Next Gen Java : Tools , Patterns and Integration

Dr.P.Pradeep Kumar /Mr.V.M.Kavin Chakaravarthi /

Dr.M.Rajeshkumar /Dr.N.Kalaichelvi/ Dr.V.Deepa

SCIENTIFIC RESEARCH PUBLICATION

Offices:New Delhi New York St Louis San Francisco AucklandBogotá Caracas Kuala Lumpur Lisbon London Madrid Mexico City Milan Montreal San Juan Santiago Singapore Sydney Tokyo Toronto

ISBN : 979-8-30-345680-7
Publications: SCIENTIFIC RESEARCH PUBLICATION

Dedicated to

To the divine,
my family,

and my mentors for their
unwavering support and inspiration.

AUTHOR PROFILE

Dr.P.Pradeep Kumar

Assistant Professor and Head, Department of Computer Science-Artificial Intelligence & Machine Learning, K.S.Rangasamy College of Arts & Science (Autonomous), Tiruchengode. He has a professional experience of 11 years and 3 years of industrial experience. Completed his research work in the area of Grid Computing and published research papers in peer reviewed journals. Further publications in the area of Grid Computing, Cloud Computing, Networking, etc. He has acted as a resource person and delivered guest lectures on topics like Design of Algorithms, Machine Learning, Database Management System, Python Programming, Networking, R Programming, etc.

Mr.V.M.Kavin Chakaravarthi

Corporate Trainer at AdroIT Technologies Innovative Solutions Pvt. Ltd. with 3 years of experience in technical training and project-based learning. His core expertise includes Machine Learning, Design Thinking, Figma, and Python for Data Science. He has trained numerous undergraduate students across various institutions and has also conducted Faculty Development Programs (FDP) on emerging technologies like Oracle APEX and IoT. His sessions focus on bridging the gap between academia and industry through hands-on, real-time learning approaches.

Dr.M.Rajeshkumar

Assistant Professor and Head, Department of Computer Applications with Augmented Reality and Virtual Reality, AJK College of Arts & Science, Coimbatore. He has a professional experience of more than 13 years and completed his research work in the area of Wireless Sensor Network in Precision Agriculture Field and published research papers in peer reviewed journals in the area of Wireless sensor Network, Data Analytics, Artificial Intelligence and Neural Networks, etc.

Dr.N.Kalaichelvi

Assistant Professor in the Department of Advanced Computing and Analytics, Vels Institute of Science, Technology and Advanced Studies, Pallavaram, Chennai. She completed her UG in Physics and PG – MCA at The Gandhigram Rural Institute (Deemed to be University) during 2004-2007 and 2007-2010 respectively. She has been working as Assistant Professor in the Centre for Geoinformatics, The Gandhigram Rural Institute (GRI) for three years, an year in Prince Sri Venkateshwara College of Arts and Science, Gowrivakkam, Chennai, two years in Fatima College (Autonomous) in Madurai and one year in Mohamed Sathak College of Arts and Science, Chennai

Dr.V.Deepa

Associate Professor at the Department of Computer Science. She earned her Ph.D. and has 16 years of experience in computer science. She has written three books on Machine Learning & Applications and Gesture Recognition. Published around 20 publications in National and International journals. Served as a Resource Person, conducting over thirty programs at various Intuitions in the field of RPA. Chaired national conferences at several colleges on IOT and machine learning concepts. Major research Areas Digital Image Processing and Robert Process Automation.

FOREWORD

As the software development landscape evolves at a rapid pace, Java remains a prominent and resilient programming language. With the emergence of contemporary development practices such as cloud-native architectures, microservices, reactive systems, and DevOps, it has become essential for Java to modernize and integrate with next-generation tools and methodologies. Next Gen Java: Tools, Patterns, and Integration addresses this evolution with both clarity and depth, offering a thorough guide to constructing robust, scalable, and future-proof Java applications.

This book serves as a timely resource for developers, architects, and technical leaders who are navigating the intricacies of today's software ecosystems. It carefully examines crucial tooling—from modern build tools and containerization to continuous integration pipelines and observability stacks—while also exploring architectural patterns such as event-driven design, domain-driven design, and reactive programming. Beyond theoretical concepts, it highlights practical integration techniques with emerging technologies like Kubernetes, Kafka, and cloud platforms.

By merging the core strengths of Java with the advancements of next-generation development, this book prepares readers with the mindset and technical expertise required for creating intelligent, maintainable, and high-performing systems. Whether you are modernizing legacy systems or developing new applications from scratch, this guide will be your reliable companion on the path to achieving engineering excellence within the Java ecosystem.

PREFACE

Java has significantly progressed since its creation in the mid-1990s. What started as a straightforward, object-oriented programming language has transformed into a robust ecosystem that supports a wide range of applications, from enterprise backends to cloud-native solutions and Internet of Things (IoT) devices. As the landscape of software development has evolved towards agility, scalability, and integration across distributed systems, Java has adapted—thanks to its extensive community, strong tooling support, and ongoing evolution through the introduction of new features and frameworks.

Next Gen Java: Tools, Patterns, and Integration emerged from the necessity to connect traditional Java development with the swiftly evolving realm of modern software architecture. Today's developers are expected not only to produce clean and efficient code but also to grasp concepts such as containerization, CI/CD, observability, event streaming, and beyond. This book seeks to offer a practical roadmap for Java professionals aiming to remain relevant and effective in this new era.

Instead of concentrating solely on syntax or specific frameworks, this book highlights integration—illustrating how to link various tools, technologies, and design patterns within a Java-centric environment. From utilizing Docker and Kubernetes for deployment to connecting with messaging platforms like Apache Kafka, and applying architectural patterns such as microservices, CQRS, and reactive programming, each chapter contributes to a comprehensive understanding of contemporary Java application development.

This book is designed for intermediate to advanced Java developers, software architects, and DevOps engineers who aspire to create production-grade systems utilizing modern best practices. Whether you are modernizing legacy monoliths or crafting cloud-native microservices from the ground up, this guide will assist you in making informed, future-ready decisions.

IMPORTANT TERMS AND ABBREVIATIONS

Core Java & Programming Concepts

- JVM – Java Virtual Machine: The execution engine that runs Java bytecode on various platforms.
- JDK – Java Development Kit: A toolkit utilized for developing Java applications.
- JRE – Java Runtime Environment: A collection that supplies the libraries and components necessary to execute Java applications.
- POJO – Plain Old Java Object: A basic Java object that is not subject to any specific constraints or requirements.
- OOP – Object-Oriented Programming: A programming model centered around the idea of "objects."

Architectural Patterns & Design

- DDD – Domain-Driven Design: A methodology in software development that emphasizes the importance of the business domain.
- CQRS – Command Query Responsibility Segregation: A design pattern that differentiates between read and write operations to enhance scalability and clarity.
- EDA – Event-Driven Architecture: An architectural style where components interact through events.
- SOA – Service-Oriented Architecture: A design paradigm in which services are made available to other components over a network.
- MVC – Model-View-Controller: A software design pattern utilized for creating user interfaces.

Tools & Technologies

- Maven / Gradle – Widely used Java build automation tools.
- Docker – A platform designed for the development, shipping, and execution of containerized applications.
- Kubernetes (K8s) – An open-source framework for automating the deployment, scaling, and management of containerized applications.
- Kafka – A distributed event streaming platform employed for high-performance data pipelines and streaming analytics.

- Spring Boot – A Java-based framework that facilitates the rapid development of production-ready microservices.
- Hibernate / JPA – Object-Relational Mapping (ORM) tools for performing database operations in Java.

TABLE OF CONTENTS

Chapter 1 Introduction to Design Patterns

1.1 Introduction

In the late 1970s, an architect named Christopher Alexander introduced the concept of patterns. His work concentrated on identifying patterns of solutions that address specific sets of forces within distinct contexts.

Christopher Alexander, who was both a civil engineer and an architect, focused his patterns on building architecture. However, his contributions sparked interest within the object-oriented (OO) community, leading several innovators to create patterns for software design. Among these were Kent Beck and Ward Cunningham, who presented a collection of design patterns for Smalltalk at an OOPSLA conference. James Coplien also played a significant role in advocating for the principles of patterns.

The patterns community began to flourish at OOPSLA, providing a platform for members to exchange their innovations and ideas regarding patterns. Another crucial forum for the development of the patterns movement was the Hillside Group, founded by Kent Beck and Grady Booch.

Design patterns represent the distillation of expertise from a vibrant and dynamic community. This exemplifies crowdsourcing at its finest. The patterns community has expanded significantly over the years since the original GoF work, becoming both large and energetic. Grady Booch and Celso Gonzalez have been diligently gathering every pattern they can discover within the industry, amassing over 2,000 patterns to date.

This course is dedicated to Design Patterns. Throughout this course, we will introduce you to the most useful and renowned design patterns. In this lesson, we will first explore what Design Patterns truly are, their applications, the reasons for their use, and how to effectively implement them.

Subsequently, we will examine how patterns are organized and categorized into various groups based on their behavior and structure. In the following lessons, we will discuss different design patterns individually, delving deeply into each one and demonstrating how to implement them in Java.

1.2 What are Design Patterns

As an Object-Oriented developer, one might assume that our code encompasses all the advantages offered by the Object-Oriented programming language.

The code we have developed is sufficiently flexible, allowing us to implement changes with minimal effort or discomfort. Our code is designed for reusability, enabling us to utilize it in various contexts without complications. We can easily maintain our code, and modifications made to one section will not impact other sections of the code.

Regrettably, these benefits do not materialize automatically. As developers, it is our duty to architect the code in a manner that ensures flexibility, maintainability, and reusability.

Design is an art form that is honed through experience. However, there exists a collection of solutions that have been crafted by advanced and seasoned developers who have encountered and resolved similar design challenges. These solutions are referred to as Design Patterns.

Design Patterns encapsulate the expertise gained in crafting Object-Oriented code.

Design Patterns represent general reusable solutions to frequently encountered issues. They embody best practices employed by experienced developers. Patterns do not constitute complete code; rather, they serve as templates that can be applied to specific problems. Patterns are reusable; they can be utilized for similar design challenges across various domains. In essence, we can regard patterns as formal documents that outline recurring design issues and their corresponding solutions. A pattern applied in one practical scenario can also be reused in different contexts.

Christopher once stated, "Each pattern describes a problem that occurs repeatedly in our environment, and then outlines the essence of the solution to that problem, in such a way that you can apply this solution countless times, without ever executing it in the same manner twice."

Typically, a pattern comprises four fundamental elements:

• The name of the pattern serves to assign a unique and significant designation to the pattern that outlines a design issue and its corresponding solution. By naming a design pattern, it becomes easier to reference it in discussions with others. Additionally, it facilitates the creation of documentation, and utilizing the appropriate terminology aids in conceptualizing the design.

• The problem section delineates the circumstances under which the pattern should be utilized. It elaborates on the issue at hand and its surrounding context. This may include specific design challenges, such as the representation of algorithms as objects. It may also detail class or object structures that indicate a rigid design. Occasionally, the problem statement will enumerate conditions that must be satisfied prior to the application of the pattern.

• The solution outlines the components that constitute the design, including their interrelations, responsibilities, and collaborative interactions. While the solution does not provide complete code, it serves as a template that can be implemented with actual code. Rather, the pattern offers an abstract representation of a design issue and illustrates how a general configuration of elements (in this case, classes and objects) addresses it.

• The outcomes and implications of employing the pattern are discussed. The implications for software typically involve trade-offs related to space and time. They may also touch upon language and implementation considerations. Given that reuse is frequently a critical aspect of object-oriented design, the implications of a pattern encompass its effects on a system's flexibility, extensibility, or portability. Clearly articulating these implications aids in understanding and assessing them.

1.3 Why use them

- Flexibility: By utilizing design patterns, your code gains flexibility. This approach facilitates the appropriate level of abstraction, resulting in objects that are loosely coupled with one another, thereby simplifying code modifications.

- Reusability: Objects and classes that are loosely coupled and cohesive enhance the reusability of your code. Such code is also easier to test compared to code that is highly coupled.

- Shared Vocabulary: A shared vocabulary simplifies the process of communicating your code and ideas with fellow team members. It fosters a greater understanding among team members regarding the code.

