (GoF):* "Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by preventing objects from referring to each other explicitly and lets you vary their interaction independently."

This pattern is especially useful when:

- Multiple components communicate frequently.
- Adding or modifying components causes ripple effects in other classes.
- You want to decouple components and make them reusable.

28.3 Motivation

Imagine designing a **chat room system**:

- Without Mediator, every user would need references to every other user.
- Sending a message would require looping through all connected users directly.
- This leads to **spaghetti dependencies** where changing one class affects many others.

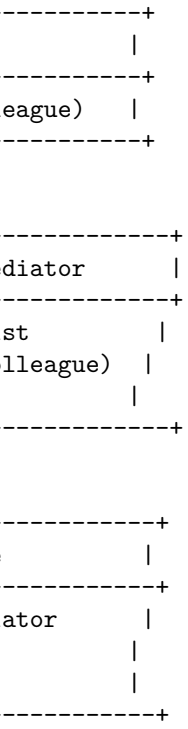
With the Mediator Pattern:

- A central `ChatRoom` acts as the mediator.
- Users send messages to the `ChatRoom`, which forwards them to other participants.
- Users remain decoupled, interacting only through the mediator.

28.4 Key Concepts

- **Mediator:** Defines the interface for communication between components.
- **Concrete Mediator:** Implements the coordination logic.
- **Colleague:** A participant in the system that communicates via the mediator.
- **Client:** Creates colleagues and registers them with the mediator.

28.5 UML Structure



28.6 Java Implementation

Step 1: Define the Mediator Interface

```
public interface ChatMediator {
 void sendMessage(String message, User user);
 void addUser(User user);
}
```

Step 2: Implement the Concrete Mediator

```
import java.util.ArrayList;
import java.util.List;

public class ChatRoom implements ChatMediator {
 private List<User> users = new ArrayList<>();

 @Override
 public void addUser(User user) {
  users.add(user);
 }

 @Override
 public void sendMessage(String message, User sender) {
  for (User user : users) {
   if (user != sender) {
    user.receive(message);
   }
  }
 }
}
```

Step 3: Define the Colleague Class

```
public abstract class User {
 protected ChatMediator mediator;
 protected String name;

 public User(ChatMediator mediator, String name) {
  this.mediator = mediator;
  this.name = name;
 }

 public abstract void send(String message);
 public abstract void receive(String message);
}
```

Step 4: Implement Concrete Colleagues

```
public class ChatUser extends User {
 public ChatUser(ChatMediator mediator, String name) {
  super(mediator, name);
 }

 @Override
 public void send(String message) {
  System.out.println(this.name + " sends: " + message);
  mediator.sendMessage(message, this);
 }

 @Override
 public void receive(String message) {
  System.out.println(this.name + " receives: " + message);
 }
}
```

Step 5: Use the Mediator in the Client

```
public class Main {
 public static void main(String[] args) {
  ChatMediator chatRoom = new ChatRoom();

  User alice = new ChatUser(chatRoom, "Alice");
  User bob = new ChatUser(chatRoom, "Bob");
  User charlie = new ChatUser(chatRoom, "Charlie");

  chatRoom.addUser(alice);
  chatRoom.addUser(bob);
  chatRoom.addUser(charlie);

  alice.send("Hello everyone!");
  bob.send("Hi Alice!");
 }
}
```

28.7 Python Implementation

```
class ChatMediator:
def send_message(self, message, sender):
```

```
pass

def add_user(self, user):
pass

class ChatRoom(ChatMediator):
def __init__(self):
self.users = []

def add_user(self, user):
self.users.append(user)

def send_message(self, message, sender):
for user in self.users:
if user != sender:
user.receive(message)

class User:
def __init__(self, mediator, name):
self.mediator = mediator
self.name = name

def send(self, message):
print(f"{self.name} sends: {message}")
self.mediator.send_message(message, self)

def receive(self, message):
print(f"{self.name} receives: {message}")

# Client code
chatroom = ChatRoom()
alice = User(chatroom, "Alice")
bob = User(chatroom, "Bob")
charlie = User(chatroom, "Charlie")

chatroom.add_user(alice)
chatroom.add_user(bob)
chatroom.add_user(charlie)

alice.send("Hi everyone!")
bob.send("Hey Alice!")
```

28.8 Advantages

- Reduces tight coupling between interacting components.
- Centralizes complex communication logic.
- Makes components more reusable and maintainable.
- Simplifies adding or removing participants without changing others.

28.9 Drawbacks

- The mediator can become a **god object** if it grows too large.
- Increases complexity when many types of messages and participants exist.
- Overuse may reduce flexibility in systems with simple communication.

28.10 Real-World Use Cases

- **Chat Systems:** Group messaging platforms like Slack and WhatsApp.
- **GUI Frameworks:** Mediating events between buttons, windows, and controllers.
- **Air Traffic Control Systems:** Control towers coordinate planes instead of planes contacting each other directly.
- **Microservices Communication:** Message brokers like Kafka and RabbitMQ.

28.11 Mediator vs. Observer vs. Facade

- **Mediator:** Coordinates communication between peers, centralizing logic.
- **Observer:** Notifies dependent objects of changes but doesn't centralize communication.
- **Facade:** Simplifies a subsystem but doesn't mediate peer-to-peer communication.

28.12 Summary

- The **Mediator Pattern** centralizes communication, reducing coupling and dependencies.
- Useful when many components need to interact dynamically.
- Commonly used in chat rooms, GUIs, message brokers, and distributed systems.
- Often paired with Observer and Command patterns in event-driven architectures.

Mediator — Interview Blurb

Encapsulates how a set of objects interact by introducing a mediator that coordinates their communication. Objects no longer refer to each other directly, reducing tight coupling and complex dependency webs. This promotes cleaner collaboration logic, easier maintenance, and centralized control over interactions.

Mediator vs. Observer

- **Intent:** Both decouple communicating objects, but **Mediator** centralizes control, while **Observer** distributes it.
- **Communication flow:** In **Mediator**, objects communicate *through* a central coordinator. In **Observer**, objects broadcast events *to many dependents*.
- **Coupling:** **Mediator** reduces many-to-many coupling to one-to-many via a hub. **Observer** reduces one-to-many coupling by standardizing notifications.
- **Control direction:** **Mediator** enforces coordination logic and often controls workflow. **Observer** merely reacts; control remains with the subjects and observers.
- **Use when:** Use **Mediator** to manage complex inter-object dependencies or GUI dialogs; use **Observer** for event propagation and reactive updates.

29 Memento Pattern

29.1 Learning Objectives

- Understand the intent and purpose of the **Memento** pattern.
- Learn how to capture and restore object state without breaking encapsulation.
- Explore UML structure and study Java/Python implementations.
- Compare Memento with Command, Prototype, and State patterns.
- Examine real-world applications, advantages, and trade-offs.

29.2 Introduction

The **Memento Pattern** is a **behavioral** design pattern that captures an object's internal state and allows restoring it later, **without exposing its implementation details**.

> *Definition (GoF):* "Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later."

This pattern is ideal for scenarios where:

- You need to implement **undo/redo** functionality.
- You want to **snapshot** the state of objects for persistence.
- You need to **roll back** to a previous configuration.

29.3 Motivation

Imagine designing a **text editor**:

- Users can type, delete, and modify text.
- If they make a mistake, they want to revert to a previous version.
- Without Memento, the editor would need to expose all internal fields to save and restore state — breaking encapsulation.

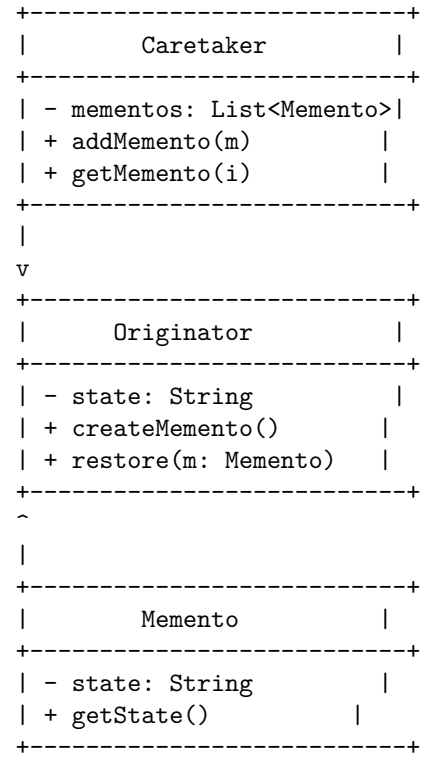
With the Memento Pattern:

- The editor (**Originator**) creates a `Memento` snapshot of its current state.
- A `Caretaker` manages these mementos and can restore one later.
- Internal details remain hidden from the caretaker.

29.4 Key Concepts

- **Originator:** The object whose state we want to save and restore.
- **Memento:** Stores the internal state of the originator in an immutable snapshot.
- **Caretaker:** Manages saved mementos but never accesses their internal structure.

29.5 UML Structure



29.6 Java Implementation

Step 1: Define the Memento Class

```
public class Memento {
 private final String state;

 public Memento(String state) {
  this.state = state;
 }

 public String getState() {
  return state;
 }
}
```

Step 2: Create the Originator

```
public class TextEditor {
 private String text;

 public void setText(String text) {
  this.text = text;
 }

 public String getText() {
  return text;
 }

 public Memento save() {
```

```
  return new Memento(text);
 }

 public void restore(Memento memento) {
  text = memento.getState();
 }
}
```

Step 3: Implement the Caretaker

```
import java.util.Stack;

public class History {
 private Stack<Memento> states = new Stack<>();

 public void save(Memento memento) {
  states.push(memento);
 }

 public Memento undo() {
  if (!states.isEmpty()) {
   return states.pop();
  }
  return null;
 }
}
```

Step 4: Use the Memento in the Client

```
public class Main {
 public static void main(String[] args) {
  TextEditor editor = new TextEditor();
  History history = new History();

  editor.setText("Version 1");
  history.save(editor.save());

  editor.setText("Version 2");
  history.save(editor.save());

  editor.setText("Version 3");
  System.out.println("Current: " + editor.getText());

  editor.restore(history.undo());
  System.out.println("After undo: " + editor.getText());

  editor.restore(history.undo());
  System.out.println("After second undo: " + editor.getText());
 }
}
```

29.7 Python Implementation

```
class Memento:
def __init__(self, state):
self._state = state
```

```
def get_state(self):
return self._state

class TextEditor:
def __init__(self):
self._text = ""

def set_text(self, text):
self._text = text

def get_text(self):
return self._text

def save(self):
return Memento(self._text)

def restore(self, memento):
self._text = memento.get_state()

class History:
def __init__(self):
self._states = []

def save(self, memento):
self._states.append(memento)

def undo(self):
return self._states.pop() if self._states else None

# Client code
editor = TextEditor()
history = History()

editor.set_text("Version 1")
history.save(editor.save())

editor.set_text("Version 2")
history.save(editor.save())

editor.set_text("Version 3")
print("Current:", editor.get_text())

editor.restore(history.undo())
print("After undo:", editor.get_text())

editor.restore(history.undo())
print("After second undo:", editor.get_text())
```

29.8 Advantages

- Preserves encapsulation by keeping state details hidden from the caretaker.
- Enables easy implementation of **undo/redo** features.
- Supports multiple state snapshots for versioning or rollback.
- Works well with command-based systems for state management.

29.9 Drawbacks

- Memory usage can grow quickly when storing many snapshots.
- Capturing large, deep object states may affect performance.
- Complexity increases when managing interdependent states.

29.10 Real-World Use Cases

- **Text Editors:** Undo/redo functionality (e.g., MS Word, VS Code).
- **Database Transactions:** Rollback mechanisms in relational databases.
- **Game Development:** Saving and restoring player progress.
- **Workflow Engines:** Version control of process states.

29.11 Memento vs. Command vs. State

- **Memento:** Captures and restores object state without exposing implementation.
- **Command:** Encapsulates an operation, often combined with Memento for undo functionality.
- **State:** Defines behaviors for state transitions rather than restoring snapshots.

29.12 Summary

- The **Memento Pattern** captures an object's internal state for future restoration without exposing implementation details.
- Ideal for undo/redo, rollback, snapshotting, and versioning features.
- Commonly used in editors, transaction systems, games, and workflow management.
- Often paired with the **Command Pattern** to implement undoable actions.

Memento — Interview Blurb

Captures and externalizes an object's internal state without violating encapsulation, so the object can be restored to that exact state later. Commonly used to implement undo/redo, checkpoints, or version history while keeping the originator's internals hidden from other objects.

Command vs. Memento

- **Intent: Command** encapsulates an action so it can be executed, queued, or undone. **Memento** captures an object's state so it can be restored later.
- **Focus: Command** represents *behavior* (what to do). **Memento** represents *state* (what data to restore).
- **Undo mechanism:** A **Command** can use a **Memento** internally to roll back changes. Memento alone cannot perform actions—it only stores snapshots.
- **Coupling: Command** couples an invoker and a receiver. **Memento** decouples an originator from external state management.
- **Analogy:** Command = "record and replay an operation." Memento = "save and restore a snapshot."

30 Interpreter Pattern

30.1 Learning Objectives

- Understand the intent and purpose of the **Interpreter** pattern.
- Learn how to define and evaluate expressions using a formal grammar.
- Explore the UML structure and study Java/Python implementations.
- Compare Interpreter with Visitor, Strategy, and Composite patterns.
- Examine real-world applications, advantages, and trade-offs.

30.2 Introduction

The **Interpreter Pattern** is a **behavioral** design pattern that defines a representation of a grammar and provides an interpreter to evaluate sentences in the language.

> *Definition (GoF):* "Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language."

This pattern is particularly useful when:

- You have a well-defined grammar and need to evaluate structured expressions.
- You need to repeatedly parse and execute instructions at runtime.
- You want to implement simple scripting, querying, or rule-based systems.

30.3 Motivation

Imagine designing a **rule-based calculator**:

- Expressions like `"5 + 10 - 3"` need to be parsed and evaluated.
- Without the Interpreter Pattern, you'd hardcode parsing logic and evaluation in one place.
- This leads to poor maintainability and difficulty adding new operations.

