ULTIMATE

Java Design Patterns

Master Creational, Structural, Behavioral, Concurrency, and Architectural Patterns for Scalable Enterprise Java Applications

Ricken Bazolo

First Published: May 2026

Published by: Orange Education Pvt Ltd, AVA®

Address: 9, Daryaganj, Delhi, 110002, India

275 New North Road Islington Suite 1314 London,

N1 7AA, United Kingdom

ISBN (PBK): 978-93-49887-90-9

ISBN (E-BOOK): 978-93-49887-70-1

Table of Contents

CHAPTER 1

Introduction to Java Design Patterns

Introduction

Java programming is central to modern software development due to its robustness and flexibility across various contexts. However, writing efficient, readable, and maintainable applications requires knowing the fundamental concepts that guide good development practices. Design patterns are one of these essential concepts, providing proven approaches to solving common software challenges cleanly and effectively.

In this chapter, we will explore the foundational principles of design patterns in Java, their purpose, historical roots, classification, and their role in simplifying complex development scenarios. We will further examine how these patterns facilitate the resolution of real-world problems by offering proven, reusable solutions.

Structure

This chapter is organized into the following sections:

- Understanding Design Patterns
    - Defining a Design Pattern
    - Why Design Patterns Matter
- History and Evolution of Design Patterns
- Categories and Classifications
- Benefits and Best Practices
- Use Cases and Examples

Understanding Design Patterns

Let us imagine that we decide to build a house. We call upon an architect to avoid fragile walls or poorly arranged rooms. The architect then offers us a plan to follow, which helps us select the right materials according to the size of the land, the preferences of the inhabitants, and other essential details. This plan is not a rigid rule, but rather a guide that can be adapted. A mansion and a cabin follow certain shared principles to ensure that they are sturdy, yet they are built in very different ways.

In the same way, Design Patterns are not fixed rules. They are standard solutions developers can use and modify according to the needs of their project, the size of the team, or the expectations of the user. Just as an architect adapts plans to each house, a developer selects and adjusts Patterns to solve specific application problems.

Defining a Design Pattern

A Design Pattern is a proven approach to solving common software design problems. Just as an architect relies on universal principles to design a staircase or a roof structure, a developer applies Design Patterns to organize code optimally.

Let us imagine we need to organize a kitchen. Without prior thought, we might place the knives far from the cutting board, store the spices in hard-to-reach cupboards, and position the stovetop near an open window.

The result: An impractical and potentially hazardous kitchen.

An experienced cook would instead apply simple yet effective principles of organization:

- **Problem**: How can we make the kitchen both functional and safe?
- **Solution**: Group tools by area of use (cutting, cooking, storage), keep the most frequently used items within easy reach, and isolate heat sources.

Design Patterns follow the same logic. They are not rigid rules but adaptable principles, like a chef adjusting the layout of the kitchen according to the menu of the day.

They transform code into a structured space where every line has its place, every interaction is predictable, and every change can be made without dismantling the whole system.

Why Design Patterns Matter

In a technological landscape where applications must constantly evolve, Design Patterns serve as a strategic compass. Imagine software as a growing city without any urban planning; adding a new district (a feature) could risk blocking existing roads (previous functionalities). Design Patterns act like urban planning regulations, allowing one part of the system to be modified without compromising the whole. For example, the Strategy Pattern allows an algorithm (such as a search engine) to be swapped out like a module in an electric vehicle without altering the rest of the car.

Beyond flexibility, these patterns transform human collaboration. Consider a team working on an online booking system. If a developer mentions using the ***Factory Pattern*** to create booking objects, their colleagues immediately understand the underlying structure. This shared language eliminates hours of technical explanations, accelerates code reviews, and reduces misunderstandings. It is as if everyone speaks the same dialect within an international project.

Finally, Design Patterns foster a mindset of active learning. Each pattern mastered becomes a new key to unlocking complex problems. Take the example of a developer who discovers the ***Observer Pattern***; by applying it to a notification system, they suddenly realize that the same model could simplify user interface update management. This revelation creates a virtuous cycle: the more patterns we master, the more use cases we identify, and the sharper our technical creativity becomes.

And what if we ignored these models? We would end up with fragile software, where each update becomes a puzzle, and teams spend more time debugging than innovating. Design Patterns are not a theoretical luxury but a pragmatic investment in sustainability and efficiency.

History and Evolution of Design Patterns

The concept of design patterns did not originate in computer science, but rather in architecture. In the 1970s, architect Christopher Alexander introduced, in his seminal work A Pattern Language, a method for designing urban and domestic spaces based on recurring patterns of quality. Each pattern represents a proven solution to a common design problem, contextualized within a specific environment. The goal was to document these solutions in order to transmit them and to build a shared language among designers. This approach profoundly influenced other disciplines, notably software engineering, which, a decade later, would draw inspiration from it to address its own challenges.

By the late 1980s, the rise of object-oriented programming had revealed practical limitations for developers. Software architectures became rigid and inflexible, code redundancy increased, and modularity struggled to take hold. In response to these issues, four experts Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides published in 1994 Design Patterns: Elements of Reusable Object-Oriented Software, a foundational work. This now classic guide formalized twenty-three design patterns aimed at optimizing the construction, organization, and maintenance of software systems. These patterns are grouped into three major categories:

- **Creational**: Govern object creation mechanisms;
- **Structural**: Define the hierarchical arrangement of components;
- **Behavioral**: Orchestrate the dynamics of interaction between entities.