- Capture best practices: Design patterns encapsulate solutions that have been effectively implemented to address various problems. By studying these patterns and their associated issues, novice developers can gain significant insights into software design.

Design patterns facilitate the reuse of successful designs and architectures.

Articulating proven techniques as design patterns renders them more accessible to developers working on new systems. Design patterns assist in selecting design alternatives that enhance system reusability while steering clear of options that may hinder it. Furthermore, design patterns can enhance the documentation and maintenance of existing systems by providing a clear specification of class and object interactions along with their intended purpose. In essence, design patterns enable designers to achieve a

1.4 How to select and use one

There exists a variety of design patterns available for selection; to make an informed choice, one must possess a comprehensive understanding of each pattern. There are numerous design patterns that appear quite similar to one another. They address nearly identical design issues and often share comparable implementations. A profound comprehension of these patterns is essential to apply the correct design pattern to a specific design challenge.

Initially, it is crucial to determine the nature of the design problem at hand. Design problems can be classified into three categories: creational, structural, or behavioral. Based on this classification, you can filter the patterns and select the most suitable one. For instance:

• If there are excessive instances of a class that represent a singular entity, where the property values of the objects are identical and utilized solely in a read-only manner: the Singleton pattern is appropriate for this design issue, as it guarantees only one instance exists throughout the application. This also aids in reducing memory consumption.

• If classes exhibit excessive interdependence, where a modification in one class impacts all other dependent classes: the Bridge, Mediator, or Command patterns can be employed to address this design concern.

• If there are two distinct incompatible interfaces in separate sections of the code, and there is a requirement to convert one interface into another for the client code to function correctly: the Adapter pattern is suitable for this scenario.

A single design pattern may be applicable to resolve multiple design issues, and conversely, a single design problem may be addressed by various design patterns. While there may be numerous design problems and corresponding solutions, the selection of the most fitting pattern relies on your expertise and understanding of the design patterns, as well as the existing codebase.

1.5 Categorization of patterns

Design patterns can be classified into the following categories:

• Creational patterns
• Structural patterns
• Behavioral patterns

Creational Patterns

Creational design patterns are utilized to structure the process of object instantiation. These patterns leverage inheritance to modify how objects are created.

There are two main themes that recur in these patterns. Firstly, they encapsulate knowledge regarding which specific classes the system employs. Secondly, they conceal the methods by which instances of these classes are generated and assembled. The system, as a whole, is only aware of the objects through their interfaces as defined by abstract classes. As a result, creational patterns provide significant flexibility regarding what is created, who is responsible for its creation, how it is created, and when it occurs.

There may be instances where two or more patterns appear suitable as solutions to a problem. In other cases, two patterns may complement each other; for instance, the Builder pattern can be combined with another pattern to determine which components should be constructed.

Structural Patterns

Structural patterns focus on the composition of classes and objects to form larger structures. Structural class patterns utilize inheritance to create interfaces or implementations. For example, consider how multiple inheritance merges two or more classes into a single entity. The outcome is a class that integrates the attributes of its parent classes. This pattern is especially beneficial for enabling independently developed class libraries to function cohesively.

Instead of composing interfaces or implementations, structural object patterns illustrate methods for composing objects to achieve new functionalities. The enhanced flexibility of object composition arises from the capacity to alter the composition at runtime, which is not feasible with static class composition.

Behavior Patterns

Behavioral patterns focus on algorithms and the distribution of responsibilities among objects. These patterns not only describe the relationships between objects or classes but also the communication patterns that exist between them. They characterize complex control flows that can be challenging to track during runtime. These patterns redirect your attention from the flow of control, allowing you to concentrate on the interactions between objects.

Behavioral object patterns prioritize object composition over inheritance. Some patterns illustrate how a collection of peer objects collaborate to accomplish a task that cannot be executed by a single object alone. A critical consideration in this context is how peer objects are aware of one another. While peers could hold explicit references to each other, this would lead to increased coupling. In a worst-case scenario, every object would be aware of all others. The Mediator pattern circumvents this issue by introducing a mediator object that sits between the peers. This mediator facilitates the necessary indirection for maintaining loose coupling.

The following tables present the list of patterns categorized accordingly:

Creational Patterns	**Structural Patterns**	**Behavioral Patterns**
Abstract Factory	Adapter	Chain of Responsibility
Builder	Bridge	Command
Factory Method	Composite	Interpreter
Prototype	Decorator	Iterator
Singleton	FaÃ§ade	Mediator
	Flyweight	Memento
	Proxy	Observer
		State
		Strategy
		Template Method
		Visitor

PRACTISE SET

MULTIPLE CHOICES QUESTION

1. What is a design pattern in software development?
A) A programming language
B) A reusable solution to a common problem in software design
C) A hardware configuration
D) A database schema

2. Why are design patterns useful?
A) They make the code run faster
B) They offer ready-made solutions for recurring design problems
C) They reduce the size of code
D) They automate code testing

3. Which of the following is NOT a category of design patterns?
A) Creational
B) Structural
C) Conditional
D) Behavioral

4. When should you consider using a design pattern?
A) When a design problem is unique and unlikely to occur again
B) When solving a well-known, recurring problem in software design
C) Only during code optimization
D) Only for large-scale enterprise projects

5. What is the main focus of creational design patterns?
A) Defining the structure of software
B) Controlling object creation mechanisms
C) Managing object interactions
D) Optimizing runtime performance

6. Structural design patterns deal with:
A) How objects are created
B) The behavior of objects

C) How objects and classes are composed to form larger structures
D) User interface layout

7. Behavioral design patterns are concerned with:
A) Object creation
B) Static class structure
C) Communication between objects
D) Hardware control

8. Which of the following is a creational design pattern?
A) Adapter
B) Strategy
C) Factory Method
D) Observer

9. How should one choose a design pattern to implement?
A) Pick one randomly
B) Use the most complex one
C) Analyze the problem and match it with a pattern that solves similar issues
D) Always use behavioral patterns

10. Design patterns are best described as:
A) Code libraries
B) Algorithms
C) General reusable solutions to common design problems
D) User interface templates

BRACE YOURSELF

1. Define design patterns in the context of software development. Explain how they differ from algorithms or code libraries.

2. Discuss the main benefits of using design patterns in software engineering. Provide examples where applicable.

3. Explain the steps or criteria involved in selecting the most appropriate design pattern for a given problem.

4. Categorize design patterns into creational, structural, and behavioral patterns. Briefly describe what each category aims to solve.

5. Describe creational design patterns. What kind of problems do they address in object creation? Mention at least two examples.

6. Explain structural design patterns. How do they help manage relationships between different parts of a system? Provide an example.

7. What are behavioral design patterns? How do they improve communication and control between objects in software design?

8. Why is it not advisable to forcefully use a design pattern in every software solution? Support your answer with reasoning.

9. Give a real-world analogy of any one design pattern (e.g., Factory, Adapter, Observer). Explain how it maps to a programming scenario.

10. How do design patterns promote code reusability, maintainability, and scalability? Illustrate your answer with examples.

Chapter 2 Adapter Design Pattern

2.1 Adapter Pattern

A software developer named Max has been involved in the development of an e-commerce website. This website enables users to shop and make payments online. It is integrated with a third-party payment gateway, allowing users to settle their bills using credit cards. Everything was proceeding smoothly until Max's manager summoned him to discuss a modification in the project.

The manager informed him that they intend to switch the payment gateway vendor, and he is required to implement this change in the code.

The challenge that arises is that the site is currently linked to the Xpay payment gateway, which utilizes an Xpay type of object. The new vendor, PayD, exclusively accepts PayD type objects for processing. Max is reluctant to modify the entire set of 100 classes that reference an object of type XPay. This situation also heightens the risk associated with the project, which is already operational in production. He is unable to alter the third-party payment gateway tool. The issue has emerged due to the incompatible interfaces between the two distinct segments of the code. To ensure the process functions correctly, Max must devise a method to align the code with the API provided by the vendor.

Current Code with the Xpay's API

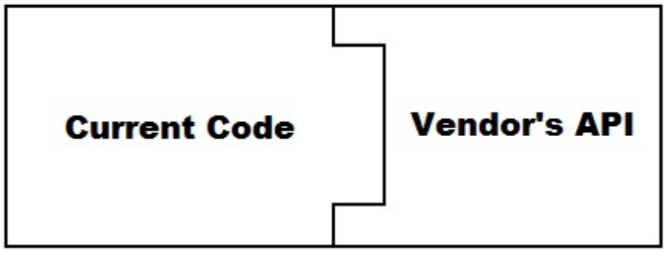


Figure 2.1: screenshot

Currently, the existing code interface does not align with the new vendor's interface.

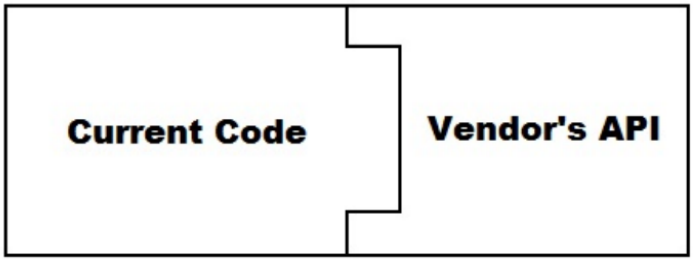


Figure 2.1: screenshot

2.2 An Adapter to rescue

What Max requires in this context is an Adapter that can function as an intermediary between the code and the vendor's API, facilitating the flow of the process.

Before delving into the solution, let us first explore what an adapter is and how it operates.

There may be instances where two objects do not align as they should in order to accomplish the task at hand. This scenario can occur when attempting to integrate legacy code with new code, or when modifying a third-party API within the code. This incompatibility arises from the differing interfaces of the two objects that do not mesh well together.

The Adapter pattern enables you to modify what an object or class presents to meet the expectations of another object or class. It transforms the interface of a class into a different interface that the client anticipates. This pattern allows classes to collaborate that otherwise could not due to incompatible interfaces. It provides a means to rectify the interface between objects and classes without necessitating direct alterations to the objects and classes themselves.

You can envision an Adapter as a real-world connector that links two distinct pieces of equipment that cannot be directly connected. An adapter resides between these pieces of equipment, capturing the flow from one and delivering it to the other in the desired format, which would otherwise be unattainable due to their incompatible interfaces.

An adapter employs composition to retain the object it is intended to adapt, and when the adapter's methods are invoked, it translates those requests into a format that the adapted object can comprehend, subsequently relaying the requests to the adapted object. The code that interacts with the adapter remains unaware that it is not engaging with the type of object it presumes, but rather with an adapted object.

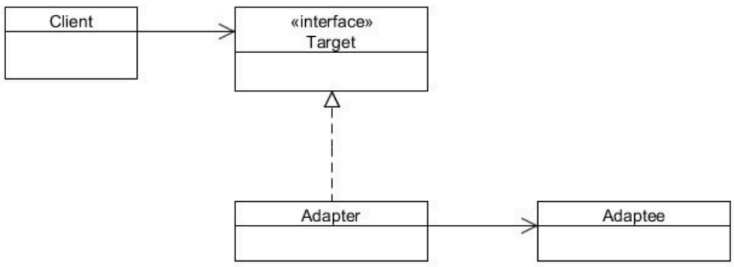


Figure 2.2: screenshot

2.3 Solution to the problem

At present, the code is accessible through the Xpay interface. The interface resembles the following:

```
package com.javacodegeeks.patterns.adapterpattern.xpay;
public interface Xpay {
public String getCreditCardNo();
public String getCustomerName();
public String getCardExpMonth();
public String getCardExpYear();
public Short getCardCVVNo();
public Double getAmount();
public void setCreditCardNo(String creditCardNo);
public void setCustomerName(String customerName);
public void setCardExpMonth(String cardExpMonth);
public void setCardExpYear(String cardExpYear);
public void setCardCVVNo(Short cardCVVNo);
public void setAmount(Double amount);
```

```
}
```

This text includes a collection of setter and getter methods utilized to retrieve information regarding the credit card and the customer's name. The Xpay interface is implemented within the code, allowing for the instantiation of an object of this type and making the object accessible to the vendor's API.