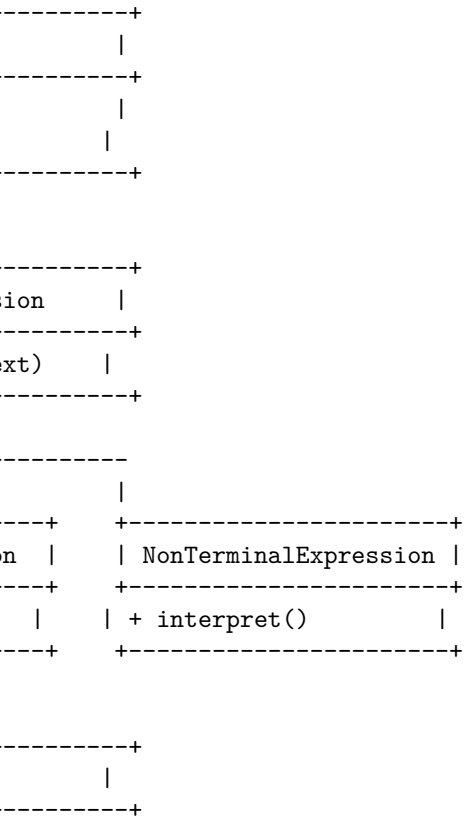
With the Interpreter Pattern:

- The grammar is defined using classes like `Expression`, `AddExpression`, and `SubtractExpression`.
- Each expression implements an `interpret()` method.
- Complex expressions are composed recursively, making evaluation extensible.

30.4 Key Concepts

- **Abstract Expression:** Defines an interface for interpreting expressions.
- **Terminal Expression:** Implements interpretation for atomic elements in the grammar.
- **Non-Terminal Expression:** Implements rules that combine terminal expressions into larger structures.
- **Context:** Contains global information used during interpretation.
- **Client:** Builds and interprets the expression tree.

30.5 UML Structure



30.6 Java Implementation

Step 1: Define the Expression Interface

```
public interface Expression {
 int interpret();
}
```

Step 2: Implement Terminal Expressions

```
public class NumberExpression implements Expression {
 private int number;

 public NumberExpression(int number) {
  this.number = number;
 }

 @Override
 public int interpret() {
  return number;
 }
}
```

Step 3: Implement Non-Terminal Expressions

```
public class AddExpression implements Expression {
 private Expression left;
 private Expression right;
```

```
 public AddExpression(Expression left, Expression right) {
  this.left = left;
  this.right = right;
 }

 @Override
 public int interpret() {
  return left.interpret() + right.interpret();
 }
}

public class SubtractExpression implements Expression {
 private Expression left;
 private Expression right;

 public SubtractExpression(Expression left, Expression right) {
  this.left = left;
  this.right = right;
 }

 @Override
 public int interpret() {
  return left.interpret() - right.interpret();
 }
}
```

Step 4: Use the Interpreter in the Client

```
public class Main {
 public static void main(String[] args) {
  Expression expr = new SubtractExpression(
  new AddExpression(new NumberExpression(5), new NumberExpression(10)),
  new NumberExpression(3)
  );

  System.out.println("Result: " + expr.interpret()); // Output: 12
 }
}
```

30.7 Python Implementation

```
class Expression:
def interpret(self):
pass

class NumberExpression(Expression):
def __init__(self, number):
self.number = number

def interpret(self):
return self.number

class AddExpression(Expression):
def __init__(self, left, right):
self.left = left
self.right = right
```

```
def interpret(self):
return self.left.interpret() + self.right.interpret()

class SubtractExpression(Expression):
def __init__(self, left, right):
self.left = left
self.right = right

def interpret(self):
return self.left.interpret() - self.right.interpret()

# Client code
expr = SubtractExpression(
AddExpression(NumberExpression(5), NumberExpression(10)),
NumberExpression(3)
)

print("Result:", expr.interpret())  # Output: 12
```

30.8 Advantages

- Simplifies the evaluation of complex expressions.
- Makes adding new grammar rules easy via subclassing.
- Promotes code reuse through composable expression objects.
- Works well with recursive grammars and tree-based evaluations.

30.9 Drawbacks

- Can lead to a **large number of classes** for complex grammars.
- Performance can degrade when interpreting very large or deeply nested expressions.
- Not suitable for general-purpose language parsing — better for simple, domain-specific grammars.

30.10 Real-World Use Cases

- **Math Expression Evaluators:** Parsing arithmetic formulas dynamically.
- **Query Languages:** SQL interpreters, rule engines, and filtering DSLs.
- **Configuration Parsing:** Processing domain-specific configuration files.
- **Regex Engines:** Regular expression parsing and evaluation.

30.11 Interpreter vs. Visitor vs. Strategy

- **Interpreter:** Defines a grammar and evaluates structured expressions.
- **Visitor:** Adds new operations to existing grammars without modifying classes.
- **Strategy:** Encapsulates algorithms but does not define grammars.

30.12 Summary

- The **Interpreter Pattern** defines a representation of grammar and an interpreter for evaluating expressions.
- Useful for simple languages, DSLs, configuration parsers, and math evaluation engines.
- Avoids hardcoding parsing and evaluation logic, improving extensibility.
- Often combined with the **Visitor Pattern** for adding new operations cleanly.

Interpreter — Interview Blurb

Defines a grammar for a simple language and represents each grammar rule as a class, allowing sentences to be interpreted by recursively evaluating the object structure. Useful for parsing expressions, commands, or configuration rules where the grammar is relatively small, stable, and frequently extended.

Interpreter vs. Visitor

- **Intent:** **Interpreter** defines and evaluates a language's grammar. **Visitor** separates algorithms from object structures to apply operations without modifying them.
- **Structure:** **Interpreter** represents grammar rules as a class hierarchy (Expression trees). **Visitor** traverses an existing hierarchy to perform multiple independent operations.
- **Variability:** **Interpreter** is easy to extend with new grammar rules but harder to add new interpretations. **Visitor** is easy to add new operations but harder to extend the element hierarchy.
- **Use when:** Use **Interpreter** for domain-specific languages or rule engines. Use **Visitor** to apply multiple operations (e.g., printing, evaluating, optimizing) on complex object trees.
- **Analogy:** Interpreter = "build a language." Visitor = "walk a structure and perform actions."

31 Chain of Responsibility Pattern

31.1 Learning Objectives

- Understand the intent and purpose of the **Chain of Responsibility** (CoR) pattern.
- Learn how to pass requests along a chain of potential handlers.
- Explore the UML structure and analyze Java/Python implementations.
- Compare CoR with Decorator, Command, and Observer patterns.
- Examine real-world applications, advantages, and trade-offs.

31.2 Introduction

The **Chain of Responsibility Pattern** is a **behavioral** design pattern that lets you pass a request along a chain of handlers until one of them processes it.

> *Definition (GoF):* "Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it."

This pattern is ideal when:

- Multiple objects could potentially handle a request.
- You want to decouple request senders from receivers.
- You need to support dynamic request handling without hardcoding dependencies.

31.3 Motivation

Imagine designing a **technical support system**:

- Customer support requests can be handled by Level 1, Level 2, or Level 3 support teams.
- Without CoR, the client would need to explicitly know which handler is responsible.
- If responsibilities change, client code breaks and needs frequent updates.

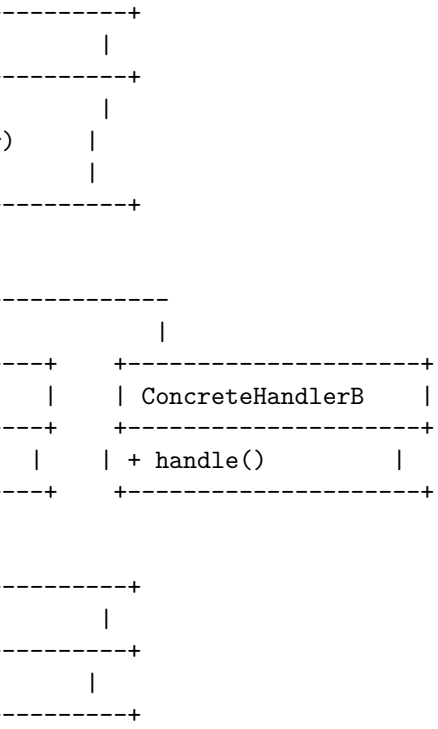
With the Chain of Responsibility Pattern:

- Each support level decides whether to handle the request or forward it.
- The client just sends the request to the first handler.
- The chain is easily configurable at runtime.

31.4 Key Concepts

- **Handler:** Defines the interface for handling requests and maintaining the reference to the next handler.
- **Concrete Handler:** Implements request handling; forwards requests if unable to handle.
- **Client:** Initiates requests without knowing which handler will process them.
- **Chain:** A linked list of handlers that sequentially process or pass along requests.

31.5 UML Structure



31.6 Java Implementation

Step 1: Define the Handler Interface

```
public abstract class Handler {
 protected Handler next;

 public Handler setNext(Handler next) {
  this.next = next;
  return next;
 }

 public abstract void handleRequest(String request);
}
```

Step 2: Implement Concrete Handlers

```
public class LevelOneSupport extends Handler {
 @Override
 public void handleRequest(String request) {
  if (request.equals("basic")) {
   System.out.println("Level 1 handled request: " + request);
  } else if (next != null) {
   next.handleRequest(request);
  }
 }
}

public class LevelTwoSupport extends Handler {
 @Override
 public void handleRequest(String request) {
```

```
  if (request.equals("intermediate")) {
   System.out.println("Level 2 handled request: " + request);
  } else if (next != null) {
   next.handleRequest(request);
  }
 }
}

public class LevelThreeSupport extends Handler {
 @Override
 public void handleRequest(String request) {
  if (request.equals("advanced")) {
   System.out.println("Level 3 handled request: " + request);
  } else if (next != null) {
   next.handleRequest(request);
  } else {
   System.out.println("Request could not be handled: " + request);
  }
 }
}
```

Step 3: Use the CoR in the Client

```
public class Main {
 public static void main(String[] args) {
  Handler level1 = new LevelOneSupport();
  Handler level2 = new LevelTwoSupport();
  Handler level3 = new LevelThreeSupport();

  // Build the chain
  level1.setNext(level2).setNext(level3);

  // Send requests
  level1.handleRequest("basic");
  level1.handleRequest("intermediate");
  level1.handleRequest("advanced");
  level1.handleRequest("unknown");
 }
}
```

31.7 Python Implementation

```
class Handler:
def __init__(self):
self.next = None

def set_next(self, handler):
self.next = handler
return handler

def handle(self, request):
raise NotImplementedError

class LevelOneSupport(Handler):
def handle(self, request):
if request == "basic":
```

```
print(f"Level 1 handled request: {request}")
elif self.next:
self.next.handle(request)

class LevelTwoSupport(Handler):
def handle(self, request):
if request == "intermediate":
print(f"Level 2 handled request: {request}")
elif self.next:
self.next.handle(request)

class LevelThreeSupport(Handler):
def handle(self, request):
if request == "advanced":
print(f"Level 3 handled request: {request}")
elif self.next:
self.next.handle(request)
else:
print(f"Request could not be handled: {request}")

# Client code
level1 = LevelOneSupport()
level2 = LevelTwoSupport()
level3 = LevelThreeSupport()

level1.set_next(level2).set_next(level3)

level1.handle("basic")
level1.handle("intermediate")
level1.handle("advanced")
level1.handle("unknown")
```

31.8 Advantages

- Decouples senders from receivers, improving modularity.
- Makes it easy to add or remove handlers without modifying client code.
- Supports dynamic request routing and flexible chain configurations.
- Promotes **Open/Closed Principle** by avoiding hardcoded logic.

31.9 Drawbacks

- Requests may go unhandled if no handler in the chain processes them.
- Debugging can be harder because control flow is distributed across multiple objects.
- Long chains can lead to performance overhead.

31.10 Real-World Use Cases

- **GUI Frameworks:** Event handling in toolkits like Swing, JavaFX, and Android.
- **Logging Systems:** Chains of loggers with different severity levels.
- **Web Frameworks:** Middleware pipelines in Flask, Spring Boot, Express, etc.
- **Authentication Flows:** Multi-step verification where requests pass through security layers.

31.11 Chain of Responsibility vs. Decorator vs. Command

- **CoR:** Passes a request along a chain until a handler processes it.
- **Decorator:** Wraps objects to add responsibilities dynamically but doesn't forward requests conditionally.
- **Command:** Encapsulates an action but doesn't chain multiple handlers automatically.

31.12 Summary

- The **Chain of Responsibility Pattern** decouples senders and receivers by passing requests along a chain.
- Flexible, dynamic, and extensible for handling complex workflows.
- Widely used in GUIs, logging, middleware, and security pipelines.
- Often combined with **Command** and **Observer** in event-driven architectures.

Chain of Responsibility — Interview Blurb

Decouples senders from receivers by passing a request along a chain of potential handlers. Each handler decides whether to process the request or forward it to the next in line. This promotes loose coupling, flexible routing, and dynamic composition of processing steps without requiring the sender to know who handles what.

32 Visitor Pattern

32.1 Learning Objectives

- Understand the intent and purpose of the **Visitor** pattern.
- Learn how to add new operations to existing object structures without modifying them.
- Explore the UML structure and study Java/Python implementations.
- Compare Visitor with Strategy, Interpreter, and Composite patterns.
- Examine real-world applications, advantages, and trade-offs.

32.2 Introduction

The **Visitor Pattern** is a **behavioral** design pattern that lets you define new operations on an object structure **without changing its classes**. Instead of embedding operations within the objects, you create a separate `Visitor` class that performs the operations.

> *Definition (GoF):* "Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates."

This pattern is useful when:

- You have a complex, hierarchical object structure.
- You need to perform many unrelated operations on the same object graph.
- You want to comply with the **Open/Closed Principle**: extend functionality without modifying existing code.

32.3 Motivation

Imagine designing a **document renderer**:

- A document has various elements: `Paragraph`, `Image`, and `Table`.
- Without Visitor, every element class would need methods for every possible operation — rendering, exporting, printing, etc.
- Adding a new operation (e.g., exporting to PDF) would require modifying all element classes.

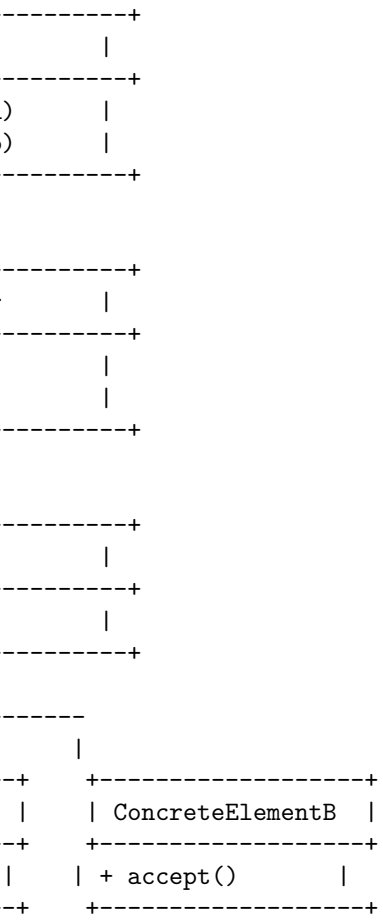
With the Visitor Pattern:

- The element hierarchy remains unchanged.
- New operations are added by introducing new `Visitor` implementations.
- The object structure stays clean and focused on data, while visitors handle operations.