Each pattern is described using a methodical framework, a problematic situation, its context of application, a generic solution, and its technical implications. This rigorous formalization not only enables the accumulation of collective experience, but also establishes a common language among professionals, thereby accelerating the resolution of recurring challenges.

In the years that followed, design patterns would no longer be limited to a mere catalog of solutions. Communities began to form to discuss, enrich, and adapt them to diverse contexts. The PLoP (Pattern Languages of Programming) conference, launched in 1994, established a rigorous peer review process. At the same time, the *Portland Pattern Repository* became one of the first technical wikis, encouraging the exchange of concrete use cases.

This movement contributed to the emergence of specialized patterns, particularly in areas such as concurrency and software testing. The very notion of a pattern thus evolved into a tool for structuring thought, one that extends well beyond object-oriented programming.

Toward Large-Scale Architectures

With the rise of web and distributed applications, design patterns are no longer confined to solutions at the class or component level. The *Model-View-Controller* (MVC) pattern, though a veteran, exemplifies this conceptual elevation by offering a clear separation between business logic, presentation, and data management.

As early as 1996, the book Pattern-Oriented Software Architecture formalized this extension by introducing patterns applicable to entire system architectures, such as:

- **Layers**: A layered structure for organizing responsibilities (user interface, business logic, data access, and so on).
- **Broker**: An intermediary that facilitates communication in distributed systems.

In 2002, Martin Fowler expanded upon this approach with *Patterns of Enterprise Application Architecture*, where he described solutions tailored to complex enterprise architectures, such as the Unit of Work or Repository patterns, designed to optimize the organization of data access and transaction management.

The Era of Microservices and Cloud Computing

The emergence of microservices architectures in the 2010s, combined with the rise of cloud computing, introduced a new set of challenges, resilience, scalability, and inter service communication. In response to this context, patterns tailored to such environments began to emerge or gain wider adoption. Among the most well-known are:

- **Circuit Breaker**: This acts as an advanced protection mechanism within your architecture. In the event of a service failure, it steps in to isolate the issue, thereby preventing a domino effect that could potentially cripple the entire system. It is a proactive approach that ensures resilience and continuity of operations.
- **Saga**: This is a distributed transaction management strategy that favors the coordination of compensating actions over the use of global transactions. It maintains data consistency by orchestrating a sequence of reversible operations, thereby offering greater flexibility in managing partial failures.

- **Command Query Responsibility Segregation (CQRS):** This introduces a clear separation between read and write operations within a system. By isolating these responsibilities, each path can be optimized according to its specific requirements, thereby enhancing both performance and scalability. It is a modular approach that encourages a more robust and adaptable architecture.
- **Event Sourcing:** This relies on recording every change as a distinct event, thereby creating a comprehensive history of modifications. This method facilitates traceability and enables the current state of data to be reconstructed from past events, offering enhanced visibility into the evolution of the system.

These patterns, often promoted by cloud providers or open-source communities, serve to enrich the toolbox of the modern architect.

A Thought in Constant Evolution

From streets and buildings to modern distributed systems, design patterns embody a method for structuring complexity. They offer a shared vocabulary, facilitate technical decision-making, and allow us to build upon decades of collective experience. Some critics argue that patterns sometimes compensate for inherent limitations in programming languages, yet their relevance remains unshaken; they help model recurring problems with clarity. More than a mere catalog, design patterns express a way of thinking about software engineering, a bridge between theory and practice, between reusability and adaptation. For the developer, they represent far more than technical solutions: they are a lever for designing robust, coherent, and scalable systems.

Categories and Classifications

When one begins studying design patterns, it is common to feel overwhelmed by their variety. Therefore, it is necessary to understand how these patterns are classified into categories. This organization not only facilitates memorization but also helps one better grasp their role within a software architecture. We shall explore the main families of patterns and the modern extensions that have emerged from the evolution of software technologies.

Foundations, the Classic Classification

This classification distinguishes three main families:

Creational Patterns

These patterns address a simple yet fundamental concern: how to create objects with flexibility, efficiency, and control. The objective is to decouple client code from direct object instantiation and introduce a level of abstraction that facilitates maintainability and extensibility.

- Singleton
- Factory method
- Abstract Factory
- Builder
- Prototype

Example:

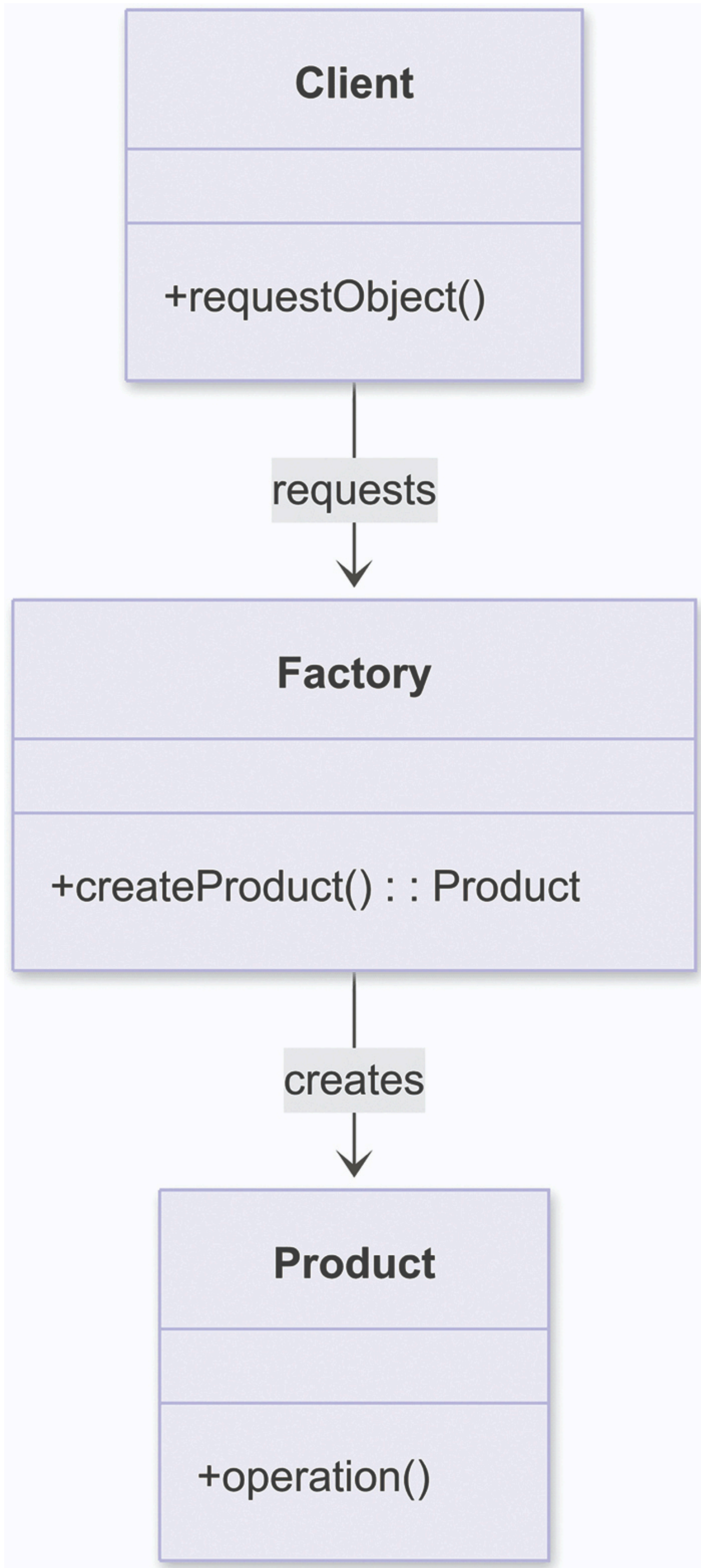
Client
+requestObject()
requests
Factory
+createProduct() : : Product
creates
Product
+operation()

Figure 1.1: Factory - Creational Pattern

The following diagram clearly illustrates the operation of a creational design pattern: The Client requests the **Factory** by invoking the `createProduct()` method, which generates an instance of the **Product**. This object executes the intended operation through the `operation()` method.

Structural Patterns

This category of patterns addresses how classes and objects are organized to form robust and extensible software architectures. It promotes object composition while reducing tight coupling, thereby enhancing flexibility and maintainability.

- Adapter
- Composite
- Decorator
- Facade
- Proxy
- Bridge
- Flyweight

Example:

Figure 1.2: *Flyweight - Structural Pattern*

The diagram above illustrates a structural pattern through the example of character management. The `Flyweight` defines a standard interface for objects that can be shared, such as the letters "A" or "B." The `ConcreteFlyweight` stores the intrinsic state and the shared information, such as the shape of the letter. The `UnsharedConcreteFlyweight`, on the other hand, represents objects that are not shared, such as a word with specific formatting. The `FlyweightFactory` centralizes the creation and management of shared instances to avoid duplication. Finally, the **Client** completes each object by providing the extrinsic state, such as position or color at the use time.

Behavioral Patterns

Behavioral patterns focus on the interactions between objects and the distribution of responsibility. They optimize communication, enable dynamic behaviors, and enhance flexibility.

- Observer
- Strategy
- Command
- Template Method
- Iterator
- State
- Chain of Responsibility
- Interpreter
- Mediator
- Memento
- Visitor

Example:

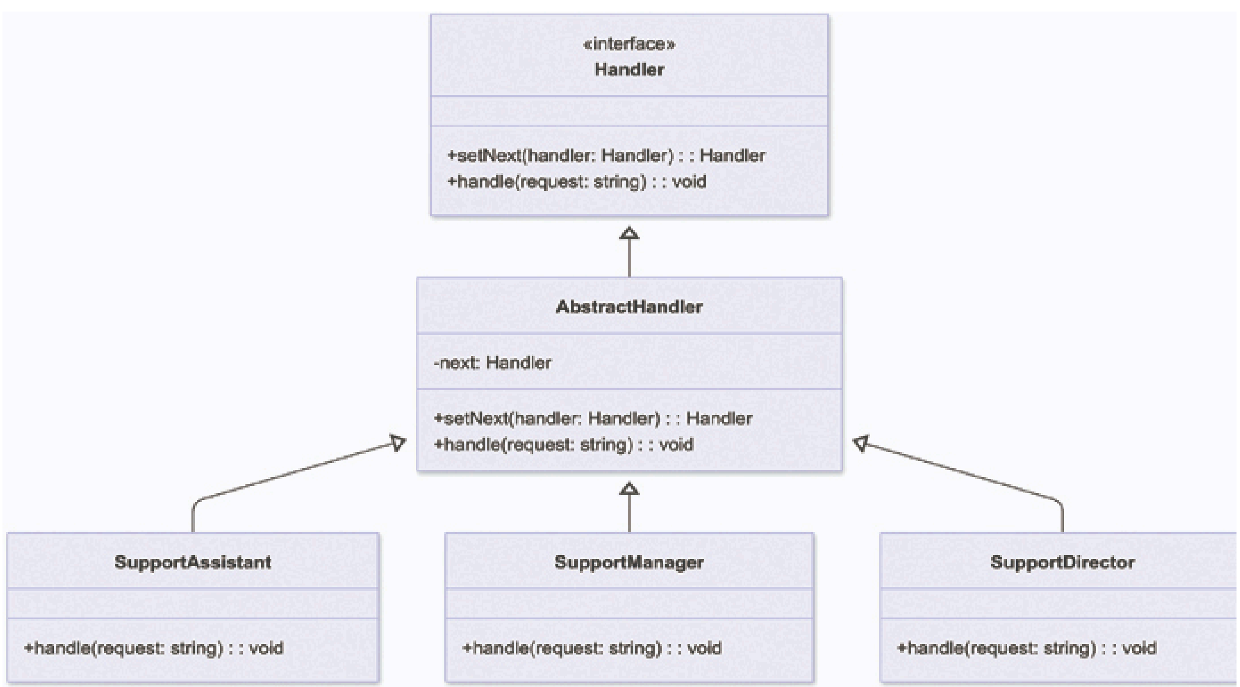

***Figure 1.3:** Chain of Responsibility - Behavioral Pattern*

The diagram above illustrates a behavioral design pattern. The role of the Handler is to provide a standardized interface for each element in the chain, while the **`AbstractHandler`** manages the forwarding logic through the **`setNext`** method. The concrete classes (**`SupportAssistant`**, **`SupportManager`**, and **`SupportDirector`**) can either handle the request directly or pass it on to the next link in the chain. Furthermore, the chain can be configured dynamically for instance, the assistant can be linked to the manager, who may, in turn, forward the request to the director if necessary.

Specialized Patterns for Distributed and Cloud Architectures

As software systems have evolved with the advent of the web, distributed applications, microservices, and cloud computing, new needs have emerged. This evolution has led to the development of complementary categories of design patterns.

Concurrency Patterns

These patterns make it possible to better leverage the multithreading capabilities of a system while ensuring consistency in a concurrent execution environment.

Examples:

- Thread Pool
- Producer-Consumer
- Read-Write Lock
- Future
- Fork/Join Framework

Example:

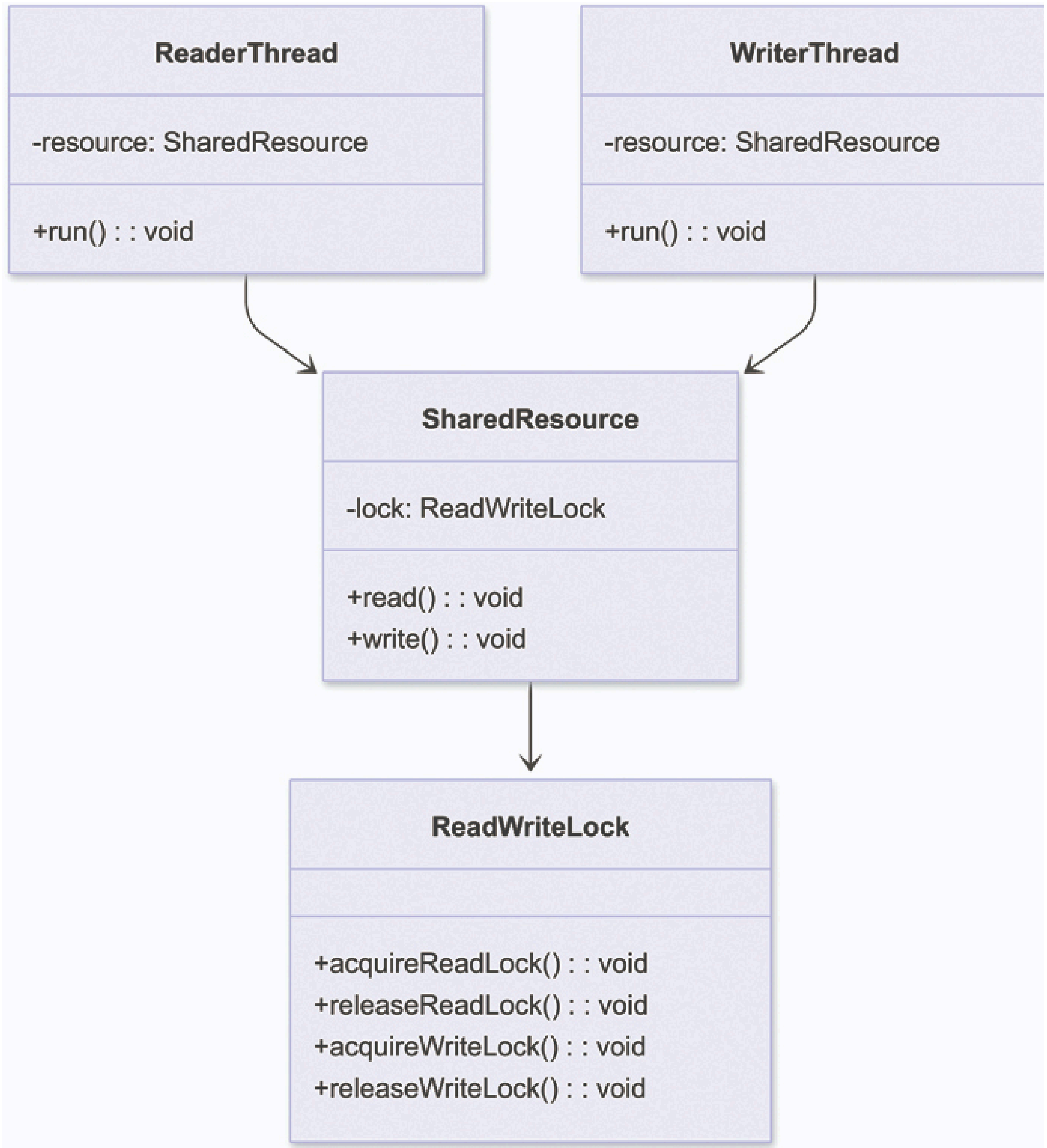

Figure 1.4: *Read-Write Lock - Concurrency Pattern*

The diagram above illustrates a concurrency pattern for managing simultaneous access to a shared resource. The role of **`ReadWriteLock`** is to control who can read from or write to this resource to prevent conflicts. **`SharedResource`** represents the shared data, accessible for both reading and writing. The **`ReaderThread`** and **`WriterThread`** threads represent read and write operations, respectively, while adhering to the rules enforced by the lock; multiple reads can occur in parallel, but a write operation requires exclusive access.

Architectural Patterns

They structure application design at a broader scale than classes and objects by organizing the system into modules, layers, or services.

Examples:

- Model View Controller (MVC)
- Microservices Architecture
- Layered Architecture
- Event-Driven Architecture
- Service-Oriented Architecture
- Repository

Example:

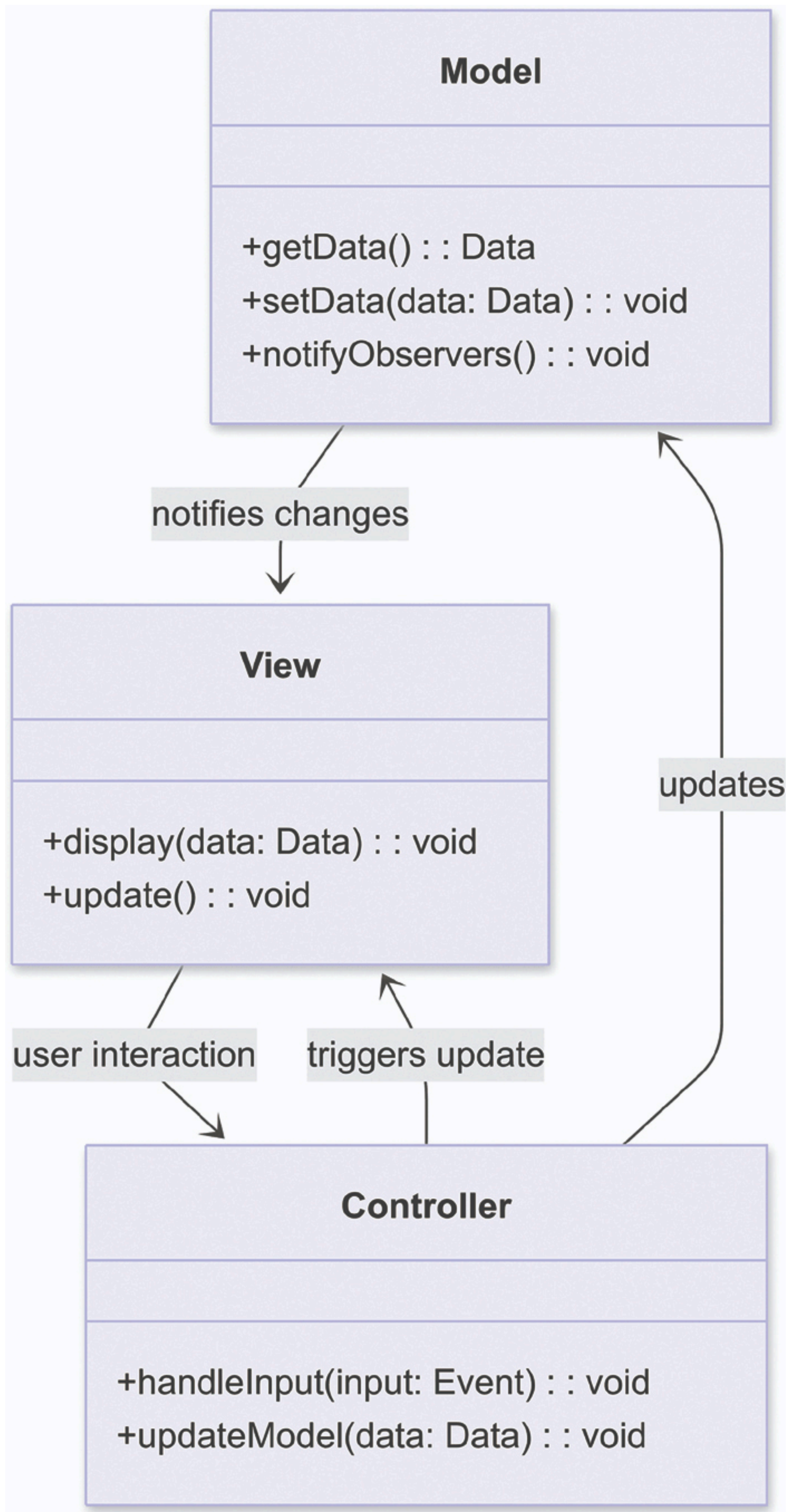
Model
+getData() : : Data
+setData(data: Data) : : void
+notifyObservers() : : void
notifies changes
View
+display(data: Data) : : void
+update() : : void
updates
user interaction
triggers update
Controller
+handleInput(input: Event) : : void
+updateModel(data: Data) : : void

Figure 1.5: *MVC - Architectural Pattern*

The diagram above illustrates an architectural pattern. The **Model** contains both the data and the business logic of the application. When a change occurs, it automatically notifies the **View**. The View is responsible for displaying this data to the user and may call upon the **Controller** to relay user actions. The Controller, in turn, plays a central role: it interprets user interactions, updates the Model if necessary, and then instructs the View to refresh itself to reflect the updated information.