The subsequent class outlines the implementation of the Xpay interface.

```
package com.javacodegeeks.patterns.adapterpattern.site;
import com.javacodegeeks.patterns.adapterpattern.xpay.Xpay;
public class XpayImpl implements Xpay{
private String creditCardNo;
private String customerName;
private String cardExpMonth;
private String cardExpYear;
private Short cardCVVNo;
private Double amount;
@Override
public String getCreditCardNo() {
return creditCardNo;
}
@Override
public String getCustomerName() {
return customerName;
}
@Override
public String getCardExpMonth() {
return cardExpMonth;
}
@Override
public String getCardExpYear() {
return cardExpYear;
}
@Override
public Short getCardCVVNo() {
return cardCVVNo;
}
```

```
@Override
public Double getAmount() {
return amount;
}
@Override
public void setCreditCardNo(String creditCardNo) {
this.creditCardNo = creditCardNo;
}
@Override
public void setCustomerName(String customerName) {
this.customerName = customerName;
}
@Override
public void setCardExpMonth(String cardExpMonth) {
this.cardExpMonth = cardExpMonth;
}
@Override
public void setCardExpYear(String cardExpYear) {
this.cardExpYear = cardExpYear;
}
@Override
public void setCardCVVNo(Short cardCVVNo) {
this.cardCVVNo = cardCVVNo;
}
@Override
public void setAmount(Double amount) {
this.amount = amount;
}
}
```

New vendor's key interface looks like this:

```
package com.javacodegeeks.patterns.adapterpattern.payd;
public interface PayD {
public String getCustCardNo();
public String getCardOwnerName();
public String getCardExpMonthDate();
public Integer getCVVNo();
```

```
public Double getTotalAmount();
public void setCustCardNo(String custCardNo);
public void setCardOwnerName(String cardOwnerName);
public void setCardExpMonthDate(String cardExpMonthDate);
public void setCVVNo(Integer cVVNo);
public void setTotalAmount(Double totalAmount);
```

As you can observe, this interface comprises a variety of methods that must be implemented within the code. However, since Xpay constitutes the majority of the code, altering the entire collection of classes is quite challenging and poses significant risks.

We require a solution that can meet the vendor's requirements for processing payments while minimizing or eliminating modifications to the existing code. The solution is offered by the Adapter pattern.

We will develop an adapter of type PayD, which will encapsulate an Xpay object (the type it is intended to adapt).

```
package com.javacodegeeks.patterns.adapterpattern.site;
import com.javacodegeeks.patterns.adapterpattern.payd.PayD;
import com.javacodegeeks.patterns.adapterpattern.xpay.Xpay;
public class XpayToPayDAdapter implements PayD{
private String custCardNo;
private String cardOwnerName;
private String cardExpMonthDate;
private Integer cVVNo;
private Double totalAmount;
private final Xpay xpay;
public XpayToPayDAdapter(Xpay xpay){
this.xpay = xpay;
setProp();
}
@Override
public String getCustCardNo() {
return custCardNo;
}
@Override
```

```
public String getCardOwnerName() {
return cardOwnerName;
}
@Override
public String getCardExpMonthDate() {
return cardExpMonthDate;
}
@Override
public Integer getCVVNo() {
return cVVNo;
}
@Override
public Double getTotalAmount() {
return totalAmount;
}
@Override
public void setCustCardNo(String custCardNo) {
this.custCardNo = custCardNo;
}
@Override
public void setCardOwnerName(String cardOwnerName) {
this.cardOwnerName = cardOwnerName;
}

@Override
public void setCardExpMonthDate(String cardExpMonthDate) {
this.cardExpMonthDate = cardExpMonthDate;
}
@Override
public void setCVVNo(Integer cVVNo) {
this.cVVNo = cVVNo;
}
@Override
public void setTotalAmount(Double totalAmount) {
this.totalAmount = totalAmount;
}
private void setProp(){
setCardOwnerName(this.xpay.getCustomerName());
```

```
setCustCardNo(this.xpay.getCreditCardNo());
setCardExpMonthDate(this.xpay.getCardExpMonth()+"/"+this.xpay. ←-
getCardExpYear());
setCVVNo(this.xpay.getCardCVVNo().intValue());
setTotalAmount(this.xpay.getAmount());
}
}
```

In the code presented above, we have established an Adapter (XpayToPayDAdapter). This adapter implements the PayD interface, as it is necessary to simulate a PayD type object. The adapter utilizes object composition to contain the object it is intended to adapt, which is an Xpay type object. The object is provided to the adapter via its constructor.

It is important to note that we are dealing with two incompatible types of interfaces that must be integrated using an adapter to ensure the functionality of the code. These two interfaces possess distinct sets of methods. However, the primary objective of these interfaces is quite similar, namely, to supply customer and credit card information to their respective vendors.

The setProp() method of the aforementioned class is employed to transfer the properties of the xpay object into the payD object. We configure the methods that perform similar functions in both interfaces. Nevertheless, there is only one method in the PayD interface for setting the month and year of the credit card, in contrast to two methods available in the Xpay interface. We combine the output of the two methods from the Xpay object (this.xpay.getCardExpMonth()+"/"+this.xpay.getCardExpYear()) and assign it to the setCardExpMonthDate() method.

Now, let us evaluate the above code to determine if it effectively addresses Max's issue.

```
package com.javacodegeeks.patterns.adapterpattern.site;
import com.javacodegeeks.patterns.adapterpattern.payd.PayD;
import com.javacodegeeks.patterns.adapterpattern.xpay.Xpay;
public class RunAdapterExample {
public static void main(String[] args) {
// Object for Xpay
```

```
Xpay xpay = new XpayImpl();
xpay.setCreditCardNo("4789565874102365");
xpay.setCustomerName("Max Warner");
xpay.setCardExpMonth("09");
xpay.setCardExpYear("25");
xpay.setCardCVVNo((short)235);
xpay.setAmount(2565.23);
PayD payD = new XpayToPayDAdapter(xpay);
testPayD(payD);
}
private static void testPayD(PayD payD){
System.out.println(payD.getCardOwnerName());
System.out.println(payD.getCustCardNo());
System.out.println(payD.getCardExpMonthDate());
System.out.println(payD.getCVVNo());
System.out.println(payD.getTotalAmount());
}
}
```

In the aforementioned class, we initially created an Xpay object and configured its properties. Subsequently, we established an adapter and provided it with the xpay object through its constructor, assigning it to the PayD interface. The static method testPayD() accepts a PayD type as an argument, which executes and displays its methods for testing purposes. Given that the type supplied to the testPayD() method is of the PayD type, the method will successfully execute the object without any issues. As previously mentioned, we supplied an adapter to it, which resembles a PayD type, but internally encapsulates an Xpay type object.

Therefore, in Max's project, our sole requirement is to implement the vendor's API within the code and pass this adapter to the vendor's method to facilitate the payment process. There is no need to modify any part of the existing code.

2.4 Class Adapter

There are two categories of adapters: the object adapter and the class adapter. Up to this point, we have examined the object adapter example, which

utilizes object composition. In contrast, the class adapter depends on multiple inheritance to convert one interface into another. Since Java does not allow multiple inheritance, we are unable to provide an example of it; however, you may consider this concept and apply it in one of your preferred Object-Oriented Languages, such as C++, which does support multiple inheritance.

To create a class adapter, the adapter would publicly inherit from Target and privately inherit from Adaptee. Consequently, the adapter would be a subtype of Target, but not of Adaptee.

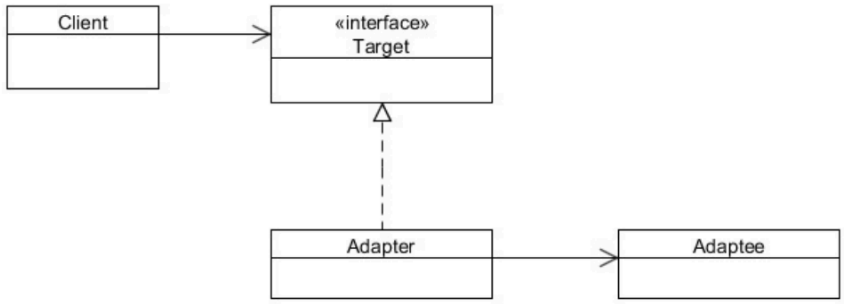


Figure 2.4: screenshot

2.5 When to use Adapter Pattern

The Adapter pattern is applicable in the following scenarios:

• When there is a pre-existing class whose interface does not align with your requirements.

• When the goal is to develop a reusable class that can interact with unrelated or unexpected classes, meaning classes that do not inherently possess compatible interfaces.

• When multiple existing subclasses are available, but it is not feasible to modify their interfaces through subclassing each one. An object adapter can modify the interface of its parent class.

PRACTISE SET

MULTIPLE CHOICES QUESTION

1. What is the primary purpose of the Adapter Design Pattern?
A) To simplify object creation
B) To resolve incompatible interfaces
C) To monitor object behavior
D) To increase memory usage

2. Which of the following best describes the Adapter Pattern?
A) It creates new objects without specifying the exact class
B) It acts as a bridge between two incompatible interfaces
C) It adds new behavior to an existing object
D) It allows multiple instances of a class

3. In which design pattern category does the Adapter Pattern belong?
A) Creational
B) Structural
C) Behavioral
D) Concurrent

4. What problem does the Adapter Pattern aim to solve?
A) High memory usage
B) Incompatibility between client and service interfaces
C) Excessive object instantiation
D) Slow application performance

5. Which of the following best represents a real-life example of the Adapter Pattern?
A) A class that extends multiple classes
B) A translator converting speech from one language to another
C) A clock showing time
D) A printer printing a document

6. What is the difference between a class adapter and an object adapter?

A) Class adapter uses inheritance, object adapter uses composition
B) Class adapter uses composition, object adapter uses inheritance
C) Both use inheritance
D) Both use composition

7. When should you use the Adapter Pattern?
A) When you want to simplify a complex class
B) When two classes with incompatible interfaces need to work together
C) When you want to enhance runtime performance
D) When objects need to be created based on a condition

8. Which of the following is a key characteristic of the Adapter Pattern?
A) Requires subclassing the original interface
B) Requires redesigning the existing interface
C) Works without changing the existing code
D) Breaks encapsulation

9. What does a class adapter rely on to function?
A) Aggregation
B) Interface implementation
C) Inheritance (extends both classes/interfaces)
D) Abstract classes only

10. The Adapter Pattern is useful when:
A) You want to alter business logic
B) A client expects a specific interface, but the service doesn’t match it
C) You are using a pattern from the behavioral category
D) You are creating a UI design

BRACE YOURSELF

1. Define the Adapter Design Pattern. What is its main purpose in software design?

2. Explain a scenario where incompatible interfaces can create problems in a system. How can an adapter help resolve this issue?

3. Discuss how the Adapter Pattern provides a solution to interface incompatibility. Include a brief example.

4. What is a class adapter? How does it differ from an object adapter in terms of implementation and use?

5. Describe a real-world analogy of the Adapter Pattern and relate it to a software development context.

6. In what types of applications or situations is the Adapter Pattern especially useful? Provide examples.

7. Compare and contrast the Adapter Pattern with the Decorator and Facade patterns. In what situations would you use each?

8. What are the advantages and potential drawbacks of using the Adapter Pattern in software design?

9. Describe the role of composition in implementing the Adapter Pattern. Why is it preferred in some cases over inheritance?

10. Illustrate with a UML diagram or pseudocode how a class adapter is implemented. Label all relevant components (Target, Adapter, Adaptee, Client).

Chapter 3 Facade Design Pattern

3.1 Introduction

In this lesson, we will explore another structural pattern, specifically the Facade Pattern. However, prior to delving into its specifics, let us first address a problem that this particular pattern aims to resolve.

Your company operates as a product-based entity and has recently introduced a product to the market called Schedule Server. This product functions as a server in its own right, designed to manage various jobs. These jobs can encompass a range of tasks, such as sending lists of emails, SMS messages, reading or writing files from a designated location, or simply transferring files from one source to another. Developers utilize this product to oversee such tasks, allowing them to focus more on their core business objectives. The server executes each job at the designated time and autonomously manages underlying issues such as concurrency and security. As a developer, one only needs to code the pertinent business requirements, as a comprehensive set of API calls is available to schedule jobs according to their specifications.

Everything was proceeding smoothly until clients began to express concerns regarding the initiation and termination processes of the server. They indicated that, while the server operates effectively, the processes for starting and shutting it down are overly complicated, and they desire a more straightforward method for these actions. The server has presented a complex interface to the clients, which they find somewhat burdensome.