32.4 Key Concepts

- **Visitor:** Declares operations for each element type.
- **Concrete Visitor:** Implements the operations defined in the visitor interface.
- **Element:** Declares the `accept()` method, allowing visitors to process it.
- **Concrete Element:** Implements the `accept()` method and calls back the appropriate visitor method.
- **Object Structure:** A collection of elements over which visitors operate.

32.5 UML Structure



32.6 Java Implementation

Step 1: Define the Visitor Interface

```
public interface Visitor {
 void visit(Paragraph paragraph);
 void visit(Image image);
}
```

Step 2: Define the Element Interface

```
public interface DocumentElement {
 void accept(Visitor visitor);
}
```

Step 3: Implement Concrete Elements

```
public class Paragraph implements DocumentElement {
 private String text;

 public Paragraph(String text) {
  this.text = text;
 }

 public String getText() {
  return text;
```

```
 }

 @Override
 public void accept(Visitor visitor) {
  visitor.visit(this);
 }
}

public class Image implements DocumentElement {
 private String fileName;

 public Image(String fileName) {
  this.fileName = fileName;
 }

 public String getFileName() {
  return fileName;
 }

 @Override
 public void accept(Visitor visitor) {
  visitor.visit(this);
 }
}
```

Step 4: Implement Concrete Visitors

```
public class HTMLRenderer implements Visitor {
 @Override
 public void visit(Paragraph paragraph) {
  System.out.println("<p>" + paragraph.getText() + "</p>");
 }

 @Override
 public void visit(Image image) {
  System.out.println("<img src='" + image.getFileName() + "' />");
 }
}

public class TextExporter implements Visitor {
 @Override
 public void visit(Paragraph paragraph) {
  System.out.println(paragraph.getText());
 }

 @Override
 public void visit(Image image) {
  System.out.println("[Image: " + image.getFileName() + "]");
 }
}
```

Step 5: Use the Visitor in the Client

```
import java.util.ArrayList;
import java.util.List;
```

```
public class Main {
 public static void main(String[] args) {
  List<DocumentElement> elements = new ArrayList<>();
  elements.add(new Paragraph("Hello World"));
  elements.add(new Image("design.png"));

  Visitor htmlVisitor = new HTMLRenderer();
  Visitor textVisitor = new TextExporter();

  for (DocumentElement e : elements) {
   e.accept(htmlVisitor);
  }

  System.out.println("---- Export as Text ----");
  for (DocumentElement e : elements) {
   e.accept(textVisitor);
  }
 }
}
```

32.7 Python Implementation

```
class Visitor:
def visit_paragraph(self, paragraph):
pass

def visit_image(self, image):
pass

class DocumentElement:
def accept(self, visitor):
pass

class Paragraph(DocumentElement):
def __init__(self, text):
self.text = text

def accept(self, visitor):
visitor.visit_paragraph(self)

class Image(DocumentElement):
def __init__(self, filename):
self.filename = filename

def accept(self, visitor):
visitor.visit_image(self)

class HTMLRenderer(Visitor):
def visit_paragraph(self, paragraph):
print(f"<p>{paragraph.text}</p>")

def visit_image(self, image):
print(f"<img src='{image.filename}' />")

class TextExporter(Visitor):
```

```
def visit_paragraph(self, paragraph):
print(paragraph.text)

def visit_image(self, image):
print(f"[Image: {image.filename}]")

# Client code
elements = [Paragraph("Hello World"), Image("design.png")]

html_renderer = HTMLRenderer()
text_exporter = TextExporter()

for e in elements:
e.accept(html_renderer)

print("---- Export as Text ----")
for e in elements:
e.accept(text_exporter)
```

32.8 Advantages

- Makes it easy to add new operations **without modifying existing classes**.
- Encourages separation of data structures and operations.
- Supports multiple operations on the same object hierarchy.
- Promotes the **Open/Closed Principle**.

32.9 Drawbacks

- Adding new element types requires updating all visitors.
- Increases the number of classes, which can make the design more complex.
- Double-dispatch mechanism may feel unintuitive at first.

32.10 Real-World Use Cases

- **Compilers & AST Traversal:** Applying semantic checks or optimizations to syntax trees.
- **Report Generators:** Rendering documents in different formats like PDF, HTML, or TXT.
- **GUI Frameworks:** Performing operations on widgets without changing their code.
- **Financial Systems:** Applying different tax rules or calculations on transaction objects.

32.11 Visitor vs. Strategy vs. Interpreter

- **Visitor:** Adds new operations without changing the element classes.
- **Strategy:** Allows swapping entire algorithms dynamically but doesn't traverse structures.
- **Interpreter:** Defines and evaluates grammars but often uses Visitor to extend evaluation.

32.12 Summary

- The **Visitor Pattern** separates operations from object structures.
- Makes it easy to introduce new behaviors without modifying existing classes.
- Particularly useful for ASTs, document renderers, report engines, and hierarchical data.
- Often paired with **Interpreter** and **Composite** for tree-based processing.

Visitor — Interview Blurb

Separates operations from the objects on which they operate by externalizing behavior into a visitor object. Each element "accepts" a visitor, allowing new operations to be added without modifying the element classes themselves. Ideal for applying multiple, unrelated algorithms (printing, evaluation, serialization) over stable object structures.

33 Behavioral Patterns Quiz with Answers & Explanations

This quiz reviews Command, State, Template, Iterator, Mediator, Memento, Visitor, Interpreter, and Chain of Responsibility patterns.

Q1. Command Pattern

Considering the Command pattern, which of the following are TRUE? (Select all correct)

- ✓ **The roles in Command Pattern include Client, Command, Invoker, and Receiver; the pattern can be simplified if Commands are allowed to execute actions directly**
- ✗ Lambda expressions are required to implement the Command pattern in Java
- ✓ **The Command pattern standardizes and encapsulates requests into command objects**
- ✓ **The Null Object pattern is used to provide a Command that does nothing, so the Client can use all commands without special concerns for null cases**

Explanation: Command encapsulates requests as objects, decoupling the sender from the receiver. Java lambdas can be used for convenience but are **not required**. Using a Null Object avoids special-case handling for missing commands.

Q2. State Pattern

Which of the following are TRUE from our discussion on state machines and the State pattern? (Select all correct)

- ✓ **In the OO State pattern, each state is represented by an implementation of a State interface (or inheritance from a State abstract class) that defines transitions as methods which all state instances must define actions (or lack of action) for**
- ✗ State transitions can only be controlled by the State classes, not the Context class
- ✓ **State machines are structured to identify states (the current system context) and transitions (events that may cause state changes)**
- ✓ **There are several ways to represent states in software: state- and event-centric state machines, table-driven state machines, and the OO State pattern are examples**

Explanation: The Context usually coordinates transitions between states, so the second statement is false. All other statements describe correct aspects of OO State patterns and finite state machines.

Q3. Template Pattern and `final`

Which of the following statements is **NOT true** regarding Template and the use of Java's `final`?

- ✗ Java's `final` keyword can also be used to create constants from variables and to prevent class inheritance
- ✗ To prevent a subclass from changing (overriding) a superclass method, Java has a keyword `final` that can be applied
- ✓ **Template patterns control variation of entire algorithms, whereas Strategy patterns control variation in the steps of an algorithm**
- ✗ Template demonstrates the Hollywood Principle ("don't call us, we'll call you") by having the template method in a high-level class invoke subclass methods

Explanation: Strategy patterns vary **entire algorithms**, whereas Template varies steps within a single overall algorithm — the reversed claim is incorrect.

Q4. Template Pattern Methods

Which of these method types is **NOT** included in the basic Template pattern definition?

- ✗ Abstract methods for which subclasses must provide implementations
- ✓ **Static methods that can be used without instantiating an object**
- ✗ Concrete (often final) methods subclasses must use as implemented
- ✗ Hook methods that subclasses can choose to override or use defaults for

Explanation: Template patterns rely on polymorphism and overridden methods, so static methods are outside its standard design.

Q5. Iterator Pattern

According to the UNC Iterator review, which of the following is **NOT correct**? (Select all correct)

- ✗ Three benefits or consequences of using iterator include: supporting variation in traversing an aggregate, simplifying the aggregate's interface, allowing multiple iterators to traverse an aggregate at once
- ✗ A robust iterator ensures that insertions or removals from the aggregate do not interfere with traversal
- ✗ In implementation of iterators, an iterator can be controlled externally by a client, or internally by the iterator itself
- ✓ **The default methods found in the Iterator examples include `Next()`, `Last()`, `IsDone()`, `CurrentItem()`**

Explanation: In Java, iterators do **not** have methods like `Last()` or `IsDone()`. The standard interface defines `hasNext()`, `next()`, and `remove()`.

Q6. Mediator Pattern

Considering the Mediator pattern, which of the following statements are TRUE? (Select all correct)

- ✗ Encapsulation of the communicating objects is broken because changes to communicating objects may require updating both the communicating object and the Mediator
- ✓ **The Mediator pattern provides centralized decision logic for a group of intercommunicating objects known as Colleagues**
- ✗ **Use of a mediator provides for increased cohesion**
- ✗ Use of a mediator provides increased coupling in the intercommunicating objects

Explanation: Mediator **reduces coupling** between objects and centralizes communication logic, which increases cohesion, not decreases it.

Q7. Memento Pattern (Fill-in-the-Blanks)

In the UML for the Memento pattern, the _____ produces and consumes _____ objects that contain a saved state instance; the _____ maintains the state instances for restoring on request.

Answer:

1. Originator
2. Memento
3. Caretaker

Explanation: - The **Originator** creates and restores snapshots of its internal state. - The **Memento** holds the saved state. - The **Caretaker** requests state saves and restores but never modifies the Memento.

Q8. Visitor, Interpreter, Chain of Responsibility

The _____ pattern provides for a series of handlers to address events or requests, the _____ pattern supports parsing formal grammars, and the _____ pattern uses a double-dispatch mechanism to apply methods to aggregates of similar object types without significant modification.

Answer:

1. Chain of Responsibility
2. Interpreter
3. Visitor

Explanation: - **Chain of Responsibility:** Sends a request through a series of potential handlers until one processes it. - **Interpreter:** Defines a grammar and provides an interpreter for language parsing. - **Visitor:** Uses double dispatch to separate algorithms from the objects they operate on.

Part VII
Composite Patterns

34 Model-View-Controller (MVC) Pattern

34.1 Learning Objectives

- Understand the intent and purpose of the **Model-View-Controller** (MVC) pattern.
- Learn how MVC separates responsibilities into distinct components.
- Explore the UML structure and analyze Java/Python implementations.
- Understand the relationship between MVC and patterns like Observer, Strategy, and Composite.
- Examine real-world applications, advantages, and trade-offs.

34.2 Introduction

The **Model-View-Controller (MVC)** pattern is an **architectural pattern** used to separate application logic, presentation, and user interaction into independent components:

- **Model:** Manages data, business rules, and application logic.
- **View:** Handles the presentation of information to the user.
- **Controller:** Interprets user input and coordinates between the Model and View.

  Intent: "Divide an application into three interconnected components to separate concerns, improve maintainability, and enable independent development of UI, logic, and data layers."

34.3 Motivation

Consider designing a **GUI-based shopping cart application**:

- The **Model** holds the list of items and their prices.
- The **View** displays items in the cart and total price.
- The **Controller** manages user actions like adding or removing products.

Without MVC:

- Data, UI, and event-handling logic would be tightly coupled.
- Making changes to the UI could break business logic.

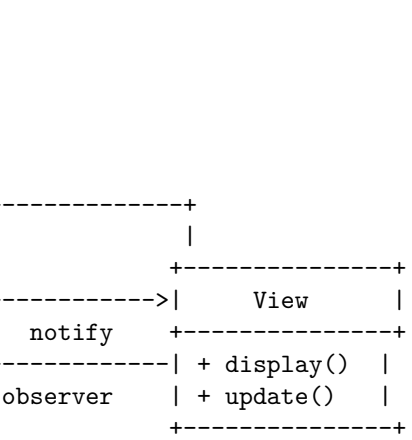
With MVC:

- The Model, View, and Controller are decoupled.
- The UI can change without modifying core logic.
- The Model can evolve independently of how data is displayed.

34.4 Key Concepts

- **Model:** Encapsulates business logic and notifies views of state changes.
- **View:** Subscribes to the model, retrieves data, and renders it to the user.
- **Controller:** Handles user input and translates it into model updates or view changes.

34.5 UML Structure



34.6 Java Implementation

Step 1: Define the Model

```
import java.util.ArrayList;
import java.util.List;

public class ShoppingCart {
 private List<String> items = new ArrayList<>();
 private List<CartView> observers = new ArrayList<>();

 public void addItem(String item) {
  items.add(item);
  notifyObservers();
 }

 public void removeItem(String item) {
  items.remove(item);
  notifyObservers();
 }

 public List<String> getItems() {
  return items;
 }

 public void attach(CartView observer) {
  observers.add(observer);
 }

 private void notifyObservers() {
  for (CartView observer : observers) {
   observer.update();
  }
 }
}
```

Step 2: Create the View

```
public class CartView {
 private ShoppingCart model;
```

```
 public CartView(ShoppingCart model) {
  this.model = model;
 }

 public void update() {
  System.out.println("Cart updated: " + model.getItems());
 }
}
```

Step 3: Implement the Controller

```
public class CartController {
 private ShoppingCart model;

 public CartController(ShoppingCart model) {
  this.model = model;
 }

 public void addItem(String item) {
  model.addItem(item);
 }

 public void removeItem(String item) {
  model.removeItem(item);
 }
}
```

Step 4: Use MVC in the Client

```
public class Main {
 public static void main(String[] args) {
  ShoppingCart model = new ShoppingCart();
  CartView view = new CartView(model);
  CartController controller = new CartController(model);

  model.attach(view);

  controller.addItem("Laptop");
  controller.addItem("Headphones");
  controller.removeItem("Laptop");
 }
}
```

34.7 Python Implementation

```
class ShoppingCart:
def __init__(self):
self.items = []
self.observers = []

def add_item(self, item):
self.items.append(item)
self._notify()

def remove_item(self, item):
self.items.remove(item)
```

```
self._notify()

def get_items(self):
return self.items

def attach(self, observer):
self.observers.append(observer)

def _notify(self):
for observer in self.observers:
observer.update()

class CartView:
def __init__(self, model):
self.model = model

def update(self):
print(f"Cart updated: {self.model.get_items()}")

class CartController:
def __init__(self, model):
self.model = model

def add_item(self, item):
self.model.add_item(item)

def remove_item(self, item):
self.model.remove_item(item)

# Client code
model = ShoppingCart()
view = CartView(model)
controller = CartController(model)

model.attach(view)

controller.add_item("Laptop")
controller.add_item("Headphones")
controller.remove_item("Laptop")
```

34.8 Advantages

- Separates concerns between data, UI, and interaction.
- Promotes reusability of Models and Views.
- Makes testing easier — controllers and models can be tested independently.
- Scales well for large applications.