Relying on architectural models that ensure resilience to failures, isolate faults effectively, and support eventual consistency across components is essential in distributed and event-driven systems.

Distributed Systems Patterns

These patterns are designed for applications distributed across multiple nodes or servers. They ensure efficient communication and decentralized role management.

Examples:

- Client-Server
- Broker
- Peer-to-Peer

Example:

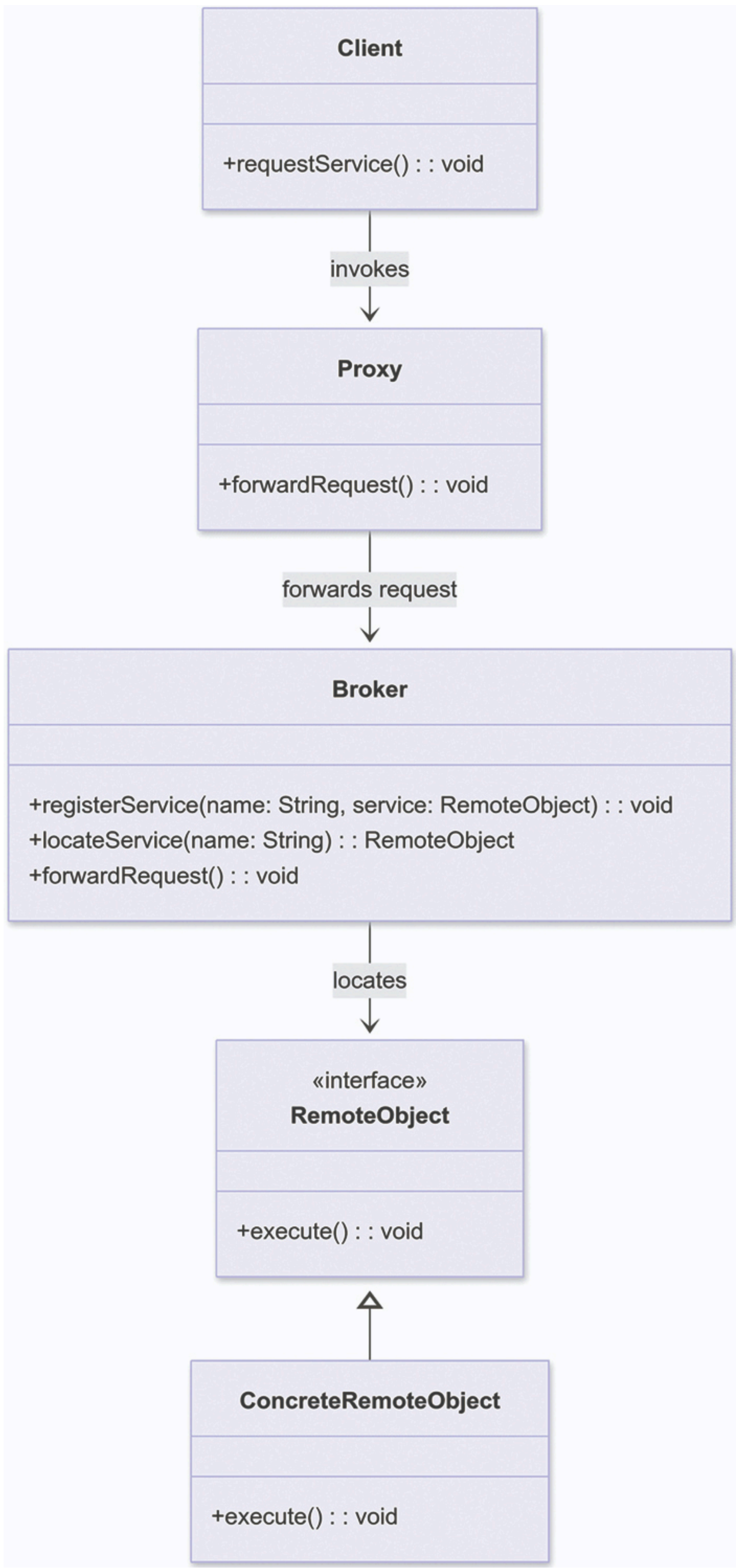
Client
+requestService() : : void
invokes
Proxy
+forwardRequest() : : void
forwards request
Broker
+registerService(name: String, service: RemoteObject) : : void
+locateService(name: String) : : RemoteObject
+forwardRequest() : : void
locates
«interface»
RemoteObject
+execute() : : void
ConcreteRemoteObject
+execute() : : void

Figure 1.6: *Broker - Distributed Systems Pattern*

The diagram above illustrates the operation of a pattern specific to distributed systems. The **Client** does not interact directly with the remote object; it communicates through a **Proxy**, which acts as a local representative. This Proxy forwards the request to the **Broker**, who is responsible for locating the appropriate remote service and transmitting the request. The remote interaction relies on the `RemoteObject` interface, which defines the available operations. On the server side, the `ConcreteRemoteObject` class provides the actual implementation of the service and executes the requested operation.