We must provide a simplified approach to starting and stopping the server.

A complex interface for the client is already viewed as a flaw in the design of the existing system. However, whether fortunate or unfortunate, we cannot commence the design and coding from the ground up. We require a solution to this issue that will render the interface more user-friendly.

The Facade Pattern can assist us in addressing this design challenge. Before we proceed, let us first examine the Facade Pattern.

3.2 What is the Facade Pattern

The Facade Pattern simplifies a complex interface by utilizing a Facade class. It offers a cohesive interface to a collection of interfaces within a subsystem. The Facade establishes a higher-level interface that enhances the usability of the subsystem.

The Facade consolidates the intricate low-level interfaces of a subsystem to provide a straightforward method for accessing that interface. It merely adds a layer to the complex interfaces of the subsystem, facilitating ease of use.
The Facade does not encapsulate the classes or interfaces of the subsystem; it simply offers a streamlined interface to their functionalities. Clients can directly access these classes, thereby retaining the complete functionality of the system for those who may require it.

A Facade not only simplifies an interface but also decouples a client from a subsystem. It follows the Principle of Least Knowledge, which helps to prevent tight coupling between the client and the subsystem. This approach allows for flexibility: for instance, if the company wishes to introduce additional steps to initiate or terminate the Schedule Server, which have their own distinct interfaces, the client code can remain unchanged if it is written to interact with the facade rather than the subsystem. Only the facade would need to be updated, which could be delivered to the client in a new version.

Clients interact with the subsystem by sending requests to the Facade, which then relays them to the relevant subsystem object(s). While the subsystem objects carry out the actual tasks, the facade may also need to perform its own functions to translate its interface into subsystem interfaces. Clients utilizing the facade do not need to directly access its subsystem objects.

It is important to note that, similar to an Adapter, a Facade can encapsulate multiple classes; however, a facade is specifically designed to simplify the use of a complex interface, whereas an adapter is intended to convert an interface into one that the client expects.

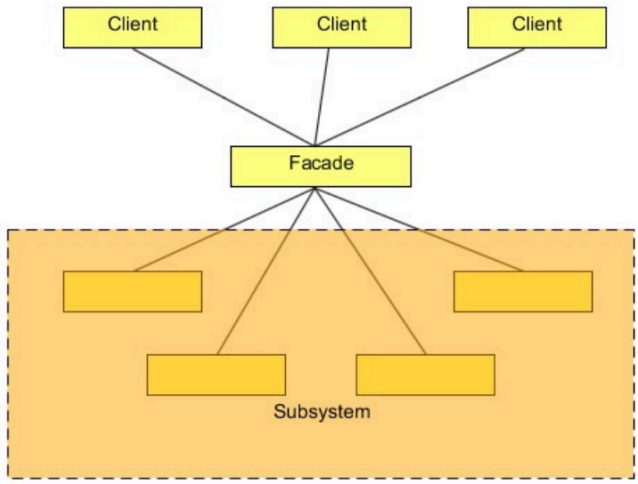


Figure 3.2: screenshot

3.3 Solution to the problem

The issue encountered by clients when utilizing the Schedule Server is the intricacy involved in starting and stopping its services. Clients desire a more straightforward approach to accomplish this task. Below is the code that clients are required to implement in order to start and stop the server.

```
ScheduleServer scheduleServer = new ScheduleServer();
```

To initiate the server, the client must instantiate an object of the ScheduleServer class and subsequently invoke the methods listed below in the specified order to start and initialize the server.

```
scheduleServer.startBooting();
scheduleServer.readSystemConfigFile();
scheduleServer.init();
scheduleServer.initializeContext();
scheduleServer.initializeListeners();
scheduleServer.createSystemObjects();
System.out.println("Start working.....");
System.out.println("After work done.....");
```

To terminate the server, the client is required to execute the following methods in the same order.

```
scheduleServer.releaseProcesses();
scheduleServer.destory();
scheduleServer.destroySystemObjects();
scheduleServer.destoryListeners();
scheduleServer.destoryContext();
scheduleServer.shutdown();
```

This appears to be a burden for them; they show no interest in engaging with all these tasks, and one might wonder why they would. Although this may seem intriguing to certain clients who are interested in the system's low-level interface, the majority of them found it unappealing.

To address this issue, we will develop a facade class that will encapsulate a server object. This class will offer straightforward interfaces (methods) for the client. Internally, these interfaces will invoke the methods on the server object. Let us first examine the code, after which we will discuss it in greater detail.

```
package com.javacodegeeks.patterns.facadepattern;
public class ScheduleServerFacade {
private final ScheduleServer scheduleServer;
public ScheduleServerFacade(ScheduleServer scheduleServer){
this.scheduleServer = scheduleServer;
}
public void startServer(){
scheduleServer.startBooting();
scheduleServer.readSystemConfigFile();
scheduleServer.init();
scheduleServer.initializeContext();
scheduleServer.initializeListeners();
scheduleServer.createSystemObjects();
}
public void stopServer(){
scheduleServer.releaseProcesses();
scheduleServer.destory();
scheduleServer.destroySystemObjects();
```

```
scheduleServer.destoryListeners();
scheduleServer.destoryContext();
scheduleServer.shutdown();
}
}
```

The class ScheduleServerFacade mentioned above serves as a facade, encapsulating a ScheduleServer object. It creates the server object via its constructor and includes two straightforward methods: startServer() and stopServer(). These methods handle the initiation and termination of the server internally. The client merely needs to invoke these simple methods. Consequently, there is no requirement to call all the lifecycle and destroy methods; only the simple methods need to be called, and the facade class will manage the rest of the process.

The following code illustrates how the facade simplifies a complex interface for ease of use.

```
package com.javacodegeeks.patterns.facadepattern;
public class TestFacade {
public static void main(String[] args) {
ScheduleServer scheduleServer = new ScheduleServer();
ScheduleServerFacade facadeServer = new
ScheduleServerFacade(scheduleServer ←-
);
facadeServer.startServer();
System.out.println("Start working.....");
System.out.println("After work done.....");
facadeServer.stopServer();
}
}
```

Additionally, it is important to understand that while the facade class offers a straightforward interface to the intricate subsystem, it does not fully encapsulate the subsystem. Clients retain the ability to access the low-level interfaces of the subsystem. Therefore, a facade serves as an additional layer, providing a simplified interface to the complex subsystem, yet it does not

entirely obscure direct access to the low-level interfaces of the intricate subsystem.

3.4 Use of the Facade Pattern

Utilize the Facade Pattern when:

• You aim to offer a straightforward interface to a complex subsystem. As subsystems develop, they often become increasingly intricate. The application of most patterns typically leads to a greater number of smaller classes. While this enhances the reusability and customization of the subsystem, it simultaneously complicates usage for clients who do not require customization. A facade can present a simple default representation of the subsystem that suffices for the majority of clients. Only those clients requiring additional customization will need to look beyond the facade.

• There exist numerous dependencies between clients and the implementation classes of an abstraction. Implement a facade to separate the subsystem from clients and other subsystems, thus fostering subsystem independence and portability.

• You have the ability to layer your subsystems. Employ a facade to establish an entry point for each level of the subsystem. In cases where subsystems are interdependent, you can streamline the dependencies among them by ensuring they communicate exclusively through their facades.

PRACTISE SET

MULTIPLE CHOICES QUESTION

1. What is the primary purpose of the Facade Design Pattern?
A) To create new objects efficiently
B) To simplify interactions with complex subsystems
C) To monitor object behavior
D) To dynamically change object behaviour

2. The Facade Pattern belongs to which category of design patterns?
A) Creational
B) Structural
C) Behavioral
D) Concurrency

3. What does the Facade Pattern typically provide to the client?
A) Multiple interfaces for flexible access
B) Direct access to internal subsystem objects
C) A unified and simplified interface to a set of interfaces
D) A pattern for creating families of related objects

4. Which of the following is a key benefit of using the Facade Pattern?
A) Reduces memory usage
B) Simplifies complex subsystems
C) Eliminates the need for any interfaces
D) Prevents inheritance

5. In which scenario should the Facade Pattern be used?
A) When a class has too many responsibilities
B) When you want to hide complex subsystem logic from the client
C) When objects need to notify others automatically
D) When you want to ensure only one instance of a class exists

6. What is the role of the Facade class in the Facade Pattern?
A) It handles object creation logic
B) It acts as an adapter between incompatible interfaces

C) It provides a higher-level interface that makes a subsystem easier to use
D) It stores data in a database

7. How does the Facade Pattern affect system coupling?
A) Increases coupling between subsystems
B) Decreases dependency of client code on internal subsystem details
C) Has no effect on coupling
D) Makes subsystems more dependent on each other

8. Which of the following is an example of using a Facade Pattern in real life?
A) A translator interpreting different languages
B) A remote control that operates various TV functions
C) A factory producing multiple car parts
D) A sorting algorithm organizing numbers

9. Which of the following is NOT a goal of the Facade Pattern?
A) To hide the complexity of a subsystem
B) To provide a simplified interface
C) To reduce the number of classes in a system
D) To promote loose coupling

10. What happens when a client uses a Facade Pattern to access a subsystem?
A) The client must know all internal subsystem details
B) The client interacts through a simplified interface and remains unaware of internal complexity
C) The client directly modifies the internal subsystem classes
D) The client implements the subsystem itself

BRACE YOURSELF

1. Define the Facade Design Pattern. What is its main purpose in software design?

2. Describe a problem scenario where a client must interact with a complex subsystem. How can the Facade Pattern help in such a situation?

3. Explain how the Facade Pattern simplifies client interaction with complex systems. Provide an example in a real-world or software context.

4. What are the benefits of using the Facade Pattern in large-scale applications?

5. In what ways does the Facade Pattern promote loose coupling between the client and the subsystem? Why is this desirable?

6. Describe a real-world analogy of the Facade Pattern and explain how it relates to software design.

7. What are the limitations or potential drawbacks of using the Facade Pattern? Under what circumstances might it not be suitable?

8. Compare the Facade Pattern with the Adapter Pattern. How do their purposes and implementations differ?

9. How does the Facade Pattern help in managing system complexity during system maintenance and evolution?

10. Describe how the Facade Pattern can be implemented in code. Include key components like the Facade class, subsystems, and the client.

Chapter 4 Tools of the Trade

4.1 IDE Faceoff: IntelliJ vs VS Code

IntelliJ IDEA and Visual Studio Code (VS Code) provide unique experiences for Java development, addressing various preferences and project requirements.

IntelliJ IDEA:

- Comprehensive Java IDE: IntelliJ serves as a complete Integrated Development Environment tailored specifically for Java and enterprise development. It features deep integrations with build tools such as Maven and Gradle, along with advanced code analysis and refactoring capabilities readily available.

- Feature-Rich: It offers a wide array of robust features, including intelligent code completion, powerful debugging tools, integrated version control, and extensive support for numerous Java frameworks and technologies (e.g., Spring, JavaFX).

- Resource-Intensive: IntelliJ IDEA may demand more system resources, particularly for larger projects, although recent updates have enhanced its performance.

Advantages of IntelliJ for Java

Exceptional, comprehensive Java support: features such as context-sensitive code completion, extensive refactoring options (including interface extraction, method signature alteration, and pull-up/push-down refactoring), structural search and replace, and inspections (covering nullability, threading, and resource leaks) – significantly superior to standard editors.

Excellent native integration with Java build tools (Maven, Gradle), frameworks (Spring, Hibernate), servers, JPA, and UI development tools.

A solid out-of-the-box "IDE experience": encompassing debugging, test runners, profiling, database management tools, version control, and more.

For Java developers in particular, there is a notably high level of adoption within the industry for "genuine Java projects."

Illustrative features

- Refactorings for renaming, extracting, and changing signatures.
- "Go to definition," "Find usages," and code inspections that identify subtle problems.
- Intelligent completion: cognizant of libraries, generics, and more.
- Integrated support for Spring Boot, JPA, and similar technologies.

Trade-offs

- Due to its extensive features and demanding operations, IntelliJ may require more resources (RAM, CPU), particularly on older systems or when working with large codebases.

- The Ultimate (paid) edition offers additional features; conversely, the free Community edition provides substantial support for Java but is deficient in certain advanced framework and tool integrations.