34.9 Drawbacks

- Introduces more components, increasing initial complexity.
- Can become difficult to manage in very large projects without strict conventions.
- Controllers may become “fat” if too much logic is added.

34.10 Real-World Use Cases

- **Web Frameworks:** Django, Ruby on Rails, Spring MVC.
- **Frontend Frameworks:** Angular, React (Flux/Redux variants).
- **Desktop Applications:** JavaFX, Qt, and .NET MVC.
- **Game Development:** Unity's UI systems.

34.11 MVC vs. MVP vs. MVVM

- **MVC:** The Controller updates both Model and View directly.
- **MVP (Model-View-Presenter):** Presenter handles all updates; the View is passive.
- **MVVM (Model-View-ViewModel):** Used in data-binding frameworks where Views automatically reflect model changes.

34.12 Summary

- The **Model-View-Controller (MVC) Pattern** separates application logic, presentation, and interaction.
- Widely used in frameworks, GUIs, and web architectures.
- Promotes modularity, scalability, and testability.
- Variants like MVP and MVVM further refine responsibilities for modern development.

Model–View–Controller (MVC) — Interview Blurb

Divides an application into three collaborating components: **Model** (data and business logic), **View** (presentation layer), and **Controller** (input handler and coordinator). This separation of concerns decouples user interfaces from domain logic, allowing independent development, easier testing, and multiple synchronized views of the same data.

35 Model-View-Presenter (MVP) Pattern

35.1 Learning Objectives

- Understand the intent and purpose of the **Model-View-Presenter** (MVP) pattern.
- Learn how MVP improves upon the traditional MVC architecture.
- Explore the UML structure and study Java/Python implementations.
- Compare MVP with MVC and MVVM to understand design trade-offs.
- Examine real-world applications, advantages, and potential drawbacks.

35.2 Introduction

The **Model-View-Presenter (MVP)** pattern is an **architectural pattern** derived from **MVC** that aims to achieve better separation of concerns by making the **View passive** and delegating all orchestration logic to the **Presenter**.

> *Intent:* "Separate the responsibilities of managing data, presenting it, and handling user interactions to make applications more testable, maintainable, and modular."

MVP is widely used in GUI frameworks and mobile apps where:

- Views must remain lightweight and testable.
- Complex UI interactions require orchestration between components.
- We need to unit-test presentation logic without tying it to UI frameworks.

35.3 Motivation

Consider designing a **weather forecast app**:

- The **Model** fetches forecast data from an API.
- The **View** displays temperature, conditions, and humidity.
- The **Presenter** handles button clicks, updates the Model, and instructs the View what to display.

In traditional MVC:

- Controllers often interact directly with Views.
- Views may end up containing presentation logic, making them harder to test.

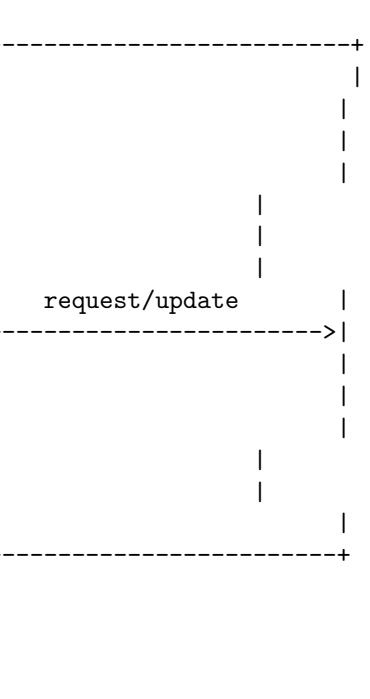
With MVP:

- The View only forwards user input to the Presenter.
- The Presenter decides **what** to fetch and **how** to present it.
- The Model stays clean and independent of UI.

35.4 Key Concepts

- **Model:** Encapsulates business logic and data handling, independent of the UI.
- **View:** Displays data to the user and forwards user actions to the Presenter.
- **Presenter:** Acts as the middleman between the Model and View, containing all orchestration logic.

35.5 UML Structure



35.6 Java Implementation

Step 1: Define the Model

```
public class WeatherModel {
 private String weather;

 public String fetchWeather() {
  // Simulate API call
  weather = "Sunny, 25°C";
  return weather;
 }
}
```

Step 2: Define the View Interface

```
public interface WeatherView {
 void displayWeather(String weather);
}
```

Step 3: Implement the Presenter

```
public class WeatherPresenter {
 private WeatherView view;
 private WeatherModel model;

 public WeatherPresenter(WeatherView view, WeatherModel model) {
  this.view = view;
  this.model = model;
 }

 public void onRefreshButtonClicked() {
  String weather = model.fetchWeather();
```

```
  view.displayWeather(weather);
 }
}
```

Step 4: Implement the Concrete View

```
public class ConsoleWeatherView implements WeatherView {
 private WeatherPresenter presenter;

 public ConsoleWeatherView() {
  WeatherModel model = new WeatherModel();
  presenter = new WeatherPresenter(this, model);
 }

 @Override
 public void displayWeather(String weather) {
  System.out.println("Weather: " + weather);
 }

 public void simulateUserAction() {
  presenter.onRefreshButtonClicked();
 }
}
```

Step 5: Use MVP in the Client

```
public class Main {
 public static void main(String[] args) {
  ConsoleWeatherView view = new ConsoleWeatherView();
  view.simulateUserAction();
 }
}
```

35.7 Python Implementation

```
class WeatherModel:
def fetch_weather(self):
return "Sunny, 25°C"

class WeatherView:
def display_weather(self, weather):
print(f"Weather: {weather}")

class WeatherPresenter:
def __init__(self, view, model):
self.view = view
self.model = model

def on_refresh_button_clicked(self):
weather = self.model.fetch_weather()
self.view.display_weather(weather)

# Client code
model = WeatherModel()
view = WeatherView()
presenter = WeatherPresenter(view, model)

presenter.on_refresh_button_clicked()
```

35.8 Advantages

- Better separation of concerns than MVC.
- The View is passive, making UI components easier to test.
- Presenter logic can be unit-tested independently.
- Supports parallel development: UI teams and backend teams can work independently.

35.9 Drawbacks

- Introduces additional boilerplate compared to MVC.
- Presenters can become "god objects" if overloaded with responsibilities.
- Requires careful coordination between View and Presenter for event handling.

35.10 Real-World Use Cases

- **Android Development:** MVP is commonly used in mobile app architectures.
- **Desktop GUI Frameworks:** Java Swing, .NET WinForms, and Qt.
- **Cross-Platform Apps:** Applications requiring strict separation between UI and logic.
- **Testing-Oriented Systems:** Environments prioritizing mock-driven development.

35.11 MVP vs. MVC vs. MVVM

- **MVC:** Controller manages both Model updates and View rendering; View can be active.
- **MVP:** Presenter completely controls View updates; View is passive and delegates everything.
- **MVVM:** Uses **data binding** between View and ViewModel, often seen in Angular, React, and WPF.

35.12 Summary

- The **Model-View-Presenter (MVP)** pattern improves testability and modularity over MVC.
- Presenter acts as the single point of coordination between Model and View.
- Widely used in GUI frameworks, Android development, and cross-platform applications.
- Often compared with MVC and MVVM; MVP is ideal when Views must remain simple and testable.

Model–View–Presenter (MVP) — Interview Blurb

Refines MVC by introducing a **Presenter** that handles all presentation logic and mediates between the **View** and **Model**. The View is passive—responsible only for rendering UI and forwarding user actions— while the Presenter retrieves data from the Model and updates the View. This improves testability and cleanly separates UI from business logic.

36 Model-View-ViewModel (MVVM) Pattern

36.1 Learning Objectives

- Understand the intent and purpose of the **Model-View-ViewModel** (MVVM) pattern.
- Learn how MVVM leverages data binding to synchronize Views and Models.
- Explore the UML structure and analyze simplified Java/Python-style implementations.
- Compare MVVM with MVC and MVP patterns.
- Examine real-world applications, advantages, and trade-offs.

36.2 Introduction

The **Model-View-ViewModel (MVVM)** pattern is an **architectural pattern** derived from MVP and MVC. It is designed to make applications **more modular, testable, and maintainable** by introducing the **ViewModel**, which exposes state and behavior to the View through **automatic data binding**.

> *Intent:* "Decouple UI components from business logic using a ViewModel that binds the Model's data to the View automatically."

MVVM is widely used in:

- Web frameworks like Angular, React, and Vue.js.
- Desktop frameworks like WPF, .NET MAUI, and Qt QML.
- Mobile frameworks like Flutter, SwiftUI, and Xamarin.

36.3 Motivation

Imagine designing a **real-time stock price dashboard**:

- The **Model** retrieves data from a stock market API.
- The **View** displays prices in a live-updating interface.
- The **ViewModel** exposes observable properties like `price`, `symbol`, and `trend`.

Without MVVM:

- Developers must manually synchronize Models and Views, introducing boilerplate.

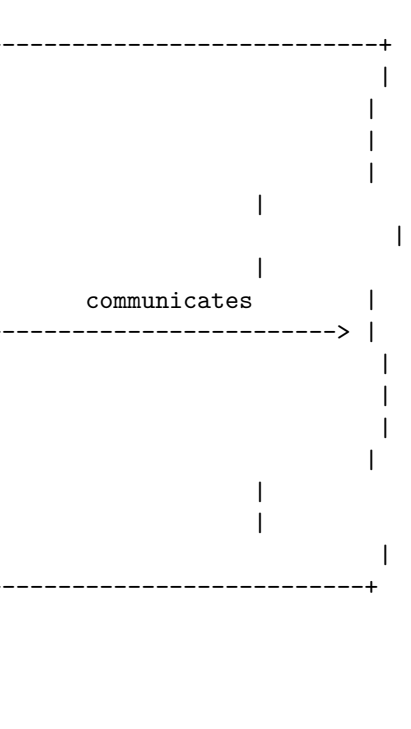
With MVVM:

- The View automatically updates when the Model changes.
- Two-way data binding ensures that user input updates the Model seamlessly.

36.4 Key Concepts

- **Model:** Handles data, business logic, and state persistence.
- **View:** Presents data to the user and reflects state changes from the ViewModel automatically.
- **ViewModel:** Acts as the bridge between the View and Model by exposing observable properties and commands.

36.5 UML Structure



36.6 Java Implementation (Simplified WPF-Style Example)

Step 1: Define the Model

```
public class Stock {
 private String symbol;
 private double price;

 public Stock(String symbol, double price) {
  this.symbol = symbol;
  this.price = price;
 }

 public String getSymbol() { return symbol; }
 public double getPrice() { return price; }
 public void setPrice(double price) { this.price = price; }
}
```

Step 2: Define the ViewModel with Observable Properties

```
import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;

public class StockViewModel {
 private Stock stock;
 private PropertyChangeSupport support;

 public StockViewModel(Stock stock) {
  this.stock = stock;
  this.support = new PropertyChangeSupport(this);
 }
```

```
 public void setPrice(double price) {
  double oldPrice = stock.getPrice();
  stock.setPrice(price);
  support.firePropertyChange("price", oldPrice, price);
 }

 public double getPrice() {
  return stock.getPrice();
 }

 public void addPropertyChangeListener(PropertyChangeListener listener) {
  support.addPropertyChangeListener(listener);
 }
}
```

Step 3: Implement the View (Auto-Binding Simulation)

```
public class StockView {
 public StockView(StockViewModel vm) {
  vm.addPropertyChangeListener(evt -> {
   System.out.println("Updated Price: " + evt.getNewValue());
  });
 }
}
```

Step 4: Use MVVM in the Client

```
public class Main {
 public static void main(String[] args) {
  Stock stock = new Stock("AAPL", 180.0);
  StockViewModel vm = new StockViewModel(stock);
  StockView view = new StockView(vm);

  vm.setPrice(185.5);
  vm.setPrice(190.0);
 }
}
```

36.7 Python Implementation (Reactive MVVM-Style Example)

```
class Observable:
def __init__(self, value=None):
self._value = value
self._observers = []

def bind(self, callback):
self._observers.append(callback)

def set(self, value):
self._value = value
for callback in self._observers:
callback(value)

def get(self):
return self._value
```

```
class Stock:
def __init__(self, symbol, price):
self.symbol = symbol
self.price = price

class StockViewModel:
def __init__(self, stock):
self.stock = stock
self.price = Observable(stock.price)

def update_price(self, new_price):
self.stock.price = new_price
self.price.set(new_price)

class StockView:
def __init__(self, view_model):
view_model.price.bind(self.display_price)

def display_price(self, price):
print(f"Updated Price: {price}")

# Client code
stock = Stock("AAPL", 180.0)
vm = StockViewModel(stock)
view = StockView(vm)

vm.update_price(185.5)
vm.update_price(190.0)
```

36.8 Advantages

- **Automatic data binding** between View and ViewModel reduces boilerplate.
- High testability since ViewModels are UI-independent.
- Promotes clear separation of concerns between UI, state, and logic.
- Enables faster parallel development between frontend and backend teams.

36.9 Drawbacks

- Two-way data binding can introduce subtle bugs if misused.
- Increased memory overhead when maintaining observables for many properties.
- Requires framework support (e.g., Angular, WPF, Vue) for efficient implementation.

36.10 Real-World Use Cases

- **Web Frameworks:** Angular, Vue.js, React with Redux or MobX.
- **Desktop Apps:** Microsoft WPF, Qt QML, Electron apps.
- **Mobile Apps:** Flutter, Xamarin, SwiftUI, and Jetpack Compose.
- **IoT Dashboards:** Real-time monitoring interfaces.