Benefits and Best Practices

Design Patterns provide a common language that facilitates understanding and communication among developers. The following are overviews of their benefits and best practices for using them effectively.

Benefits

- **Improved Maintainability:** Using standardized and well-recognized structures makes code easier to understand, modify, and evolve.
- **Reusability of Solutions:** Design patterns are generic templates that can be applied to various projects. They help avoid "reinventing the wheel" and save time and effort.
- **Clear Communication:** Pattern names and concepts (Factory, Singleton, Observer, Strategy, and so forth) form a common language among developers. A simple pattern name can easily convey the intent and role of a code segment.
- **Better Code Organization and Decoupling:** Most design patterns are designed to separate responsibilities or reduce tight coupling, resulting in more modular and scalable code.
- **Enhanced Testability:** Code structured according to design patterns is generally easier to test, as patterns often encourage behavior isolation and the definition of clear interfaces.

Best Practices for Using Design Patterns

- **Do not overuse Design Patterns:** The goal is not to force a pattern into every line of code. Excessive use can make the code unnecessarily complex. It is better to choose a pattern only when it genuinely addresses an identified problem.
- **Understand the problem before choosing a pattern:** The best results come from starting with a concrete need or difficulty. A pattern should solve a specific issue rather than simply be used because it is *in fashion*.
- **Favor simplicity:** Even if a design pattern seems appealing, choosing a more straightforward solution is sometimes wiser when the problem is simple. Code readability should remain a top priority.
- **Master the core concepts of each pattern:** Design patterns are based on SOLID principles (Single Responsibility, Open/Closed, and the like). Understanding why and how each pattern works makes implementation easier and helps avoid common mistakes.
- **Combine patterns with discernment:** Multiple patterns can sometimes complement each other (for example, combining Observer and Factory). However, one should avoid adding layers of complexity without a valid reason.
- **Document and share:** When implementing a design pattern, clearly indicate in the documentation why it was chosen, what problem it solves, and its limitations. This transparency facilitates maintenance and knowledge sharing within the team.

Use Cases and Examples

We will now examine two concrete cases in which several classic design patterns are effectively combined to address common challenges in enterprise applications. These practical examples demonstrate how to structure code to maximize flexibility, testability, and maintainability.

Case 1: Payment Processing System with Factory Method and Strategy

Context: An e-commerce platform commonly manages various payment methods (credit card, PayPal, cryptocurrency, Mobile Money, and the list continues). Each method requires specific rules for execution and validation.

Problematic: How can we efficiently manage the instantiation of different payment processors while making the amount validation logic dynamic?

Proposed Solution:

- **Creating Payment Processors with the Factory Method Pattern**

  Common Interface: A sealed interface is defined to allow only specific classes to implement it. This ensures that only predefined implementations (in this case, for credit card or PayPal) will be used.

  ```
  public sealed interface PaymentProcessor permits
  CreditCardProcessor, PayPalProcessor {
   String process(double amount);
  }
  ```

  Concrete Implementations: Each class that implements this interface provides its version of the *process method* tailored to the corresponding payment method.

  - **PayPal Processor**

    ```
    final class PayPalProcessor implements PaymentProcessor
    {
     @Override
     public String process(double amount) {
      return "PayPal payment: %s".formatted(amount);
     }
    }
    ```

  - **Credit Card Processor**

    ```
    final class CreditCardProcessor implements
    PaymentProcessor {
     @Override
     public String process(double amount) {
      return "Credit card payment: %s".formatted(amount);
     }
    }
    ```

  This design approach relies on two key principles: **separation of concerns**, where each payment processor contains autonomous logic dedicated to its specific function, ensuring modularity and simplified maintenance and **extensibility**, which allows new payment methods to be integrated easily by simply creating additional implementations of the `PaymentProcessor` interface, without affecting existing systems. This structure promotes code clarity by avoiding complex interdependencies and supports flexible system evolution to meet future needs.

- **Implementing Dynamic Validation with the Strategy Pattern**

  Defining the Validation Strategy: A functional interface represents a payment validation strategy, allowing validation rules to be defined interchangeably and flexibly.

```
@FunctionalInterface
public interface PaymentValidationStrategy {
 booleanvalidate(double amount);
}
```

  `PaymentEngine` Class: The `PaymentEngine` class receives a validation strategy through its constructor. It then uses this strategy to verify whether the amount is valid before delegating the processing to the selected payment processor.

```
public class PaymentEngine {
 private final PaymentValidationStrategyvalidationStrategy;
 public PaymentEngine(PaymentValidationStrategy strategy) {
this.validationStrategy = strategy;
 }
 public String executePayment(PaymentProcessor processor,
 double amount) {
  return validationStrategy.validate(amount) ?
processor.process(amount) :
    "Validation Failure";
 }
}
```

The modular design isolates the validation logic within a dedicated interface, providing the flexibility to modify validation rules without impacting the core processing logic. This separation of concerns also enhances the system's reusability: a single instance of `PaymentEngine` can be adapted to different scenarios by associating it with various payment processors, depending on the strategy or specific requirements, without requiring any structural overhaul. This approach combines simplified maintenance with operational versatility.