- If you are engaged in a small project or lightweight development across multiple languages, you may find that not all features are necessary.

Visual Studio Code:

- Lightweight and Versatile Editor: VS Code is fundamentally a code editor that can be transformed into a powerful IDE through the use of extensions. Its lightweight design allows it to function effectively even on less capable machines.

- Extensive Extension Ecosystem: Although VS Code lacks built-in Java-specific features like those found in IntelliJ, the Java Extension Pack offers essential tools for Java development, including language support, debugging, and build tool integration. Additionally, you can further tailor your environment with a variety of other extensions for different languages and functionalities.

- Multi-Language Support: VS Code is exceptionally versatile, supporting a broad spectrum of programming languages, making it an ideal option for developers engaged with multiple technologies beyond Java.

Advantages of VS Code for Java

Lightweight / quick startup / responsive: an excellent option if you are using a modest machine or prefer a more agile environment.

Highly extensible and supports multiple languages: if your workflow involves more than just Java (for instance, Java + JavaScript + Python + Go), VS Code provides a cohesive editor ecosystem.

Free, open-source, and boasts a vast ecosystem of extensions. For Java, you can install the 'Java Extension Pack' (by Red Hat and others), which offers language support, debugging, and a test runner.

Ideal for smaller, simpler Java projects, prototyping, or when you desire minimal overhead.

Trade-offs

Although VS Code can be tailored for Java development, numerous advanced features specific to Java IDEs (such as deep refactoring, inspections, code analysis, and framework-aware support) are less robust or dependent on plugins, potentially falling short of the inherent capabilities of IntelliJ.

Additionally, it may necessitate more initial configuration (selecting/installing extensions, setting up tool integrations) in contrast to IntelliJ, which operates seamlessly for Java.

In the case of large and intricate enterprise Java codebases (characterized by extensive frameworks, multiple modules, etc.), you might encounter the limitations of VS Code more quickly.

Conclusion

To summarize:

Choose IntelliJ if you are dedicated to Java, particularly for large or intricate Java applications that require frameworks, refactoring, team collaboration, and long-term upkeep.

Opt for VS Code when you prefer a lighter option, perhaps if you are working with various languages, handling smaller Java projects, or prioritizing speed and adaptability over comprehensive Java tooling.

And if you wish: Utilize both! Some developers employ VS Code for "quick tasks / scripting / multi-language projects" and IntelliJ for "serious Java modules."

4.2 Build Tools: Maven, Gradle & JBang

The main distinction among Maven, Gradle, and JBang is found in their scope and methods of configuration, although all three serve as build tools within the Java ecosystem.

Maven: The convention-based workhorse

Maven is a well-established and robust build automation tool recognized for its "convention over configuration" philosophy. It operates on a standardized project structure and follows a predefined lifecycle.

- Configuration: It employs a declarative XML file, known as pom.xml, to outline the project's specifics, dependencies, and plugins.

- Workflow: It carries out a sequence of predefined build phases in a set order, including compile, test, package, and install.

Strengths:

- Predictable and consistent: The standardized methodology facilitates developers' transitions between various Maven projects.
- Large ecosystem: Supported by an extensive and continually expanding central repository of libraries.
- Mature: Particularly effective for large, multi-module, and enterprise-level projects where consistency is crucial.

Best for:

- Enterprise applications and intricate, multi-module projects.
- Teams that emphasize a standardized and reliable build process.

Example:

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-core</artifactId>
  <version>5.3.10</version>
</dependency>
```

Gradle: The contemporary, adaptable builder

Gradle serves as a contemporary and adaptable build automation tool that is founded on the principles of Maven and Apache Ant. It is recognized for its exceptional performance and strong support for multi-project builds.

- Configuration: It employs a more expressive and customizable Domain-Specific Language (DSL), which can be crafted in Groovy or Kotlin. This provides enhanced control over the build process.
- Workflow: It structures the build as a Directed Acyclic Graph (DAG) of tasks. This structure facilitates fine-grained control, incremental builds, and parallel task execution, resulting in improved performance.

Strengths:

- Exceptional performance: It outpaces Maven due to its incremental builds and build caching, particularly for extensive projects.

- Adaptability: The Groovy/Kotlin DSL empowers developers to implement intricate or custom build logic that is challenging to achieve with Maven's XML.
- Versatile: It is the standard for Android development and is frequently favored for contemporary microservices and multi-project builds.

Best suited for:

- Large and intricate applications with specific build requirements.
- Microservices-based architectures and Android development.

Example:

```
dependencies {
   implementation 'com.google.guava:guava:31.0.1-jre'
}
```

JBang: The Fast Script Executor

JBang is a relatively recent tool designed to simplify the use of Java as a scripting language, removing the complexities associated with establishing comprehensive projects.

- Configuration: Directives are specified within the script file itself through comments (for instance, //DEPS for dependencies). This enables the entire configuration to reside within a single source file.
- Workflow: Executes Java source files directly from the command line, automatically managing compilation, dependency retrieval, and execution.

Strengths:

- No setup required: Allows developers to write and execute Java with the same ease as scripting languages such as Python.
- Immediate experimentation: Ideal for prototyping, creating small utility applications, and teaching Java without the need for a build system.

- Adaptable to projects: Supports dependencies and multiple source files while also permitting the export of scripts into conventional Maven or Gradle projects.

Best suited for:

- Single-file utility scripts and compact command-line tools.
- Educators, students, and developers engaged in rapid proofs-of-concept.

Example:

```
//DEPS com.google.guava:guava:31.0.1-jre
import com.google.common.collect.Lists;
import java.util.*;

public class HelloJBang {
   public static void main(String[] args) {
      List<String> list = Lists.newArrayList("Vanakkam", "JBang", "Ulagam");
      list.forEach(System.out::println);
   }
}
```

4.3 Testing Arsenal: Mockito, TestNG

Testing Arsenal: Mockito

Mockito is a free and open-source mocking framework designed for Java, which facilitates unit testing by enabling developers to generate "mock" objects that represent external dependencies. A mock serves as a simulated counterpart of a genuine object, which can be programmed to act in a particular manner during testing.

This functionality is essential for separating the class being tested from its collaborators, thereby enhancing the speed, reliability, and independence of tests from external influences such as databases, web services, or network

connections. Mockito is frequently utilized in conjunction with a test runner such as JUnit or TestNG.

Core concepts and features

Mock objects: These are substitute objects that replicate the behavior of actual dependencies. They serve the purpose of isolating the code that is being tested.

Stubbing: This refers to the act of pre-defining how a mock object should behave. You can specify what a mock should return when a particular method is invoked.

Verification: Once a test has been executed, you can confirm that certain methods on a mock object were invoked as anticipated, including the frequency of calls and the arguments used.

Argument matchers: These facilitate flexible verification and stubbing. Rather than supplying precise values, you can utilize generic matchers such as anyString() or anyInt().

Spies: This is a type of partial mock that encapsulates a real object. With a spy, you have the ability to override specific methods while retaining the actual behavior for others.

Annotations: To minimize boilerplate code, Mockito provides annotations for the creation and injection of mocks.

- @Mock: This annotation generates a mock instance of a specified class or interface.
- @Spy: This annotation creates a partial mock of an actual object.
- @InjectMocks: This annotation instantiates the class that is being tested and injects the fields that are annotated with @Mock or @Spy into it.
- @ExtendWith(MockitoExtension.class): This is a JUnit 5 annotation that activates Mockito's annotations for a given test class.

Fundamental usage illustration

Take into account a UserService that relies on a UserRepository to locate a user.

1. Specify the dependencies

```
public class UserService {
    private final UserRepository userRepository;

    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public User getUserById(int id) {
        return userRepository.findById(id);
    }
}

// UserRepository.java (can be an interface)
public interface UserRepository {
    User findById(int id);
}

// User.java (simple data class)
public class User {
    private int id;
    private String name;
    // ...getters and setters
}
```

2. Establish the test class

```
 import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.Mockito.*;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
```

```
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;

@ExtendWith(MockitoExtension.class) // Enable Mockito annotations
public class UserServiceTest {

    @Mock // Create a mock of the UserRepository
    private UserRepository userRepository;

    @InjectMocks // Inject the mock into the UserService
    private UserService userService;

    @Test
    void testGetUserById_whenUserExists_shouldReturnUser() {
        // Arrange
        int userId = 1;
        User mockUser = new User();
        mockUser.setId(userId);
        mockUser.setName("John Doe");

        // Stubbing: Define the behavior of the mock
        when(userRepository.findById(userId)).thenReturn(mockUser);

        // Act
        User result = userService.getUserById(userId);

        // Assert
        assertEquals("John Doe", result.getName());

        // Verification: Ensure the mock was used as expected
        verify(userRepository, times(1)).findById(userId);
    }

    @Test
    void testGetUserById_whenUserDoesNotExist_shouldReturnNull() {
        // Arrange
        int userId = 2;
```

```
        // Stubbing: Define behavior for a non-existent user
        when(userRepository.findById(userId)).thenReturn(null);

        // Act
        User result = userService.getUserById(userId);

        // Assert
        assertEquals(null, result);

        // Verify that the repository method was called
        verify(userRepository).findById(userId);
    }
}
```

Key Insights

Mockito enables you to:

Isolate the class being tested by supplying mock versions of its dependencies.

Test situations that are challenging to replicate, such as network failures or database errors, by instructing mocks to raise exceptions.

Confirm that your code interacts appropriately with its dependencies by validating method invocations.

Compose clear and comprehensible tests using a fluent API and useful annotations.

Testing Arsenal: TestNG

TestNG (Test Next Generation) is a powerful and adaptable testing framework specifically designed for Java applications, aimed at simplifying and improving the processes involved in writing, managing, and executing tests. It overcomes various limitations present in earlier frameworks such as JUnit,

providing a broader array of functionalities across different testing categories, including unit, functional, integration, and end-to-end testing.

Key Features and Advantages of TestNG:

- Annotations: TestNG employs annotations (e.g., @Test, @BeforeSuite, @AfterClass) to specify test methods, configure setup and teardown processes, and control the flow of test execution.

- Test Grouping: Tests can be categorized into logical groups, enabling selective execution of particular test sets.

- Parallel Execution: TestNG allows for the concurrent execution of multiple tests, which significantly decreases the total time required for test execution.

- Prioritization and Dependencies: Tests can be assigned priorities to dictate their execution sequence, and dependencies can be established to ensure that certain tests are executed only after the successful completion of others.

- Parameterized Testing: The @Parameters annotation permits the passing of dynamic values to test methods, thereby facilitating testing with various data sets.

- Data-Driven Testing: The @DataProvider annotation enables the provision of multiple sets of test data to a single test method, making it efficient for scenarios that require diverse inputs.

- Customizable Test Suites: TestNG utilizes an XML file (testng.xml) to define and configure test suites, allowing for flexible management of tests, classes, and groups.

- Detailed Reporting: It automatically produces comprehensive HTML and XML reports following test execution, offering insights into the results of the tests.

- Integration: TestNG integrates smoothly with widely used tools such as Selenium, Maven, Gradle, and CI/CD tools like Jenkins, making it a versatile option for automated testing pipelines.

How to Utilize TestNG:

- Incorporate TestNG Dependency: Add the TestNG library to your project's build configuration (for instance, pom.xml if using Maven).

- Develop Test Classes: Construct Java classes that encapsulate your testing logic.

- Employ Annotations: Utilize TestNG annotations (such as @Test for test methods) to specify your tests and oversee their lifecycle.

- Set Up testng.xml (Optional but Advisable): Generate an XML file to outline test suites, include or exclude tests, and adjust various execution parameters.

- Execute Tests: Run your tests via your IDE, Maven/Gradle, or from the command line utilizing the testng.xml file.

Illustration of a fundamental TestNG test class:

```
import org.testng.annotations.Test;

public class MyFirstTestNGTest {

   @Test
   public void verifyAddition() {
      int a = 5;
      int b = 3;
      int sum = a + b;
      assert sum == 8 : "Addition failed: Expected 8, got " + sum;
   }

   @Test
   public void verifySubtraction() {
```

```
        int a = 10;
        int b = 4;
        int difference = a - b;
        assert difference == 6 : "Subtraction failed: Expected 6, got " + difference;
    }
}
```

4.4 Performance Monitors: JFR, VisualVM, Micrometer

Java Performance Monitors: JFR, VisualVM, and Micrometer

This document serves as a guide and provides sample code for utilizing Java Flight Recorder (JFR), VisualVM, and Micrometer to effectively monitor the performance of Java applications.

Java Flight Recorder (JFR)

JFR is an event-driven profiling tool integrated into the JVM, designed to operate with minimal overhead, making it ideal for use in production settings. It captures data regarding JVM, JDK, and application activities, which can subsequently be analyzed using a tool such as JDK Mission Control (JMC).

How to utilize JFR

1. Activate JFR at startup:

To initiate a JFR recording upon the launch of your application, include the following JVM arguments:

java-XX:StartFlightRecording=filename=my_recording.jfr,duration=60s MyApp.java

- -XX:StartFlightRecording: Activates and sets up a flight recording.
- filename=my_recording.jfr: Defines the output file for the recording.
- duration=60s: Establishes the recording duration as 60 seconds.

2. Manage an active JVM using jcmd:

You can initiate, halt, or create a dump of a recording on an active Java process by utilizing the jcmd utility, where <PID> represents the process ID.

1. Begin recording:

jcmd <PID> JFR.start name=myRecording duration=60s filename=my_recording.jfr

2. Verify status:

jcmd <PID> JFR.check

3. Create dump of recording:

jcmd <PID> JFR.dump name=myRecording filename=dumped_recording.jfr

4. Halt recording:

jcmd <PID> JFR.stop name=myRecording

Sample application for JFR

Here is a straightforward Java program that executes a memory-intensive operation.