36.11 MVVM vs. MVC vs. MVP

- **MVC:** Controller manages updates; data binding is manual.
- **MVP:** Presenter handles orchestration but still requires explicit View updates.
- **MVVM:** Relies on **automatic binding** between View and ViewModel for seamless synchronization.

36.12 Summary

- The **MVVM Pattern** introduces automatic two-way data binding between View and ViewModel.
- Separates concerns cleanly while reducing synchronization boilerplate.
- Ideal for modern frameworks where UI state changes frequently.
- Often used in Angular, Vue.js, React, Flutter, and WPF architectures.

Model–View–ViewModel (MVVM) — Interview Blurb

Separates UI logic into a **ViewModel** that exposes observable data and commands to the **View**, which binds to them declaratively. The ViewModel holds no UI references, allowing unit testing of behavior and leveraging data binding so that UI updates automatically when the underlying Model changes. Ideal for frameworks supporting reactive or declarative UI architectures (e.g., WPF, Angular, SwiftUI).

37 Comprehensive OO Patterns Quiz with Answers

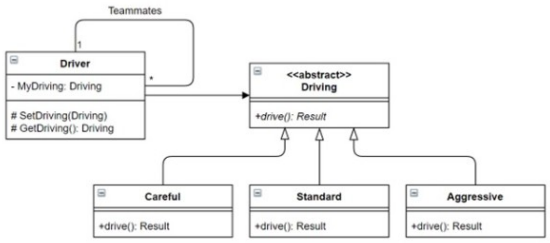
Question 1 When you meet with the team at the start of the project, you remind them you'll be using OO Patterns in the design cycle, and you review what benefits OO patterns can provide. Which of these are correct points you can make with the team on OO pattern benefits? (Select all correct)

- ✓ Gaining experience reuse from things learned by previous OO designers
- Getting code reuse from OO pattern solutions
- ✓ Help in identifying common design problems and standard solutions
- ✓ Shared vocabulary to discuss design approaches

Question 2 One of the first things that happens in the game flow is the selection and creation of Drivers, which is represented as an abstract superclass with four possible concrete subclasses. In a code review, you notice that a developer has just cut all the `new` statements for the drivers from a piece of code into a separate method, but you then ask them to develop a family of creation methods for each driver subclass.

- The pattern the developer first used was ⟨ **Simple Factory** ⟩.
- The pattern you're asking to be used is ⟨ **Factory** ⟩.
- What principle focused on depending on abstraction supports the pattern you're asking them to use? ⟨ **Dependency Inversion** ⟩.

Question 3 A developer shows you this UML class diagram for assigning a Driving algorithm to Driver objects:



- What pattern is being used between Driver and Driving? ⟨ **Strategy** ⟩
- That pattern favors ⟨ **delegation** ⟩ over inheritance.

Question 4 The program allows for a variety of race cars, each of which is made up of selected parts – engine, suspension, body, tires, fuel system, decorations, etc. If each car is a combination of these parts, and the parts are each represented by an object or objects of their own, which pattern is probably best for building race cars composed of the parts?

- Decorator
- Composite
- ✓**Factory**

Question 5 Reviewing legacy designs for the system, you find this UML:

- The `ErrorHandler` UML represents an example of a Singleton pattern: ⟨ **True** ⟩
- `ErrorHandler` as specified could be created with Eager Instantiation: ⟨ **True** ⟩
- The following Java statement could be used outside of the `ErrorHandler` class, based on the UML diagram: `ErrorHandler eh = new ErrorHandler();` ⟨ **False** ⟩

Question 6 The game uses an entire library of audio routines divided into six inter-related frameworks. You'd like to provide an interface to this code that just focuses on the methods needed to set up audio for the game and hide all the complexities of the rest of the subsystem. Which pattern should you use?

- Template
- Strategy
- Adapter
- **Facade**

Question 7 Once you begin to implement some driving algorithms, you can see that some of the methods for the steps of the algorithms are the same for each algorithm, some must be specialized for each method implementation, and some may use a default implementation but could be overridden.

- What pattern applies to this design of algorithms with varying steps? ⟨ **Template** ⟩
- What do we call the methods with a default impl that we could choose to override? ⟨ **Hook** ⟩

Question 8 Cars can be added and removed from the races while the races are happening. The cars that are running may need to receive event notifications on track conditions, weather changes, etc. but other cars that are not racing do not need to know about these things. What pattern would we use to allow cars that want events to sign up for them?

- ✓**Observer**
- Template
- Command
- Composite

Question 9 There are two different graphics engines you have to have connected to your race car logic client. There is an existing interface to one of them that has a design you like, but the second engine has a different interface doing essentially the same operations. If you want to use the existing client interface without changing it for both graphics engines, what pattern applies?

- Command
- Observer
- ✓**Adapter**
- Façade

Question 10 For some game interaction, your team decides to look at using the Command pattern. Usually, the UML for the Command pattern is presented with four primary elements, a Client, Commands, an Invoker, and a Receiver. Which of these four elements have we noted are optional in the implementation of a Command pattern-based process? (Select all that apply)

- ✓Invoker
- Command
- ✓Receiver
- Client

Question 11 One of your team sees some multiple inheritance in some C++ legacy code. Based on the rules discussed in OOAD for multiple inheritance, consider the following:

- The first rule is to assume the multiple inheritance is wrong, and to prove otherwise: ⟨ **True** ⟩
- A `Car` is inheriting from classes `Engine` and `Tires`, this is likely correct: ⟨ **False** ⟩
- A `Car` is inheriting from classes `Vehicle` and `Metallics`, this is likely correct: ⟨ **True** ⟩

Question 12 Reviewing some pattern-based design in your application, you try to remember which of these statements is true (select all correct):

- ✓A Finite State Machine must represent each possible state and transition in a given process
- Extrinsic object state refers to information held in an object that can be read but is not usually changed by outside clients
- ✓The steps in the "Thinking in Patterns" design approach are to identify patterns for the problem space, analyze and apply patterns by context (this step may repeat and may find other patterns), and finally add detail
- The Chain of Responsibility pattern automatically allows for the case where no handler routine recognizes a parsed request

Question 13 Select the best patterns for these four hypothetical scenarios:

- (1) I am making a CAD system, and I need to implement a save/undo/redo functionality so I can keep different states of the drawing in case someone wants to go back to an earlier version: ⟨ **Memento** ⟩
- (2) When my message arrives at the cloud, I have a set of prioritized serverless handlers I want to pass the message to; once one of them handles the message, we can wait for the next message to arrive: ⟨ **Chain of Responsibility** ⟩

- (3) I have an embedded device that goes into several processing modes in a cycle – a sleep mode, a checkin mode, a sensing mode, a low battery mode, and an error mode – all caused by timer events or changes in the device or environment: ⟨ **State** ⟩
- (4) I have an application with multiple web pages, a database for storing application state information, another database for transaction data, and some complex business logic depending on what pages are up and what selections are made: ⟨ **MVC** ⟩

Question 14 Select the best patterns for these four hypothetical scenarios:

- (1) I will be connecting to many remote servers to get some standard information, but I would like the client application to work as if it is talking to something local: ⟨ **Proxy** ⟩
- (2) There is a lot of crosstalk between these six objects, maybe I can make a central object they can all connect to instead; even though that central object will know a lot about the operations, it will be a lot less coupled: ⟨ **Mediator** ⟩
- (3) If I instantiate each of these objects with the normal constructor, I have to make six database transactions and do four complicated calculations; but if I just clone the first object to make the other objects I can avoid all that: ⟨ **Prototype** ⟩
- (4) When I create this object, I have a bunch of different options I can use to initialize various properties of the object; I want to be able to select just the options I need every time I make a specific new object: ⟨ **Builder** ⟩

Part VIII
Practice & Architecture

38 Refactoring

38.1 Learning Objectives

- Understand **what refactoring is** (Fowler): changing the *internal structure* of code to improve its design while preserving external behavior.
- Explain **why refactoring matters**: improves design, understandability, defect discovery, development speed, and feature agility; supports "litter-pickup" (Boy Scout Rule).
- Apply a **disciplined workflow**: Red–Green–Refactor, tiny steps, strong tests.
- Recognize **when** to refactor (e.g., Rule of Three) and **when not** to.
- Anticipate **problems with refactoring**: published interfaces, branching models, testing constraints, legacy code.

38.2 Definition (Fowler)

> *Refactoring* is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.

Two essentials: **preserve behavior** and **improve design**.

38.3 Why Refactor?

- **Improves design:** Reduces duplication, clarifies responsibilities, increases cohesion, lowers coupling.
- **Easier to understand:** Clearer names, smaller functions, simpler dependencies shorten onboarding and review time.
- **Helps find bugs:** While extracting/isolating logic, hidden assumptions surface; complex conditionals become testable.
- **Makes programming faster:** Clean code is cheaper to modify; the cost of small refactors amortizes quickly.
- **Easier to add features:** Refactoring exposes extension points (*seams*), transforming "hard-to-change" areas into flexible ones.
- **Litter-pickup (Boy Scout Rule):** Each change leaves the code a little cleaner (rename a var, extract a method, remove dead code).

38.4 Core Principles

- **Behavior must not change.** Use tests as a safety net; keep steps tiny and reversible.
- **Small, incremental steps.** One refactoring at a time; commit frequently.
- **Red–Green–Refactor:** Write/see a failing test (Red), make it pass (Green), improve the design (Refactor) with all tests staying green.
- **Name things well.** Renaming for intent-revealing APIs is one of the highest-value micro-refactors.

38.5 Common Smells & Typical Remedies

Code Smell	Representative Refactorings (Fowler)
Long Function	Extract Function; Introduce Explaining Variable; Replace Temp with Query
Large Class	Extract Class; Extract Superclass; Collapse Hierarchy
Duplicated Code	Extract Function; Pull Up Method; Form Template Method
Long Parameter List	Introduce Parameter Object; Preserve Whole Object; Replace Parameter with Query
Feature Envy	Move Method; Extract Class
Divergent Change / Shotgun Surgery	Consolidate by Extract Class/Module; Move Method/Field
Primitive Obsession	Introduce Value Object; Replace Primitive with Object; Encapsulate Record
Inappropriate Intimacy	Move Method/Field; Hide Delegate; Introduce Mediator
Switch/Type Conditionals	Replace Conditional with Polymorphism; Strategy
Temporary Field / Data Clumps	Extract Class; Introduce Parameter Object
Dead Code	Remove Dead Code
Speculative Generality	Collapse Hierarchy; Inline Class

38.6 Mini Catalog (a few high–leverage refactorings)

- **Extract Function/Method:** Move a coherent chunk into a well-named function to reduce cognitive load.
- **Rename Symbol:** Clarify intent; most IDEs make this safe across a codebase.
- **Introduce Parameter Object:** Replace long parameter lists with a small value object.
- **Move Method/Field:** Cohesion: place behavior next to data it uses most.
- **Replace Conditional with Polymorphism:** Convert large switch/case on type/flag into subtype-specific behavior (Strategy/State).

38.7 Refactoring Workflow Example

Before (Java)

```
double price(Order order) {
 double total = 0;
 for (LineItem it : order.items()) {
  double base = it.quantity * it.unitPrice;
  if (it.category.equals("bulk") && it.quantity > 100) {
   base *= 0.9;
  }
  total += base;
 }
 if (order.country.equals("IE")) {
  total *= 1.23;
 }
 return total;
}
```

After (behavior preserved, design improved)

```
double price(Order order) {
 double subtotal = subtotal(order);
 return applyTax(subtotal, order.country());
}

double subtotal(Order order) {
 return order.items().stream()
 .mapToDouble(this::lineTotal)
 .sum();
}

double lineTotal(LineItem it) {
 double base = it.quantity() * it.unitPrice();
 return isBulk(it) ? base * 0.9 : base;
}

boolean isBulk(LineItem it) {
 return it.category().equals("bulk") && it.quantity() > 100;
}

double applyTax(double amount, String country) {
 return country.equals("IE") ? amount * 1.23 : amount;
}
```

Benefits: smaller functions, testable units (`isBulk`, `applyTax`), names convey intent, easier to change tax rules or pricing.

38.8 When to Refactor (Rules of Thumb)

- **Rule of Three:** The third time you do something similar, refactor (extract, generalize, or template).
- **Make the change easy, then make the easy change:** If adding a feature is difficult because of structure, refactor first to create a seam.
- **When you touch code anyway:** Opportunistic "litter-pickup" while fixing a bug or adding a feature.
- **When a smell slows you down:** If a name or layout makes you pause, pay the small tax now.
- **Pre-commit polish:** Tiny safety refactors (rename, extract) that reduce reviewer friction.

38.9 When *Not* to Refactor

- **No pressing modification need:** If messy code is stable and untouched, a refactor might not pay back the risk/cost right now.
- **Right before a deadline/release:** Structural changes increase risk; prefer minimal, targeted fixes unless tests are very strong.
- **Unstable requirements:** If the target is moving, defer large refactors until direction settles.
- **Weak safety net:** With poor or no tests, favor characterization tests first (see Legacy Code section) before altering structure.

38.10 Problems & Pitfalls in Refactoring

Published Interfaces (Public APIs) Refactoring can be *breaking* for customers. Use semantic versioning, deprecate with clear migration paths, maintain adapters/shims, and avoid changing observable behavior or API shapes without a plan.

Branches & Merges Large refactors across long-lived branches create painful conflicts. Prefer **trunk-based development**, **branch by abstraction**, and **incremental** refactors behind feature flags. Keep changes *small and frequent*.

Testing Refactoring relies on tests to prove behavior is unchanged. Gaps in coverage erode confidence. Prioritize fast, reliable tests: unit tests around extracted functions, and a thin layer of integration/end-to-end tests.

Legacy Code Legacy often lacks tests and has tangled dependencies.

- **Characterization Tests:** Capture current behavior (even if odd) to protect it during refactor.
- **Seams (Fowler/Feathers):** Introduce points (interfaces, dependency injection) where behavior can be substituted for testing.
- **Strangler Fig Pattern:** Incrementally wrap and replace legacy components with new ones, routing traffic over time.
- **Boy Scout Cleanup:** Continuous micro-improvements rather than big-bang rewrites.

38.11 Team Practices & Tooling

- **IDE automated refactorings:** Rename, Extract, Move, Inline are safer via tooling.
- **Static analysis/linters:** Highlight smells (duplication, complexity, dead code).
- **Code review culture:** Celebrate small refactors; block risky "drive-by" large rewrites without tests.
- **Metrics with caution:** Track complexity, coupling, coverage—but optimize for developer flow and change safety, not numbers alone.