Usage Example: We demonstrate how to assemble the system by defining a control strategy *in this case, a simple anti-fraud check* and executing payments using different processors.

```
public class PaymentProcessingSystem {
  public static void main(String[] args) {
    var fraudCheckStrategy = (PaymentValidationStrategy) amount -
    > amount < 10_000;
    var engine = new PaymentEngine(fraudCheckStrategy);
    System.out.println(engine.executePayment(new
    CreditCardProcessor(), 500));
    System.out.println(engine.executePayment(new
    PayPalProcessor(), 15_000));
  }
}
```

Console Output

```
Credit card payment: 500.0
Validation Failure
```

The Factory Method isolates the processors' instantiation logic, while the Strategy pattern allows business rules to be modified dynamically without affecting the main processing flow. This highlights the system's flexibility and adaptability to different validation policies.

Case 2: Notification System with the Observer and Decorator Patterns

Context: A banking application is looking to develop an adaptable notification system that supports *email*, *SMS*, and in-app *notifications*. This system should also have the capability to integrate advanced features like encryption and logging.

Problematic: How can we structure a modular system where observers react to events and cross-cutting features can be easily added without complicating the core business logic?

Proposed Solution:

- **Implementation of the Observer Pattern**

  Defining a Listener for Notifications: An interface is defined for notification listeners so that each listener class implements an update method.

```
public interface NotificationListener {
  void update(String recipient, String message);
}
```

  Notification Service: This class manages all subscribers and dispatches notifications to each.

```
public class NotificationService {
  private final List<NotificationListener> listeners = new
  ArrayList<>();
  public void subscribe(NotificationListener listener) {
    listeners.add(listener);
  }
  public void notifyAll(String recipient, String message) {
    listeners.forEach(listener ->listener.update(recipient,
    message));
  }
}
```

- **Enhancing Notifications with the Decorator Pattern**

  Notification Interface: A sealed interface that enforces the use of specific components for notification.

```
public sealed interface Notification permits
BaseNotification, LoggedNotification {
  void send();
}
```

  Basic Implementation: The basic notification contains the content and provides a simple sending mechanism.

```
public record BaseNotification(String content) implements
Notification {
  @Override
  public void send() {
    System.out.printf("Send: %s%n", content);
  }
```

```
}
```

Decorator for Logging: The decorator adds additional functionality (in this case, logging) while preserving the sending logic of the basic notification.

```
public final class LoggedNotification implements
Notification {
 private final Notification wrapped;
 public LoggedNotification(Notification n) {
  this.wrapped = n;
 }
 @Override
 public void send() {
  System.out.println("[LOG] Sending started");
  wrapped.send();
 }
}
```

The **`LoggedNotification`** class ensures the separation of concerns by focusing exclusively on logging without altering the sending logic of **`BaseNotification`**. At the same time, feature stacking allows decorators (for example, encryption) to be added dynamically as needed without affecting existing code.

- **Concrete Execution (Main Class with a Listener)**

Implementation of a Specific Listener for Emails:

```
public class EmailListener implements NotificationListener
{
 @Override
 public void update(String recipient, String message) {
  new LoggedNotification(new BaseNotification("Email to
  %s: %s".formatted(recipient, message))).send();
 }
}
```

Final Assembly: The service is instantiated, the listener is added, and a notification is triggered.

```
public class NotificationSystem {
 public static void main(String[] args) {
  var service = new NotificationService();
```

```
    service.subscribe(new EmailListener());
    service.notifyAll("client@bank.com", "Transfer
    approved");
  }
}
```

Console Output

```
[LOG] Sending started
Send: Email to client@bank.com: Transfer approved
```

The combination of patterns allows the listener to use the decorator to enrich the notification, while the Observer service handles distribution without knowing the details of the enhancement. The modular design ensures that each component, the decorator, the listener, and the service, can be tested or extended independently.

Conclusion

In this introductory chapter, we have discussed design patterns, their historical context, and their role in software development. We have also explored their classifications, benefits, and best practices, complemented by practical examples.

The next chapter focuses on creation patterns, which aim to delegate instantiation logic to promote flexibility, decoupling, and code reuse. Here, we detail Java implementations of the *Singleton*, *Factory Method*, *Abstract Factory*, *Builder*, and *Prototype patterns*.

Points to Remember

- Patterns provide structure and clarity in software design.
- Choosing the correct pattern requires analyzing the specific problem.
- Design patterns improve code reusability, scalability, and maintainability.

Multiple Choice Questions

1. Amongst which of these options do Design Patterns provide?

a. Ready-to-use code snippets
b. Template solutions to common problems
c. Language-specific functionalities
d. Debugging techniques

2. The Singleton Pattern falls under?
   a. Creational Patterns
   b. Structural Patterns
   c. Behavioral Patterns
   d. Architectural Patterns
3. Who introduced design patterns to software engineering?
   a. Alan Turing
   b. James Gosling
   c. Gang of Four (GoF)
   d. Dennis Ritchie

Answers

1. b
2. a
3. c

Questions

1. How does the Observer pattern enhance event handling?
2. Compare Singleton and Factory patterns. When would you use each?
3. How does the Adapter Pattern promote code reusability in legacy systems?

Key Terms

- **Design Pattern:** Reusable solutions for common software problems.

- **Gang of Four (GoF):** Authors of the foundational book on design patterns.
- **Creational Patterns:** Deal with object creation mechanisms.
- **Structural Patterns:** Manage the composition of classes or objects.
- **Behavioral Patterns:** Define communication between objects.