```
// JFRDemo.java
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.TimeUnit;

public class JFRDemo {
    public static void main(String[] args) throws InterruptedException {
        System.out.println("Starting JFR Demo Application...");
        List<byte[]> memoryLeak = new ArrayList<>();

        // Simulate a task that allocates memory and runs for a while
```

```
        for (int i = 0; i < 100; i++) {
            System.out.println("Allocating memory: " + i);
            memoryLeak.add(new byte[1024 * 1024]); // Allocate 1MB
            TimeUnit.MILLISECONDS.sleep(100);
        }
        System.out.println("Memory allocation complete. Now idling.");

        // Let the application run so JFR can capture data
        TimeUnit.SECONDS.sleep(30);
        System.out.println("Exiting JFR Demo Application.");
    }
}
```

Executing and examining the JFR recording

1. Compile the Java source code: javac JFRDemo.java
2. Run with JFR activated

java-XX:StartFlightRecording=filename=jfr_recording.jfr,duration=30s JFRDemo

Analyze the output: Open jfr_recording.jfr in JDK Mission Control (JMC), which offers a graphical interface for viewing comprehensive metrics related to garbage collection, memory allocation, and thread activity.

VisualVM

VisualVM is a graphical tool that consolidates various command-line JDK utilities to deliver detailed, real-time monitoring of Java applications. It is applicable for both local and remote applications.

How to use VisualVM

Launch VisualVM:

VisualVM comes bundled with the JDK. Access your JDK's bin directory and execute jvisualvm.

Alternatively, you may download and install the standalone version.

Monitor a local application: VisualVM automatically identifies and displays all Java applications currently running locally in the "Applications" window located on the left side.

Connect to a remote application: You can monitor applications operating on remote servers by setting up JMX.

Profile an application: In the "Applications" window, double-click on a running application to access its monitoring tabs.

Memory/CPU Profiling: Click on the "Profiler" tab and select a task (e.g., "CPU" or "Memory") to initiate data collection.

Heap Dump: Right-click on the application and choose "Heap Dump" to obtain a snapshot of the application's memory utilization.

Example application for VisualVM

This example illustrates a multi-threaded application that may encounter a deadlock, which can be diagnosed with the assistance of VisualVM.

```
public class VisualVMDeadlock {
   public static void main(String[] args) {
      final Object resource1 = new Object();
      final Object resource2 = new Object();

      Thread t1 = new Thread(() -> {
         synchronized (resource1) {
            System.out.println("Thread 1: Holding lock on resource 1");
            try { Thread.sleep(100); } catch (Exception e) {}
            System.out.println("Thread 1: Waiting for lock on resource 2");
            synchronized (resource2) {
               System.out.println("Thread 1: Holding lock on resource 2");
            }
         }
```

```
        });

        Thread t2 = new Thread(() -> {
            synchronized (resource2) {
                System.out.println("Thread 2: Holding lock on resource 2");
                try { Thread.sleep(100); } catch (Exception e) {}
                System.out.println("Thread 2: Waiting for lock on resource 1");
                synchronized (resource1) {
                    System.out.println("Thread 2: Holding lock on resource 1");
                }
            }
        });

        t1.start();
        t2.start();
    }
}
```

Running and analyzing with VisualVM

Compile and execute VisualVMDeadlock.java.

- Launch VisualVM. The VisualVMDeadlock application will be listed under the "Local" applications section.
- Double-click on the application name to access its tabs.
- Diagnosing the deadlock: Select the "Threads" tab. You will observe the two threads (t1 and t2) and can utilize the "Thread Dump" button to create a snapshot. The thread dump will distinctly illustrate the deadlock, with each thread waiting for a lock that the other holds.

Micrometer

Micrometer serves as a vendor-neutral metrics facade for applications based on the JVM. It offers a straightforward, idiomatic API for recording metrics, which are subsequently published to a monitoring system such as Prometheus, Datadog, or InfluxDB.

How to utilize Micrometer with Spring Boot

In a Spring Boot application, Micrometer is automatically set up by Spring Boot Actuator. The essential steps include adding dependencies and employing the MeterRegistry to create and log metrics.

Maven dependencies

Incorporate the spring-boot-starter-actuator along with the registry dependency for the chosen monitoring system (for instance, micrometer-registry-prometheus).

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
   <groupId>io.micrometer</groupId>
   <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

Example application utilizing Micrometer

This is a straightforward Spring Boot REST controller that employs Micrometer to monitor requests.

```
// MicrometerDemoController.java
import io.micrometer.core.instrument.Counter;
import io.micrometer.core.instrument.MeterRegistry;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class MicrometerDemoController {

   private final Counter requestCounter;

   public MicrometerDemoController(MeterRegistry registry) {
```

```
        this.requestCounter = registry.counter("demo.requests.processed", "type",
"api");
    }

    @GetMapping("/api/demo")
    public String demoEndpoint() {
        requestCounter.increment();
        return "Micrometer demo endpoint accessed!";
    }
}
```

Executing and observing Micrometer metrics

- Set up application.properties: Activate the Prometheus endpoint to reveal metrics.

  management.endpoints.web.exposure.include=prometheus

- Launch the application: Utilize mvn spring-boot:run to initiate the Spring Boot application.

- Reach the endpoint: Visit http://localhost:8080/api/demo multiple times.

- Examine metrics: Navigate to http://localhost:8080/actuator/prometheus to view the exposed metrics. You will discover a demo_requests_processed_total metric that has been increased.

4.5 Security & Code Quality: SpotBugs, SonarQube, OWASP Tools

Ensuring strong security and excellent code quality in Java applications is essential.

Tools such as SpotBugs, SonarQube, and several OWASP tools assist in this endeavor through static analysis and vulnerability identification.

1. SpotBugs:

SpotBugs is a static analysis tool designed to identify bug patterns within Java bytecode. It uncovers frequent programming mistakes, possible performance concerns, and security weaknesses.

Sample Code and SpotBugs Integration (Maven):

Incorporate the SpotBugs Maven plugin into your pom.xml:

```
<build>
  <plugins>
    <plugin>
      <groupId>com.github.spotbugs</groupId>
      <artifactId>spotbugs-maven-plugin</artifactId>
      <version>4.8.3</version>
      <configuration>
        <effort>Max</effort>
        <threshold>Low</threshold>
        <failOnError>true</failOnError>
        <includeFilterFile>spotbugs-include.xml</includeFilterFile>
      </configuration>
      <executions>
        <execution>
          <goals>
            <goal>check</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Execute SpotBugs using the command mvn spotbugs:check

2. SonarQube:

SonarQube is a free and open-source platform designed for the ongoing evaluation of code quality and security. It conducts static analysis to identify bugs, code smells, and security vulnerabilities in multiple programming languages, including Java.

Sample Code and SonarQube Integration (Maven):

Set up the SonarQube plugin in your pom.xml:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.sonarsource.scanner.maven</groupId>
      <artifactId>sonar-maven-plugin</artifactId>
      <version>3.10.0.2594</version>
    </plugin>
  </plugins>
</build>
```

Evaluate your project using SonarQube by executing mvn sonar:sonar. This necessitates an operational SonarQube server.

3. OWASP Tools:

The OWASP (Open Web Application Security Project) Foundation offers a variety of tools and resources aimed at improving application security. Notable tools for Java comprise:

- OWASP Dependency-Check: Inspects project dependencies for recognized vulnerabilities.
- OWASP ZAP (Zed Attack Proxy): A comprehensive penetration testing tool designed to identify vulnerabilities in web applications.

Sample Code and Integration of OWASP Dependency-Check (Maven):

Incorporate the Dependency-Check Maven plugin into your pom.xml:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.owasp</groupId>
      <artifactId>dependency-check-maven</artifactId>
      <version>9.2.0</version>
      <executions>
        <execution>
          <goals>
            <goal>check</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Execute the dependency verification using mvn dependency-check:check.

Illustration of a Vulnerable Java Code Snippet (for detection demonstration):

```
import java.io.IOException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class VulnerableApp {

  public static void main(String[] args) throws IOException, SQLException {
    // Insecure password storage (hardcoded)
    String password = "mysecretpassword";

    // SQL Injection vulnerability
    String userInput = "admin' OR '1'='1"; // User input
        String query = "SELECT * FROM users WHERE username = '" +
userInput + "' AND password = '" + password + "'";
```

```
                                        try (Connection conn = DriverManager.getConnection("jdbc:h2:mem:testdb", "sa", "");
            Statement stmt = conn.createStatement()) {
                stmt.execute("CREATE TABLE users (username VARCHAR(255), password VARCHAR(255))");
                            stmt.execute("INSERT INTO users VALUES ('admin', 'mysecretpassword')");

            // This query is vulnerable to SQL injection
            stmt.executeQuery(query);
            System.out.println("Query executed (potentially vulnerable).");
        }
    }
}
```

Utilizing static analysis tools such as SpotBugs and SonarQube on this code would highlight the hardcoded password and the SQL injection vulnerability. OWASP Dependency-Check would detect vulnerabilities present in any outdated libraries utilized in the project.

4.6 AI Assistants: GitHub Copilot & CodeWhisperer

AI coding assistants such as GitHub Copilot and Amazon CodeWhisperer utilize large language models to offer code suggestions and expedite development. Both tools are designed to improve developer productivity by generating code, yet they possess unique features and areas of emphasis.

GitHub Copilot

GitHub Copilot, which is powered by OpenAI's Codex, serves as a versatile AI coding assistant. It provides real-time code suggestions, completions, and even entire functions or classes based on the context of your code and comments. It accommodates a broad spectrum of programming

languages, including Java, and seamlessly integrates with popular IDEs such as VS Code and JetBrains products.

Sample Java Code with Copilot:

```
// Create a method to calculate the factorial of a number
public class FactorialCalculator {

    public static long calculateFactorial(int n) {
        // Copilot will likely suggest the following implementation
        if (n < 0) {
                        throw new IllegalArgumentException("Number must be non-negative.");
        }
        long result = 1;
        for (int i = 1; i <= n; i++) {
            result *= i;
        }
        return result;
    }

    public static void main(String[] args) {
        int number = 5;
        // Copilot can suggest printing the result
                    System.out.println("Factorial of " + number + " is: " + calculateFactorial(number));
    }
}
```

Amazon CodeWhisperer

Amazon CodeWhisperer is an AI coding companion that concentrates on delivering recommendations specifically designed for AWS services. It produces code suggestions for AWS APIs, infrastructure as code (utilizing AWS CDK), and integrates with AWS development tools like Cloud9 and SageMaker Studio, in addition to popular IDEs. CodeWhisperer also features a security scan capability to detect potential vulnerabilities in the generated code.

Sample Java Code with CodeWhisperer (AWS S3 example):

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.s3.S3Client;
import software.amazon.awssdk.services.s3.model.CreateBucketRequest;
import software.amazon.awssdk.services.s3.model.S3Exception;

public class S3BucketCreator {

    public static void createS3Bucket(String bucketName, Region region) {
        // CodeWhisperer can suggest the following S3 client and bucket creation
logic
        S3Client s3Client = S3Client.builder()
                .region(region)
                .build();

        try {
                                        CreateBucketRequest    createBucketRequest    =
CreateBucketRequest.builder()
                    .bucket(bucketName)
                    .build();
            s3Client.createBucket(createBucketRequest);
            System.out.println("Bucket " + bucketName + " created successfully.");
        } catch (S3Exception e) {
            System.err.println("Error creating bucket: " + e.getMessage());
        } finally {
            s3Client.close();
        }
    }

    public static void main(String[] args) {
        String bucketName = "my-unique-java-bucket-12345";
        Region region = Region.US_EAST_1;
        // CodeWhisperer can suggest calling the bucket creation method
        createS3Bucket(bucketName, region);
    }
}
```

PRACTISE SET

MULTIPLE CHOICES QUESTION

1. Which of the following best describes IntelliJ IDEA compared to VS Code?

A. IntelliJ is a lightweight text editor, while VS Code is a full-fledged IDE

B. IntelliJ is a full-featured IDE designed primarily for Java, while VS Code is a general-purpose code editor with extensions

C. Both are equally optimized for Java development without additional plugins

D. VS Code provides more advanced refactoring tools than IntelliJ

2. What is Maven primarily used for in Java projects?

A. Managing user interfaces

B. Automating builds and managing project dependencies

C. Running unit tests only

D. Monitoring application performance

3. Which build tool uses a domain-specific language (DSL) based on Groovy or Kotlin?

A. Maven

B. Gradle

C. Ant

D. JBang

4. What is JBang mainly used for?

A. Compiling C++ applications

B. Writing Java one-liners and scripts without full project setup

C. Managing Java dependencies for large-scale enterprise projects

D. Profiling Java applications

5. Mockito is primarily used for:

A. Integration testing

B. Mocking dependencies in unit tests

C. Load testing

D. Code coverage analysis

6. Which of the following statements about TestNG is TRUE?

A. TestNG can only be used with JUnit
B. TestNG supports parallel test execution and flexible test configuration
C. TestNG is a build automation tool
D. TestNG is deprecated in modern Java testing

7. Which Java performance tool is built into the JVM and records profiling data for later analysis?

A. VisualVM
B. Micrometer
C. Java Flight Recorder (JFR)
D. JConsole

8. Micrometer is mainly used for:

A. Code compilation
B. Performance monitoring and metrics collection across applications
C. Security auditing
D. Static code analysis

9. SpotBugs is best described as:

A. A tool for detecting runtime performance issues
B. A static analysis tool for finding potential bugs in Java code
C. A dependency management tool
D. A performance monitoring agent

10. Which AI coding assistant is developed by Amazon?

A. GitHub Copilot
B. CodeWhisperer
C. TabNine
D. IntelliCode

BRACE YOURSELF

1.Explain the key differences between IntelliJ IDEA and VS Code in terms of features, performance, Java integration, and ecosystem support.

2.Discuss how Maven and Gradle handle project dependencies and builds differently.

3.Describe what JBang is and how it simplifies running Java programs compared to traditional compilation and packaging processes.

4.Explain how Mockito enables unit testing of Java applications through mocking.

5.Compare TestNG and JUnit in terms of features such as parameterization, parallel testing, and test configuration.

6.Describe the purpose and workflow of Java Flight Recorder (JFR).

7.What role does Micrometer play in application performance monitoring?

8.Discuss how SpotBugs and SonarQube contribute to improving code quality.

9.What are OWASP Tools, and how do they assist developers in identifying and mitigating security vulnerabilities in Java applications?

10.Compare GitHub Copilot and Amazon CodeWhisperer in terms of functionality, integration with IDEs, and privacy/security considerations.

Chapter 5 CommandDesign Pattern

5.1 Introduction

The Command Design Pattern is classified as a behavioral design pattern, which facilitates the decoupling of the invoker from the request's receiver.
To comprehend the Command Design Pattern, let us consider an example that demonstrates the execution of various types of jobs. A job can represent any task within a system, such as sending emails, SMS messages, logging activities, or performing input/output functions.

The Command Pattern serves to separate the invoker from the receiver, enabling the execution of any job type without requiring knowledge of its implementation details. To enhance this example, we will introduce threads that allow for the concurrent execution of these jobs.

Since these jobs operate independently, the order in which they are executed is not particularly significant. We will establish a thread pool to restrict the number of threads available for job execution. A command object will encapsulate the jobs and assign them to a thread from the pool for execution.

Before we proceed with the implementation of this example, let us delve deeper into the Command Design Pattern.

5.2 What is the Command Design Pattern

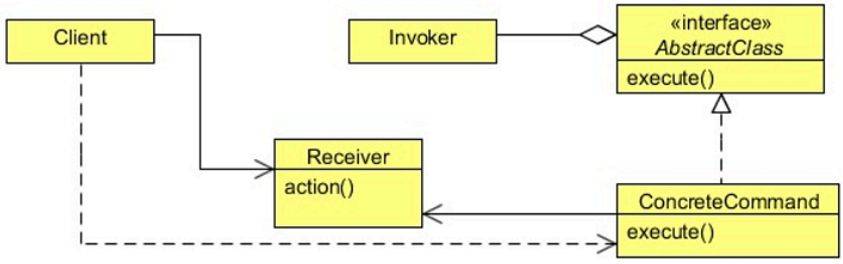
The purpose of the Command Design Pattern is to encapsulate a request within an object, allowing developers to parameterize clients with various requests, queue or log these requests, and facilitate undoable operations.

Generally, an object-oriented application comprises a collection of interacting objects, each providing specific, focused functionalities. In response to user interactions, the application performs a certain type of processing. To achieve this, the application utilizes the services of different objects to meet its processing needs.

From an implementation perspective, the application may rely on a specific object that calls methods on these objects by supplying the necessary data as arguments. This specific object is known as an invoker, as it triggers operations on various objects. The invoker can be considered a component of the client application. The group of objects that actually implement the services required for processing requests is referred to as Receiver objects.

By employing the Command pattern, the invoker that issues a request on behalf of the client can be decoupled from the set of service-rendering Receiver objects. The Command pattern advocates for the creation of an abstraction for the processing to be executed or the action to be performed in response to client requests. This abstraction is designed to declare a common interface that different concrete implementers, known as Command objects, must implement. Each Command object corresponds to a distinct type of client request and its associated processing.

A specific Command object is tasked with providing the functionality necessary to process the request it represents, yet it does not include the actual implementation of that functionality. Command objects utilize Receiver objects to deliver this functionality.



Fig(5.2 Command Design Pattern)

Command
• Establishes an interface for performing an operation.

ConcreteCommand
• Specifies a connection between a Receiver object and a specific action.
• Implements Execute by calling the relevant operation(s) on the Receiver.

Client
• Instantiates a ConcreteCommand object and assigns its receiver.
Invoker
• Requests the command to execute the task.

Receiver
• Understands how to execute the operations related to fulfilling a request. Any class can function as a Receiver.

5.3 Implementing the Command Design Pattern

We will execute the example utilizing a command object. This command object will be associated with a common interface and will include a method designated for executing the requests. The specific command classes will override this method and will provide their unique implementation to carry out the request.

```
package com.javacodegeeks.patterns.commandpattern;
public interface Job {
public void run();
}
```

The Job interface serves as the command interface, containing a single method, run, which is executed by a thread. The execute method of our command corresponds to the run method, which will be employed by a thread to accomplish the task at hand.

Various types of jobs can be executed. Below are the distinct concrete classes whose instances will be executed by the different command objects.

```
packagecom.javacodegeeks.patterns.commandpattern;
publicclass Email{
publicvoidsendEmail(){
System.out.println("Sendingemail.....");
```

```
}
}


packagecom.javacodegeeks.patterns.commandpattern;
publicclass FileIO{
publicvoidexecute(){
System.out.println("ExecutingFileIOoperations...");
}
}


packagecom.javacodegeeks.patterns.commandpattern;
publicclass Logging{
publicvoidlog(){
System.out.println("Logging...");
}
}

packagecom.javacodegeeks.patterns.commandpattern;
publicclass Sms{
publicvoidsendSms(){
System.out.println("SendingSMS...");
}
}
```

The following are the various command classes that encapsulate the aforementioned classes and implement the Job interface.

```
packagecom.javacodegeeks.patterns.commandpattern;
publicclass EmailJobimplementsJob{
privateEmailemail;
publicvoidsetEmail(Emailemail){
this.email=email;
}
@Override
publicvoidrun(){
System.out.println("JobID:"+Thread.currentThread().getId()+"executing
```

```
 emailjobs.");
 if(email!=null){
 email.sendEmail();
 }
 try{
 Thread.sleep(1000);
 }catch(InterruptedExceptione){
 Thread.currentThread().interrupt();
 }
 }
 }

packagecom.javacodegeeks.patterns.commandpattern;
 publicclass FileIOJobimplementsJob{
 privateFileIOfileIO;
 publicvoidsetFileIO(FileIO fileIO){
 this.fileIO=fileIO;
 }
 @Override
 publicvoidrun(){
 System.out.println("JobID:"+Thread.currentThread().getId()+"executing
 fileIOjobs.");
 if(fileIO!=null){
 fileIO.execute();
 }
 try{
 Thread.sleep(1000);
 }catch(InterruptedExceptione){
 Thread.currentThread().interrupt();
 }
 }
 }

packagecom.javacodegeeks.patterns.commandpattern;
 publicclass LoggingJobimplementsJob{
 privateLogginglogging;
 publicvoidsetLogging(Logging logging){
 this.logging=logging;
```

```
}
@Override
publicvoidrun(){
System.out.println("JobID:"+Thread.currentThread().getId()+"executing
loggingjobs.");
if(logging!=null){
logging.log();
}
try{
Thread.sleep(1000);
}catch(InterruptedExceptione){
Thread.currentThread().interrupt();
}
}
}

packagecom.javacodegeeks.patterns.commandpattern;

publicclass SmsJobimplementsJob{
privateSmssms;
publicvoidsetSms(Smssms){
this.sms=sms;
}
@Override
publicvoidrun(){
System.out.println("JobID:"+Thread.currentThread().getId()+"executing
smsjobs.");
if(sms!=null){
sms.sendSms();
}
try{
Thread.sleep(1000);
}catch(InterruptedExceptione){
Thread.currentThread().interrupt();
}
}
}
```

The above classes maintain a reference to their respective classes that will be utilized to complete the job. The classes override the run method and perform the requested work. For instance, the SmsJob class is responsible for sending SMS; its run method invokes the sendSms method of the Sms object to ensure the job is completed.

You can assign different objects one by one to the same command object. The following is the ThreadPool class, which is used to create a pool of threads and allows a thread to retrieve and execute the job from the job queue.

```
packagecom.javacodegeeks.patterns.commandpattern;
importjava.util.concurrent.BlockingQueue;
importjava.util.concurrent.LinkedBlockingQueue;
publicclass ThreadPool{
privatefinalBlockingQueue<Job>jobQueue;
privatefinalThread[]jobThreads;
privatevolatilebooleanshutdown;
publicThreadPool(intn)
{
jobQueue=newLinkedBlockingQueue<>();
jobThreads=newThread[n];
for(inti=0;i<n; i++){
jobThreads[i]=newWorker("PoolThread"+i);
jobThreads[i].start();
}
}
publicvoidaddJob(Jobr)
{
try{
jobQueue.put(r);
}catch(InterruptedExceptione){
Thread.currentThread().interrupt();
}
}
publicvoidshutdownPool()
{
```

```
while(!jobQueue.isEmpty()){
try{
Thread.sleep(1000);
}catch(InterruptedExceptione){
e.printStackTrace();
}
}
shutdown=true;
for(ThreadworkerThread:jobThreads){
workerThread.interrupt();
}
}
privateclassWorkerextends Thread
{
publicWorker(String name)
{
super(name);
}
publicvoidrun()
{
while(!shutdown){
try{
Jobr=jobQueue.take();
r.run();
}catch(InterruptedExceptione){
}
}
}
}
}
```

The aforementioned class is utilized for the creation of threads (worker threads). Each worker thread will await a job in a queue, subsequently executing the job before returning to a waiting state. The class encompasses a job queue; when a new job is added to the queue, a worker thread from the pool will execute the job.