38.12 Summary

- Refactoring (per Fowler) improves internal design *without* changing behavior.
- It makes code easier to understand, exposes bugs, accelerates development, and eases feature work—and supports continual "litter-pickup."
- Use tests, small steps, and a steady cadence. Apply the Rule of Three and refactor when code slows you down.
- Avoid risky refactors when there's no need, before deadlines, or without a safety net.
- Plan for API stability, branching strategy, testing, and legacy constraints to reduce refactoring risk.

39 Dependency Injection and Inversion of Control

39.1 Learning Objectives

- Understand the concepts of **Dependency Injection (DI)** and **Inversion of Control (IoC)**.
- Learn why DI promotes **loose coupling** and testable, maintainable systems.
- Differentiate between DI techniques: constructor, setter, and interface injection.
- Recognize the role of IoC containers and frameworks.
- Identify benefits, trade-offs, and best practices.

39.2 Introduction

In traditional object-oriented programming, classes create and manage their own dependencies. This leads to **tight coupling**, making systems brittle and difficult to test.

> *Martin Fowler:* "Dependency Injection is a specific form of Inversion of Control where the responsibility of obtaining a service is delegated to an external actor."

Key idea: Instead of classes creating their own dependencies, they *receive* them from the outside, often via constructors or setters.

39.3 The Problem: Tight Coupling

Example (Before DI)

```
public class UserService {
 private EmailService emailService = new EmailService();

 public void registerUser(User user) {
  // save user...
  emailService.sendWelcomeEmail(user);
 }
}
```

Issues:

- **Hard to test:** Cannot easily mock `EmailService`.
- **Difficult to replace implementations:** Changing to `SMSService` means editing `UserService`.
- **Hidden dependencies:** Consumers cannot see or control required services.

39.4 Inversion of Control (IoC)

Inversion of Control refers to the broader principle that objects should not create their dependencies but instead **receive them** from an external coordinator.

- In traditional systems, your code *calls* libraries and manages control flow.
- Under IoC, the *framework or container* controls the program's wiring and calls your components.

Examples of IoC beyond DI

- GUI frameworks: Event loops call your handlers, not the other way around.
- Web frameworks: The framework dispatches requests to your controllers.
- Test runners: The framework invokes your test methods.

Dependency Injection is **one specific implementation** of IoC: the control of dependency creation is inverted.

39.5 Dependency Injection (DI)

DI is the process of supplying an object with its dependencies rather than letting it construct them itself.

Example (After DI, using Constructor Injection)

```
public class UserService {
 private final EmailService emailService;

 public UserService(EmailService emailService) {
  this.emailService = emailService;
 }

 public void registerUser(User user) {
  // save user...
  emailService.sendWelcomeEmail(user);
 }
}

EmailService emailService = new EmailService();
UserService userService = new UserService(emailService);
```

Benefits:

- **Loosely coupled:** `UserService` depends only on the `EmailService` *interface*, not a concrete implementation.
- **Testable:** You can substitute a fake or mock implementation.
- **Explicit dependencies:** Consumers know exactly what `UserService` requires.

39.6 How DI Enables Loosely Coupled Systems

- Classes communicate via *interfaces* rather than concrete types.
- Object creation is delegated to higher-level orchestrators (factories or frameworks).
- Components can evolve independently since implementations can be swapped without changing clients.
- Improves maintainability: developers modify behavior by wiring new objects, not rewriting internals.

39.7 Types of Dependency Injection

39.7.1 1. Constructor Injection (Preferred)

- Dependencies are provided through the constructor.
- Makes required dependencies explicit and enforces immutability.

```
public class PaymentProcessor {
 private final PaymentGateway gateway;

 public PaymentProcessor(PaymentGateway gateway) {
  this.gateway = gateway;
 }
}
```

39.7.2 2. Setter Injection

- Dependencies are provided via public setters.
- More flexible but optional dependencies must be handled carefully.

```
public class PaymentProcessor {
 private PaymentGateway gateway;

 public void setGateway(PaymentGateway gateway) {
  this.gateway = gateway;
 }
}
```

39.7.3 3. Interface Injection

- Less common; the dependency is passed via a method defined on an interface.
- Typically handled by frameworks, not manually.

39.8 IoC Containers and Frameworks

In larger systems, **IoC containers** manage object creation, configuration, and injection automatically.

- Popular frameworks: Spring, Guice, Dagger, .NET Core DI.
- Developers declare dependencies (via annotations or configuration).
- The container resolves the dependency graph and wires components.

Example (Spring DI)

```
@Service
public class UserService {
 private final EmailService emailService;

 @Autowired
 public UserService(EmailService emailService) {
  this.emailService = emailService;
 }
}
```

Here, the Spring container instantiates `EmailService` and injects it into `UserService` automatically.

39.9 Benefits of Dependency Injection

- **Loose coupling:** Classes depend on abstractions, not implementations.
- **Testability:** Easy to substitute mocks or stubs in tests.
- **Reusability:** Shared components work across multiple contexts.
- **Configurable:** Behavior changes without rewriting code—swap components via configuration.
- **Scalability:** Especially powerful when combined with IoC frameworks that manage lifecycles.

39.10 Challenges and Pitfalls

- **Over-engineering:** Small projects may not benefit from DI frameworks; manual wiring might be simpler.
- **Steeper learning curve:** IoC containers add complexity; debugging dependency graphs can be tricky.
- **Hidden dependencies:** Poorly documented injection configurations can obscure actual runtime wiring.
- **Performance:** Large DI frameworks can introduce startup overhead.

39.11 Best Practices

- Prefer **constructor injection** for required dependencies.
- Depend on **interfaces**, not concrete implementations.
- Keep IoC configuration close to the composition root.
- Avoid deep dependency graphs; refactor services that need too many collaborators.
- Use lightweight DI manually for small apps; consider frameworks for large, evolving systems.

39.12 Summary

- **Inversion of Control** delegates dependency creation to an external orchestrator.
- **Dependency Injection** is one implementation of IoC that provides dependencies from outside.
- DI promotes **loose coupling**, better testability, maintainability, and flexibility.
- IoC containers simplify wiring in large systems, but come with trade-offs.
- Focus on small, incremental adoption—design APIs around interfaces, inject collaborators, and evolve toward DI when complexity demands it.

40 Reflection in OOAD

40.1 Learning Objectives

- Understand what **reflection** is and why it matters in object-oriented systems.
- Learn how reflection enables dynamic inspection, invocation, and modification of objects.
- Examine its uses in OOAD: frameworks, dependency injection, meta-programming, and runtime adaptability.
- Understand trade-offs: performance, security, and maintainability.

40.2 Introduction

Reflection is the ability of a program to **examine, analyze, and modify its own structure, behavior, and metadata at runtime**.

In OOAD, reflection provides a mechanism for:

- Inspecting classes, methods, fields, and annotations dynamically.
- Invoking methods or creating objects without knowing their names at compile time.
- Building highly flexible, reusable frameworks and architectures.

Reflection underpins many modern OOAD practices, especially in frameworks like **Spring**, **Hibernate**, **JUnit**, and **.NET Core**.

40.3 Core Concepts in Reflection

- **Type Introspection:** Discover class structure at runtime (fields, methods, constructors).
- **Dynamic Instantiation:** Create objects based on class names provided at runtime.
- **Dynamic Invocation:** Call methods or set fields without compile-time knowledge.
- **Metadata Inspection:** Read annotations or attributes attached to classes and methods.

40.4 Reflection in OO Languages

40.4.1 Java Example

```
// Inspecting Class Metadata
Class<?> cls = Class.forName("com.example.Customer");

// Print class name
System.out.println("Class: " + cls.getName());

// List all declared methods
for (Method method : cls.getDeclaredMethods()) {
 System.out.println("Method: " + method.getName());
}

// Create an instance dynamically
Object customer = cls.getDeclaredConstructor().newInstance();

// Invoke a method dynamically
Method setName = cls.getDeclaredMethod("setName", String.class);
setName.invoke(customer, "Alice");
```

40.4.2 Python Example

```
class Customer:
def greet(self):
print("Hello!")

# Inspect attributes and methods
print(dir(Customer))

# Create an instance dynamically
customer = Customer()
method = getattr(customer, "greet")
method()  # Invokes greet() dynamically
```

These capabilities make reflection central to **meta-programming**, enabling software to adapt to runtime conditions without prior compile-time knowledge.

40.5 Applications of Reflection in OOAD

1. **Framework Development**
   - Spring and Guice use reflection for **Dependency Injection**.
   - JUnit discovers and invokes test methods dynamically.
2. **Object-Relational Mapping (ORM)**
   - Hibernate inspects entity annotations to map objects to database tables.
3. **Plugin Architectures**
   - Dynamically load and execute user-provided modules without recompilation.
4. **Serialization and Deserialization**
   - Frameworks like Jackson or Gson use reflection to map fields to JSON/XML dynamically.
5. **Testing and Mocking**
   - Unit testing frameworks use reflection to automatically locate test classes and methods.
6. **Dynamic Behavior in Distributed Systems**
   - Microservices frameworks inspect annotations to expose REST endpoints dynamically.

40.6 Reflection and Loose Coupling in OOAD

Reflection supports **Inversion of Control** and **Dependency Injection**, making software **loosely coupled**:

- Code depends on **abstractions**, not concrete implementations.
- Frameworks wire dependencies at runtime without hardcoded references.
- Enables flexible architectures where components can evolve independently.

40.7 Advantages of Using Reflection

- **Runtime Flexibility:** Create dynamic, extensible systems where behavior adapts without recompilation.
- **Framework Enablement:** Supports ORMs, DI frameworks, serializers, and testing libraries.
- **Supports Plugin Architectures:** Load modules dynamically without altering existing code.
- **Meta-Programming Power:** Allows programs to reason about and modify themselves.

40.8 Drawbacks and Trade-offs

- **Performance Overhead:** Reflection-based calls are slower than direct invocation.
- **Weaker Type Safety:** Errors that could be caught at compile time now manifest at runtime.
- **Reduced Readability:** Dynamically invoked code can be harder to trace and debug.
- **Security Concerns:** Reflective access to private fields/methods can bypass encapsulation.
- **Overuse Risk:** Excessive reflection can make systems brittle and difficult to maintain.

40.9 Best Practices

- Use reflection for **framework-level infrastructure**, not for everyday business logic.
- Always validate reflective calls against annotations or naming conventions to avoid runtime errors.
- Prefer **dependency injection** and **interfaces** where possible to minimize reliance on reflection.
- Wrap reflective calls in utility classes to centralize usage and reduce scattering.
- Consider caching reflective lookups to reduce performance penalties.

40.10 Summary

- **Reflection** enables runtime inspection and dynamic modification of program behavior.
- It powers frameworks, ORMs, serializers, dependency injection, and plugin systems.
- In OOAD, reflection supports designing **loosely coupled, extensible systems**.
- It comes with costs: weaker type safety, performance overhead, security risks, and potential maintainability issues.
- Use reflection *strategically*—for infrastructure, not core domain logic.

41 Project Management in OOAD

41.1 Learning Objectives

- Understand the role of **project management** in OOAD-based software engineering.
- Compare traditional and modern development methodologies.
- Explore **Agile**, **Scrum**, **Kanban**, **DevOps**, and **DevSecOps**.
- Understand how methodology selection impacts object-oriented design and architecture.

41.2 Introduction

In Object-Oriented Analysis and Design (OOAD), project management is essential for:

- Aligning stakeholder goals with the software development lifecycle.
- Coordinating analysis, design, implementation, and testing phases.
- Ensuring timely delivery of maintainable, extensible object-oriented systems.

Various methodologies can be adopted depending on project scale, risk tolerance, and desired flexibility. Broadly, these approaches fall into two categories: **traditional (plan-driven)** and **adaptive (Agile-based)**.

41.3 Waterfall Project Management

Waterfall is a linear, sequential software development methodology. Each phase must be completed before moving on.

41.3.1 Key Characteristics

- Phases: Requirements → Design → Implementation → Testing → Deployment → Maintenance.
- Documentation-heavy and predictable.
- Changes are expensive once development starts.

41.3.2 Pros

- Well-suited for projects with **stable, well-defined requirements**.
- Easy progress tracking due to fixed milestones.
- Strong emphasis on planning and documentation.

41.3.3 Cons

- Limited flexibility for evolving requirements.
- Testing occurs late, increasing defect cost.
- Often mismatched with modern OOAD's iterative prototyping mindset.

41.4 Agile Project Management

Agile is an iterative, incremental approach focusing on delivering working software quickly while embracing change.

41.4.1 Key Principles (Agile Manifesto)

- Individuals and interactions over processes and tools.
- Working software over comprehensive documentation.
- Customer collaboration over contract negotiation.
- Responding to change over following a rigid plan.

41.4.2 Agile in OOAD

Agile aligns naturally with OOAD:

- Emphasizes **incremental delivery** of features, often represented as evolving classes and components.
- Supports iterative refinement of object models and architecture.
- Encourages close developer-stakeholder collaboration for dynamic requirement discovery.

41.4.3 Pros

- Continuous delivery of value.
- Supports frequent course correction.
- Fits well with test-driven development (TDD) and refactoring cycles.

41.4.4 Cons

- Requires mature, self-organizing teams.
- Scope creep risk if backlog control is poor.
- Less suitable for highly regulated environments needing heavy documentation.

41.5 Scrum Framework

Scrum is the most widely adopted Agile framework. It uses time-boxed development cycles called **sprints** (typically 2–4 weeks).

41.5.1 Roles

- **Product Owner:** Defines vision, prioritizes backlog, interfaces with stakeholders.
- **Scrum Master:** Facilitates processes, removes blockers, ensures adherence to Scrum practices.
- **Development Team:** Self-organizing, cross-functional engineers implementing features.

41.5.2 Core Artifacts

- **Product Backlog:** Ordered list of features, user stories, and enhancements.
- **Sprint Backlog:** Subset of backlog items committed to during a sprint.
- **Increment:** Working, tested code delivered at the end of each sprint.

41.5.3 Events

- Sprint Planning
- Daily Stand-ups
- Sprint Reviews
- Sprint Retrospectives

Scrum integrates seamlessly with OOAD by enabling iterative refinement of **class hierarchies, object models, and interfaces** sprint by sprint.

41.6 Kanban Framework

Kanban is a visual Agile methodology focused on optimizing workflow and reducing cycle time.

41.6.1 Key Principles

- Visualize work using a **Kanban board** (e.g., `To Do → In Progress → Testing → Done`).
- Limit work-in-progress (**WIP**) to avoid bottlenecks.
- Continuously improve flow based on metrics like throughput and lead time.

41.6.2 Benefits

- No time-boxed iterations; work is delivered continuously.
- Flexible and lightweight—easy to adopt incrementally.
- Ideal for maintenance-heavy or operational OOAD projects where change requests flow continuously.

41.7 DevOps in OOAD

DevOps integrates **software development (Dev)** and **operations (Ops)** to shorten feedback cycles, improve reliability, and automate deployment pipelines.

41.7.1 Key Practices

- **Continuous Integration (CI):** Regularly merging changes into a shared repository.
- **Continuous Delivery/Deployment (CD):** Automating builds, tests, and deployments.
- **Infrastructure as Code (IaC):** Managing infrastructure through version-controlled configurations.
- **Monitoring and Observability:** Using real-time telemetry to ensure system health.

41.7.2 Relevance to OOAD

- OOAD artifacts (class hierarchies, components, and APIs) must integrate seamlessly into automated pipelines.
- Frequent refactoring and test-driven workflows work well with DevOps' CI/CD models.

41.8 DevSecOps: Extending DevOps with Security

DevSecOps extends DevOps by embedding security practices directly into the software delivery lifecycle.

41.8.1 Principles

- **Shift Left on Security:** Incorporate security testing early in the design and development process.
- Automated vulnerability scanning integrated into CI/CD pipelines.
- Secure coding standards applied throughout OOAD phases.

41.8.2 Benefits

- Proactively identifies and mitigates risks.
- Reduces costly late-stage fixes.
- Ensures compliance for regulated environments.

41.9 Comparison of Approaches

Methodology	Philosophy	Best For	Drawbacks
Waterfall	Plan-driven, sequential	Stable requirements, highly regulated domains	Inflexible, high late-stage defect cost
Agile	Iterative, adaptive	Dynamic requirements, rapid delivery, frequent feedback	Scope creep risk, requires strong team collaboration
Scrum	Time-boxed sprints, self-organizing teams	Product-focused, fast-paced development	Overhead from ceremonies and role dependencies
Kanban	Visual flow, WIP limits	Operational teams, continuous requests, dynamic workloads	Less structure; teams may drift without clear priorities
DevOps	Automate, integrate Dev + Ops	Teams needing faster releases and operational feedback	Cultural shift required; toolchain complexity
DevSecOps	Embed security into DevOps	High-security and regulated environments	Higher upfront investment, complexity in tooling

41.10 Summary

- Project management methodology selection directly impacts OOAD practices.
- **Waterfall** favors fixed, predictable OOAD models, while **Agile** supports iterative refinement.
- **Scrum** and **Kanban** operationalize Agile principles for teams with different constraints.
- **DevOps** and **DevSecOps** enable OOAD designs to move seamlessly into automated delivery pipelines.
- Modern OOAD projects often combine methodologies: e.g., Scrum for feature work, Kanban for maintenance, and DevOps for CI/CD.

42 Software Architecture in OOAD

42.1 Learning Objectives

- Understand key concepts of **software architecture** in the context of OOAD.
- Learn common **architectural modelling frameworks** and their trade-offs.
- Explore different **architecture styles** and when to apply them.
- Understand **Clean Architecture** and its relevance to modern OOAD.

42.2 1. Fundamental Concepts of Architecture

Architecture The high-level structure of a software system, including its components, their relationships, and guiding principles.

System Architecture A broader perspective: the entire system, including hardware, software, networks, and integrations.

System Architect A role responsible for defining architecture, ensuring alignment with business needs, and maintaining scalability, security, and maintainability.

Architectural Patterns Reusable solutions to recurring architecture-level problems, e.g., MVC, microservices, layered architectures.

Software architecture bridges **OOAD** and project implementation, ensuring that object-oriented design choices align with business goals and technical constraints.

42.3 2. Architectural Modelling Approaches

Architectural modelling provides a way to visualize and communicate a system's structure. Below are common frameworks:

42.3.1 2.1 4+1 Architectural View Model (Philippe Kruchten)

- Describes a system using **five views**:
  - **Logical View:** Object models, class diagrams, main entities.
  - **Process View:** Runtime interactions, concurrency, synchronization.
  - **Development View:** Code organization (modules, packages, components).
  - **Physical View:** Deployment topology, servers, nodes, infrastructure.
  - **Scenarios (+1):** Use cases tying all views together.
- Well-suited for OOAD since it integrates use-case-driven analysis.

42.3.2 2.2 C4 Model (Simon Brown)

- A modern, lightweight visual modelling method for software architecture.
- Defines **four levels of diagrams**:
  1. **Context:** High-level system boundaries and dependencies.
  2. **Container:** Decomposes the system into apps, APIs, databases.
  3. **Component:** Breaks containers into services, modules, or layers.
  4. **Code:** Lowest level, linking diagrams to actual source structure.
- Works well with Agile and OOAD since diagrams evolve with the codebase.

42.3.3 2.3 TOGAF (The Open Group Architecture Framework)

- Enterprise-level architecture framework.
- Uses the **Architecture Development Method (ADM)** cycle:
  1. Vision → Business Architecture → Information Systems Architecture → Technology Architecture.
- Focused more on governance, less on OO-level design, but important in integrating software architecture into enterprise contexts.

42.3.4 2.4 Arc42

- A template-driven approach for documenting architecture.
- Key sections: goals, constraints, context, solution strategies, building blocks, runtime views, cross-cutting concerns.
- Encourages comprehensive but lightweight architecture documentation.

42.3.5 2.5 Architecture Inception Canvas

- Visual tool inspired by the **Business Model Canvas** to kick-start architecture planning.
- Captures at-a-glance: stakeholders, key quality attributes, assumptions, constraints, and candidate solutions.

42.4 3. Common Software Architecture Styles

42.4.1 3.1 Big Ball of Mud

- Informal, unstructured systems where architecture emerges accidentally.
- Characteristics: tangled dependencies, high technical debt.
- Common in rapidly evolving prototypes or legacy systems.

42.4.2 3.2 Unitary Architecture

- One codebase, one executable, minimal modularity.
- Suitable for small applications but lacks scalability.

42.4.3 3.3 Client-Server Architecture

- Divides applications into two main layers: **clients** (UI, requests) and **servers** (data, computation).
- Common in web applications, database-backed systems, and remote APIs.

42.4.4 3.4 N-Tier Architectures

- Extends client-server into multiple tiers, each with distinct responsibilities:
  - **Presentation Layer** (UI, controllers).
  - **Application Layer** (services, use cases).
  - **Data Layer** (databases, repositories).
- Examples: **MVC** (Model-View-Controller), **MVVM**, and layered enterprise applications.

42.4.5 3.5 Monolithic Architectures

- A single deployable unit encapsulating all layers.
- Variants:
  - **Layered Monolith:** Follows strict separation of concerns within one deployable.
  - **Pipeline Architecture:** Passes data through staged processing steps.
  - **Microkernel (Plug-in) Architecture:** A core system with pluggable modules.
- Advantages: simpler deployments, consistent performance.
- Disadvantages: scaling and independent releases become difficult.

42.4.6 3.6 Distributed Architectures

Service-Based Architecture

- Decomposes functionality into independently deployable services.
- Looser coupling than monoliths but less complex than microservices.

Event-Driven Architecture (EDA)

- Components communicate asynchronously via events.
- Ideal for high-throughput, loosely coupled systems.

Space-Based Architecture

- Focuses on in-memory data grids and parallel computation.
- Used for high-performance transaction systems like trading platforms.

Service-Oriented Architecture (SOA)

- Introduces enterprise-level services exposed over standardized interfaces (SOAP, REST).
- Often managed by orchestration layers like ESBs (Enterprise Service Buses).

Microservices Architecture

- Extreme form of distributed architecture where every business capability is its own independently deployable service.
- Characteristics:
  - Each service owns its own data.
  - APIs for communication, often REST or gRPC.
  - Encourages domain-driven design and independent scaling.
- Trade-offs: requires DevOps maturity, monitoring, and CI/CD automation.

42.5 4. Uncle Bob Martin's Clean Architecture

42.5.1 4.1 Overview

- Proposed by Robert C. Martin ("Uncle Bob") as a modern architectural paradigm.
- Goal: create architectures that are **independent of frameworks**, **UI**, **databases**, and **external agencies**.
- Core idea: **Separation of concerns** and strict control over dependency direction.

42.5.2 4.2 The Dependency Rule

- Source code dependencies always point **inward** toward higher-level policies.
- Inner layers know nothing about outer layers.

42.5.3 4.3 Layered Model

Clean Architecture organizes systems into concentric layers:

1. **Entities (Innermost Layer):**
   - Business objects and enterprise-wide rules.
   - Independent of UI, DB, or frameworks.
2. **Use Cases (Application Layer):**
   - Application-specific business logic.
   - Orchestrates entities to achieve user goals.
3. **Interface Adapters:**
   - Translate data between external systems (DB, web, UI) and inner layers.
4. **Frameworks and Drivers (Outermost Layer):**
   - Databases, web frameworks, UIs, external tools.
   - Minimal code lives here.

42.5.4 4.4 Benefits of Clean Architecture

- **Framework Independence:** You can switch technologies without rewriting core logic.
- **UI Independence:** Swap interfaces without breaking business rules.
- **Testability:** Core use cases can be tested without involving databases, frameworks, or UIs.
- **Maintainability:** Promotes modularity, loose coupling, and long-term agility.

42.6 Summary

- Architecture defines the structural and behavioral foundations of a system.
- Frameworks like **4+1**, **C4**, **TOGAF**, **Arc42**, and **Inception Canvas** help visualize and document design decisions.
- Common styles include monolithic, layered, service-oriented, event-driven, and microservices.
- **Clean Architecture** represents a modern approach to maintainable, framework-agnostic, testable systems—highly aligned with OOAD principles.

43 Databasing in the Context of OOAD

43.1 Learning Objectives

- Understand the role of databases in Object-Oriented Analysis and Design (OOAD).
- Compare **Relational** and **NoSQL** approaches.
- Learn how Object-Relational Mapping (ORM) bridges the object-relational impedance mismatch.
- Explore key challenges of integrating OOAD models with relational databases:
  - Granularity
  - Inheritance
  - Identity
  - Associations
  - Navigation

43.2 Introduction

In OOAD, the goal is to design maintainable, extensible, and well-structured systems. Almost all non-trivial systems require persistent data storage, making database integration a critical architectural concern.

Key considerations:

- How object-oriented designs map to storage schemas.
- Trade-offs between relational and non-relational databases.
- Tooling and abstractions (like ORM frameworks) that ease integration.

43.3 Relational Databases (RDBMS)

43.3.1 Overview

- Use structured tables of rows and columns to represent data.
- Based on **set theory** and **relational algebra**.
- Querying is primarily performed using **SQL** (Structured Query Language).

43.3.2 Benefits

- Strong consistency guarantees (ACID compliance).
- Mature ecosystems, stable performance, and well-known query optimization techniques.
- Widely supported across frameworks, languages, and industries.

43.3.3 Challenges in OOAD Context

Relational databases operate on tabular structures, while OOAD emphasizes rich object hierarchies, encapsulation, and relationships. Bridging this mismatch introduces complexity known as the **object-relational impedance mismatch**.

Key issues arise in:

- Mapping objects to tables.
- Preserving inheritance hierarchies.
- Managing associations between objects.
- Tracking object identity consistently.

43.4 NoSQL Databases

43.4.1 Overview

NoSQL (Not Only SQL) databases relax strict relational principles to handle scalability, flexibility, and performance challenges.

43.4.2 Types of NoSQL Databases

- **Document Stores** (e.g., MongoDB, Couchbase): Store hierarchical JSON/BSON objects closely matching OOAD models.
- **Key-Value Stores** (e.g., Redis, DynamoDB): Ideal for fast lookups and caching.
- **Column-Family Stores** (e.g., Cassandra, HBase): Optimized for large-scale analytics.
- **Graph Databases** (e.g., Neo4j): Model networks and relationships naturally.

43.4.3 Benefits for OOAD

- Flexible schemas, aligning naturally with dynamic domain models.
- Store nested objects directly (reduces joins, simplifies persistence).
- Better suited for high-volume, distributed architectures like microservices.

43.4.4 Trade-offs

- May lack strong consistency (favoring availability and partition tolerance instead).
- Query languages vary and lack universal standards.
- Limited transactional guarantees in many systems.

43.5 Object-Relational Mapping (ORM)

43.5.1 Definition

ORM frameworks map OOAD-designed domain models to relational database schemas automatically, reducing boilerplate and bridging conceptual mismatches.

43.5.2 Examples

- **Java:** Hibernate, JPA.
- **.NET:** Entity Framework.
- **Python:** SQLAlchemy, Django ORM.
- **Ruby:** ActiveRecord.

43.5.3 Benefits

- Eliminates repetitive CRUD boilerplate.
- Maps complex objects directly to database tables.
- Encourages working in an object-first paradigm consistent with OOAD.
- Improves maintainability and testability.

43.5.4 Limitations

- Overhead for complex queries—ORM-generated SQL may not always be optimal.
- High learning curve for tuning ORM performance.
- Certain OOAD features (e.g., polymorphic associations) require non-trivial mapping strategies.

43.6 Challenges of Using an RDBMS Directly in OOAD Systems

Directly coupling OO systems to relational databases can cause long-term maintenance and scalability challenges. Martin Fowler identifies five key problem areas:

43.6.1 1. Granularity Mismatch

- OOAD emphasizes rich, nested objects; RDBMS store flat tabular data.
- Example: A `Customer` object may embed multiple `Address` objects.
- Solutions:
  - Normalize to separate tables (introduces joins and complexity).
  - Store serialized data (loses queryability).
  - Consider NoSQL for nested data.

43.6.2 2. Inheritance Mismatch

- OOAD supports inheritance and polymorphism; RDBMS tables do not natively.
- Strategies:
  - **Single Table Inheritance:** All subclasses in one table (simplifies queries, wastes space).
  - **Table per Subclass:** One table per subclass (normalized but join-heavy).
  - **Table per Concrete Class:** No joins, but duplication of common fields.

43.6.3 3. Identity Mismatch

- In OOAD, objects are compared by **object identity** (reference equality).
- In RDBMS, rows are identified by **primary keys**.
- Result: Synchronizing persistent identity and in-memory identity becomes challenging.
- ORMs often handle this via identity maps and first-level caching.

43.6.4 4. Associations Mismatch

- OOAD: Associations are typically navigated in memory using object references.
- RDBMS: Associations are modeled via foreign keys and join tables.
- Example: A `Customer` has many `Orders`. In OOAD, that's a direct list; in RDBMS, it requires keys and queries.

43.6.5 5. Navigation Mismatch

- OOAD: Relationships are traversed freely (`customer.orders[0].items`).
- RDBMS: Queries are directional—data must be explicitly selected via SQL joins.
- ORMs solve this partially but can introduce performance issues (**N+1 problem**).

43.7 Summary

- Databasing is central to OOAD-driven systems but introduces the **object-relational impedance mismatch**.
- **Relational databases** provide strong guarantees but require careful mapping of OO models.
- **NoSQL databases** better align with nested, dynamic object graphs common in OOAD.
- **ORM frameworks** simplify integration but introduce abstractions that must be tuned for performance.
- Direct RDBMS use requires addressing mismatches in:
  - Granularity
  - Inheritance
  - Identity
  - Associations
  - Navigation

44 Anti-Patterns in OOAD

44.1 Learning Objectives

- Understand what **anti-patterns** are and how they differ from design patterns.
- Identify common anti-patterns in software **development**, **architecture**, and **management**.
- Recognize early warning signs and mitigation strategies.

44.2 Introduction

In software engineering, a **pattern** is a proven solution to a recurring problem in a specific context. An **anti-pattern**, by contrast, is a **commonly used approach that looks useful at first but leads to negative consequences**.

Key distinctions:

- **Design Patterns:** Best practices, reusable solutions, promote maintainability.
- **Anti-Patterns:** Tempting solutions that increase complexity, reduce flexibility, and introduce technical or organizational debt.

Understanding anti-patterns is as critical as knowing good design patterns. They help developers and managers **recognize danger signals early** and adopt better alternatives.

—

44.3 1. Development Anti-Patterns

These anti-patterns arise during software design and implementation.

44.3.1 1.1 Lava Flow

- **Definition:** Code that persists even after its purpose is obsolete because nobody understands its impact.
- **Causes:** Poor documentation, lack of ownership, unclear business rules.
- **Consequences:** Fear of refactoring, growing technical debt.
- **Solution:** Use automated tests, maintain design documentation, and perform incremental cleanups.

44.3.2 1.2 Boat Anchor

- **Definition:** Keeping unused or obsolete components "just in case" they're needed.
- **Consequences:** Increases system complexity, maintenance cost, and confusion.
- **Solution:** Remove dead code and redundant modules unless there is a concrete business case for retention.

44.3.3 1.3 Golden Hammer

- **Definition:** Applying a familiar technology, library, or design pattern to every problem regardless of suitability.
- **Example:** Using microservices for a simple CRUD app or forcing all features into a single framework.
- **Solution:** Evaluate alternatives objectively based on problem requirements, not developer preference.

44.3.4 1.4 Spaghetti Code

- **Definition:** Code with tangled control flows, poor modularity, and no clear structure.
- **Causes:** Frequent patches, lack of upfront design, ignoring encapsulation.
- **Consequences:** Hard to maintain, error-prone, resistant to refactoring.
- **Solution:** Use OOAD principles like modularity, separation of concerns, and consistent code reviews.

44.3.5 1.5 Walking Through a Minefield

- **Definition:** Working in codebases where small changes unpredictably break unrelated features.
- **Causes:** Tight coupling, poor testing, undocumented dependencies.
- **Solution:** Adopt automated testing, continuous integration, and architecture reviews.

44.3.6 1.6 Mushroom Management

- **Definition:** Developers are "kept in the dark and fed fertilizer," meaning they lack visibility into design rationale or business goals.
- **Consequences:** Frustration, low morale, misaligned implementations.
- **Solution:** Encourage transparency, document architectural decisions, and involve engineers early in planning.

—

44.4 2. Architecture Anti-Patterns

These occur when the system's high-level structure becomes inefficient or fragile.

44.4.1 2.1 Autogenerated Stovepipe

- **Definition:** Over-reliance on auto-generated boilerplate code without understanding its structure or purpose.
- **Consequences:** Inflexibility, hidden complexity, and difficult debugging.
- **Solution:** Use generation tools judiciously and refactor generated code where necessary.

44.4.2 2.2 Stovepipe Enterprise

- **Definition:** Independent systems within an organization evolve in isolation with no integration strategy.
- **Consequences:** Data silos, duplicated logic, and poor enterprise scalability.
- **Solution:** Establish enterprise architecture guidelines and integration patterns (e.g., SOA, microservices).

44.4.3 2.3 Jumble

- **Definition:** Systems where architectural responsibilities are completely mixed—UI code interacts directly with persistence, business logic leaks everywhere.
- **Solution:** Introduce layered or hexagonal architectures to enforce separation of concerns.

44.4.4 2.4 Stovepipe System

- **Definition:** A self-contained system designed without consideration for future reuse or integration.
- **Causes:** Short-term thinking, deadline pressure.
- **Solution:** Incorporate API-first design, domain-driven boundaries, and interface-based abstractions.

44.4.5 2.5 Cover Your Assets

- **Definition:** Over-architecting to mitigate perceived blame rather than solving real problems.
- **Consequences:** Unnecessary complexity, delayed timelines.
- **Solution:** Focus architecture on business outcomes and simplicity.

44.4.6 2.6 Warm Bodies

- **Definition:** Staffing projects based purely on headcount rather than relevant expertise.
- **Consequences:** Increased defects, long onboarding times, communication overhead.
- **Solution:** Match tasks to skill sets; invest in training where necessary.

—

44.5 3. Management Anti-Patterns

Management anti-patterns occur at the organizational or project coordination level.

44.5.1 3.1 Analysis Paralysis

- **Definition:** Excessive focus on documentation and planning delays implementation indefinitely.
- **Solution:** Adopt iterative approaches; aim for "just enough" design and refactor incrementally.

44.5.2 3.2 Viewgraph Engineering

- **Definition:** Producing endless slides and mockups instead of working software.
- **Consequences:** Stakeholders get a false sense of progress.
- **Solution:** Prioritize running code over documentation—align with Agile principles.

44.5.3 3.3 Fear of Success

- **Definition:** Avoiding optimizations or deployments due to fear of future scaling challenges or support needs.
- **Solution:** Prepare incremental scaling plans and focus on predictable growth rather than defensive stagnation.

44.5.4 3.4 Intellectual Violence

- **Definition:** Experts dominating discussions with jargon, belittling others, and discouraging collaboration.
- **Consequences:** Low morale, missed ideas, team fragmentation.
- **Solution:** Promote inclusive communication and mentoring practices.

44.5.5 3.5 Smoke and Mirrors

- **Definition:** Demonstrating flashy prototypes or partial implementations to create the illusion of progress.
- **Risks:** Stakeholders overestimate project readiness, resulting in mismatched expectations.
- **Solution:** Communicate project maturity honestly and manage expectations proactively.

44.5.6 3.6 Project Mismanagement

- **Definition:** Lack of coordination, unclear requirements, shifting priorities, or inadequate resource allocation.
- **Consequences:** Missed deadlines, budget overruns, low-quality deliverables.
- **Solution:** Use Agile or hybrid methodologies, prioritize transparency, and clarify project ownership.

—

44.6 Summary

- **Anti-patterns** represent common mistakes that initially seem beneficial but lead to poor software quality, low productivity, and fragile systems.
- Development anti-patterns compromise code maintainability.
- Architecture anti-patterns create structural rigidity and integration challenges.
- Management anti-patterns drive project delays and reduce team effectiveness.
- Recognizing these patterns early enables teams to steer toward sustainable, OOAD-aligned solutions.

45 Assorted Practice & Architecture Questions & Answers

1. "When looking into a design effort, it's important to not confuse supplemental domain information for the problem statement." This statement highlights:

   - How design is limited to specification and implementation details
   - That analysis and design are the same activity
   - ✓ The need to do analysis to determine clearly what problem you'll solve in order to develop a design
   - How design is used to develop the problem statement

2. The SOLID principles are:

   - The complete basis for object-oriented designs
   - Rarely mentioned as support for object-oriented design
   - ✓ Often cited as the core principles for OO, even though there are other associated important principles
   - Obsolete

3. The DRY principle… (select all correct):

   - It applies to code but not requirements
   - ✓ Supports the rule of one requirement in one place
   - ✓ Stands for Don't Repeat Yourself
   - ✓ Helps avoid errors from changes to one copy of an item that are missed for another

4. Which of the following was **NOT** cited as a cost of violating YAGNI, i.e. writing code before it is needed:

   - Cost of carry
   - ✓ Cost of usability
   - Cost of delay
   - Cost of build/refactoring time

5. The discussion of Alexander's term *complexification* pointed out that… (select all correct):

   - Complexification is limited to a bottom-up design perspective
   - ✓ Patterns applied to a design are operators that differentiate the design space
   - ✓ Each part of a design is given its specific form by its existence in a larger whole
   - ✓ Design decisions add more complexity as designs are progressively elaborated on

6. In reviewing other conceptual design approaches, which of the following are true? (select all correct):

   - ✓ Robustness diagrams show boundaries, controllers, and entities
   - In the Analysis Matrix approach, variations are row headers, concepts are matrix entries, and related variations are joined in rows
   - ✓ Logical Data Models, Entity-Relationship Models, and Dimensional Data Models are alternative approaches for modeling the design of databases
   - ✓ Class Responsibility Collaborator Modeling (aka CRC Cards) provides a good design approach to use with non-technical stakeholders

Final Knowledge Test — Architecture, OO, and Practices

Question 1: When sending messages across byte-stream based networks over an API or storing object information in non-object-oriented databases, what process is used to convert between object data structures and flat (or nested) data representations?

(a) Serialization and deserialization

(b) Publish and subscribe

(c) Aggregation and resequencing

(d) Request and reply

Question 2: Which of the following is **NOT** true regarding the concept of "complexification"?

(a) An alternative way to consider this is that complexification combines the bottom-up detail view of a design with the overall system view from the top-down

(b) Alexander defined this as a part being given form based on its existence in the content of the larger whole

(c) Each decision made in the design of a software system is likely adding complexity in the process of progressive elaboration

(d) In complexification, design of parts come first, and result in the final whole design

Question 3: Which of the following is **NOT** true regarding ORMs (Object Relational Mappers)?

(a) Even using an ORM, you must still understand the underlying database design, whether it is SQL or non-SQL based

(b) ORMs solve leaky abstraction by using metadata to define data representations and connections

(c) ORMs provide a way to handle data persistence transactions using standard object, method, and attribute representations

(d) There is significant difference in data representation between objects and database records, ORMs provide a mapping between the two

Question 4: Which of the following statements are true about refactoring **(select all correct)**?

(a) ☐ Follow the camping rule in refactoring, which is: make sure all fires are out before starting the next one

(b) ☐ The introduction of Fowler's *Refactoring* says the true test of good code is how easy it is to change the code

(c) ☐ Before adding features in code that isn't structured for the changes, refactor the code to allow the changes, and then add the feature

(d) ☐ Performing refactoring in small steps reduces the likelihood of introducing errors, but you should still have test cases to run to verify changes have not broken existing code

Question 5: Which of the following are true about dependency injection (or DI) **(select all correct)**?

(a) ☐ DI in Java can be performed with setter injection methods or injector methods from interfaces, but not via object constructor injection

(b) ☐ DI is closely related to Inversion of Control, which represents control of part of a system being transferred out of that system into a container or other framework

(c) ☐ The DI framework shown for Java and Spring has a separate assembler module that injects implementations into the code requesting the object reference

(d) ☐ The Java and Spring DI examples showed annotations or XML files used to define mapping of objects that could be injected

Question 6: Considering reflective OO languages, which of these statements are true **(select all correct)**?

(a) ☐ In implementing reflection in the language, Java provides full intercession capabilities, but has very limited introspection functionality

(b) ☐ Structural reflection looks at program data or code, behavioral reflection looks at the runtime environment

(c) ☐ To perform intercession operations, a program must provide encoded execution data using introspection

(d) ☐ Reflection is the ability of a language to analyze or query itself (via introspection) or modify itself (via intercession) at runtime

Question 7: In discussing architecture, we looked a bit at Microservices. Which of the following are true statements regarding Microservices? **(select all correct)**

(a) ☐ Microservices are usually independently deployable, more often than not, a microservice is designed to be replaced vs. maintained in place

(b) ☐ APIs cannot be used as microservice messaging interfaces

(c) ☐ Microservices are an alternative to a monolithic architecture, where multiple services are handled by a single code structure

(d) ☐ Microservices are goal oriented – balancing speed, safety, and the ability to scale applications

Question 8: The SOLID principles include key OO principles such as Favoring Delegation Over Inheritance and Encapsulating What Varies.

(a) True

(b) False

Question 9: The Single Responsibility Principle supports the ideas that classes should have one reason to change and they should be strongly cohesive.

(a) True

(b) False

Question 10: "There's no silver bullet'' refers to the idea that no single software technique or tool will yield an order of magnitude change in productivity, reliability, or simplicity of software.

(a) True

(b) False

Question 11: The steps in the "Thinking in Patterns'' design approach are to identify patterns, analyze and apply patterns (which may repeat or iterate), and then add detail.

(a) True

(b) False

Question 12: The only feasible alternative to in-line SQL statements using JDBC in OO Java code is use of an ORM like Hibernate.

(a) True

(b) False

Question 13: The "+1" in the 4+1 Architecture approach are class diagrams that provide the architecture's structure.

(a) True

(b) False

Question 14: An API provides for the principle of "code to an interface" by separating out the design of interaction methods from the server's or client's implementations or use of those interactions.

(a) True

(b) False

Question 15: UX/UI frameworks are rarely object-oriented as there is no natural hierarchical relationship between UI elements (for example, in the atomic design scheme).

(a) True

(b) False

Question 16: AI tools supported by LLMs are improving software development capabilities; improvements reviewed included coding autocompletion, improved technical searches, and chat-driven program generation.

(a) True

(b) False

Answer Key

Q1: (a)

Q2: (d) – parts-first is *not* the idea; the whole informs the parts

Q3: (b)

Q4: (b), (c), (d)

Q5: (b), (c), (d)

Q6: (b), (d)

Q7: (a), (c), (d)

Q8: False

Q9: True

Q10: True

Q11: True

Q12: False

Q13: False

Q14: True

Q15: False

Q16: True