Additionally, we have incorporated a shutdownPool method, which is employed to terminate the pool by interrupting all worker threads only when the job queue is empty. The addJob method is responsible for adding jobs to the queues.

```
packagecom.javacodegeeks.patterns.commandpattern;
 publicclass TestCommandPattern{
 publicstaticvoidmain(String[]args)
 {
 init();
 }
 privatestaticvoidinit()
 {
 ThreadPoolpool=newThreadPool(10);
 Emailemail=null;
 EmailJob emailJob=newEmailJob();
 Smssms=null;
 SmsJobsmsJob=newSmsJob();
 FileIOfileIO=null;;
 FileIOJobfileIOJob=newFileIOJob();
 Logginglogging=null;
 LoggingJoblogJob=newLoggingJob();
 for(inti=0;i<5;i++){
 email=newEmail();
 emailJob.setEmail(email);
 sms=newSms();
 smsJob.setSms(sms);
 fileIO=newFileIO();
 fileIOJob.setFileIO(fileIO);
 logging=newLogging();
 logJob.setLogging(logging);
 pool.addJob(emailJob);
 pool.addJob(smsJob);
 pool.addJob(fileIOJob);
 pool.addJob(logJob);
 }
 pool.shutdownPool();
 }
 }
```

Now, let us proceed to test the code.

```
JobID:9executingemailjobs.
 Sendingemail.......
 JobID:12executingloggingjobs.
 JobID:17executingemailjobs.
 Sendingemail.......
 JobID:13executingemailjobs.
 Sendingemail.......
 JobID:10executingsmsjobs.
 SendingSMS...
 JobID:11executingfileIOjobs.
 ExecutingFile IOoperations...
 JobID:18executingsmsjobs.
 SendingSMS...
 Logging...
 JobID:16executingloggingjobs.
 Logging...
 JobID:15executingfileIOjobs.
 ExecutingFile IOoperations...
 JobID:14executingsmsjobs.
 SendingSMS...
 JobID:12executingfileIOjobs.
 ExecutingFile IOoperations...
 JobID:10executingloggingjobs.
 Logging...
 JobID:18executingemailjobs
Sending email.......
 Job ID: 16 executing sms jobs.
 Sending SMS...
 Job ID: 14 executing fileIO jobs.
 Executing File IO operations...
 Job ID: 9 executing logging jobs.
 Logging...
 Job ID: 17 executing email jobs.
 Sending email.......
 Job ID: 13 executing sms jobs.
 Sending SMS...
```

```
Job ID: 15 executing fileIO jobs.
Executing File IO operations...
Job ID: 11 executing logging jobs.
Logging...
```

In the above class, we established a thread pool consisting of 10 threads. Subsequently, we assigned various command objects with distinct jobs and added these jobs to the queue utilizing the addJob method of the ThreadPool class. Once a job is inserted into the queue, a thread executes the job and removes it from the queue.

We have defined various types of jobs; however, by employing the Command Design Pattern, we decouple the job from the invoking thread. The thread is capable of executing any object that implements the Job interface. The different command objects encapsulate the various objects and perform the requested operations on these objects.

The output illustrates the different threads executing the various jobs. By observing the job ID in the output, it is evident that a single thread is executing multiple jobs. This occurs because, after executing a job, the thread returns to the pool.

The benefit of the Command Design Pattern is that it allows for the addition of more diverse types of jobs without necessitating changes to the existing classes. This results in increased flexibility, maintainability, and a reduction in the likelihood of bugs within the code.

5.4 When to use the Command Design Pattern

Utilize the Command pattern when you wish to:

• Parameterize objects based on an action to execute.

• Specify, queue, and carry out requests at various times. A Command object can exist independently of the original request. If the recipient of a request can be represented in a manner that is independent of the address space, you can

transfer a command object for the request to another process and fulfill the request there.

• Facilitate undo functionality. The Execute operation of the Command can retain state for reversing its effects within the command itself. The Command interface must include an additional Un-execute operation that negates the effects of a prior call to Execute.

Executed commands are maintained in a history list. Unlimited levels of undo and redo can be achieved by navigating this list in both directions, invoking Un-execute and Execute, respectively.

• Enable logging of changes to allow for reapplication in the event of a system failure. By enhancing the Command interface with load and store operations, you can maintain a persistent log of changes. Recovery from a crash entails reloading logged commands from disk and re-executing them using the Execute operation.

• Organize a system around high-level operations constructed from primitive operations. This type of structure is prevalent in information systems that support transactions. A transaction encapsulates a series of modifications to data. The Command pattern provides a method to model transactions. Commands share a common interface, allowing you to invoke all transactions uniformly. Additionally, the pattern simplifies the process of extending the system with new transactions.

PRACTISE SET

MULTIPLE CHOICES QUESTION

1. What is the primary purpose of the Command Design Pattern?
A. To encapsulate a request as an object, allowing parameterization and queuing of requests
B. To simplify complex conditional statements
C. To manage multiple threads efficiently
D. To enforce a strict class hierarchy

2. Which of the following best describes a "Command" in the Command Design Pattern?
A. A concrete implementation of an algorithm
B. An object that encapsulates all information needed to perform an action or trigger an event
C. A database transaction manager
D. A UI component responsible for event handling

3. In the Command pattern, the Invoker is responsible for:
A. Storing and executing commands
B. Creating new command objects
C. Handling system errors
D. Logging user activity

4. Which of the following is a key advantage of using the Command Design Pattern?
A. Reduces code readability
B. Increases coupling between sender and receiver
C. Enables undo/redo functionality easily
D. Makes class design rigid and less flexible

5. What role does the Receiver play in the Command pattern?
A. It defines the interface for executing a command
B. It carries out the actual work when the command is executed
C. It initiates the request
D. It logs the command history

6. Which of the following is NOT a benefit of the Command pattern?

A. Simplifies adding new commands without changing existing code

B. Provides a way to queue and schedule commands

C. Allows complex command chains

D. Forces commands to execute immediately without scheduling

7. In Java, which of the following could best represent a Command interface?

A. interface Command { void execute(); }

B. interface Command { void runTask(); }

C. interface Command { void undo(); }

D. interface Command { void execute(String arg); }

8. When should you consider using the Command Design Pattern?

A. When you need to separate objects that issue requests from those that process them

B. When your application has no user interactions

C. When commands do not require history or logging

D. When using procedural programming only

9. Which of the following real-world examples best represents the Command Design Pattern?

A. A remote control that can turn devices on or off

B. A loop in a for-each statement

C. A static helper class for math operations

D. A recursive function call

10. Which of these statements about the Command Design Pattern is TRUE?

A. The pattern tightly couples the sender and receiver

B. It is used only for GUI applications

C. It can support undo and redo operations by storing command history

D. It cannot be combined with other patterns

BRACE YOURSELF

1. Explain the main idea behind the Command Design Pattern.

2.Define the Command Design Pattern and describe its core components.

3.Describe the step-by-step process of implementing the Command Design Pattern in Java (or any OOP language).

4.What is the significance of the Command interface in this pattern?

5.Discuss how the Command pattern decouples the object that issues a request from the one that knows how to perform it.

6.Explain how the Command Design Pattern can be extended to support undo/redo operations.

7.Under what circumstances should a software developer consider using the Command Design Pattern?

8.Compare the Command Design Pattern with the Strategy Design Pattern.

9.List and explain the main advantages and possible drawbacks of using the Command Design Pattern.

10.Describe a real-world scenario or software system that effectively uses the Command Design Pattern.

Chapter 6: The Memento Design Pattern

6.1 Introduction

At times, it is essential to document the internal state of an object. This documentation is necessary when establishing checkpoints and 'undo' mechanisms that allow users to retract tentative actions or recover from mistakes. It is imperative to store state information in a designated location, enabling the restoration of objects to their prior conditions. However, objects typically encapsulate some or all of their state, rendering it inaccessible to other objects and preventing external saving. Revealing this state would breach encapsulation, potentially jeopardizing the application's reliability and extensibility.

The Memento pattern serves as a solution to this issue without disclosing the internal structure of the object. The object whose state is to be captured is known as the originator.

To demonstrate the application of the Memento Pattern, let us consider an example. We will develop a class that contains two double-type fields and perform various mathematical operations on it. Users will be provided with the undo functionality. If the outcomes of certain operations do not meet a user's satisfaction, the user can invoke the undo operation, which will revert the object's state to the last saved point.

The example will also feature a save point mechanism that allows the user to preserve the object's state. Additionally, we will offer a range of undo operations. A basic undo will revert the object's state to the previous save point. An undo with a specified save point will restore that particular state of the object, while an undo all operation will erase all saved states of the object and revert it to its initialized state, as it was upon creation.

Before we proceed with the implementation of the pattern, let us delve deeper into the Memento Design Pattern.

6.2 What is the Memento Design Pattern

The intent of the Memento Pattern is to capture and externalize the internal state of an object without compromising encapsulation, allowing the object to be restored to this state at a later time.

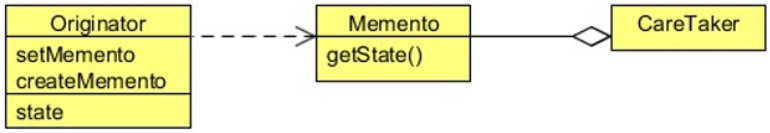


Figure 6.2: screenshot

• It stores the internal state of the Originator object. The memento can retain as much or as little of the originator's internal state as deemed necessary by the originator itself.

• It safeguards against access by entities other than the originator. Mementos effectively has two interfaces. The Caretaker has a limited interface to the Memento, allowing it only to pass the memento to other objects. In contrast, the Originator has access to a comprehensive interface that enables it to retrieve all the data required to revert to its previous state. Ideally, only the originator that created the memento should be allowed to access the internal state of the memento.

Originator
• It creates a memento that encapsulates a snapshot of its current internal state.
• It utilizes the memento to restore its internal state.

Caretaker
• It is tasked with the safekeeping of the memento.
• It does not operate on or inspect the contents of a memento.

When a client wishes to save the state of the originator, it requests the current state from the originator. The originator then stores all attributes necessary for restoring its state in a separate object known as a Memento and returns it to the client. Therefore, a Memento can be regarded as an object that encapsulates the internal state of another object at a specific moment in time. A Memento object must conceal the variable values of the originator from all objects except the originator itself. In essence, it should safeguard its internal state from access by entities other than the originator. To achieve this, a

Memento should be structured to offer restricted access to other objects while permitting the originator to access its internal state.

6.3 Implementing the Memento Design Pattern

```
package com.javacodegeeks.patterns.mementopattern;
public class Originator {
private double x;
private double y;
private String lastUndoSavepoint;
CareTaker careTaker;
public Originator(double x, double y,CareTaker careTaker){
this.x = x;
this.y = y;
this.careTaker = careTaker;
createSavepoint("INITIAL");
}
public double getX(){
return x;
}
public double getY(){
return y;
}
publicvoidsetX(doublex){
this.x=x;
}
publicvoidsetY(doubley){
this.y=y;
}
publicvoidcreateSavepoint(StringsavepointName){
careTaker.saveMemento(newMemento(this.x,this.y),savepointName);
lastUndoSavepoint=savepointName;
}
publicvoidundo(){
setOriginatorState(lastUndoSavepoint);
}
publicvoidundo(StringsavepointName){
```

```
setOriginatorState(savepointName);
}
publicvoidundoAll(){
setOriginatorState("INITIAL");
careTaker.clearSavepoints();
}
privatevoidsetOriginatorState(StringsavepointName){
Mementomem=careTaker.getMemento(savepointName);
this.x=mem.getX();
this.y=mem.getY();
}
@Override
publicStringtoString(){
return"X:"+x+",Y: "+y;
}
}
```

The above is the Originator class whose object state should be preserved in a memento. The class contains two double type fields, x and y, and also takes a reference to a CareTaker. The CareTaker is utilized to save and retrieve the memento objects that represent the state of the Originator object.

In the constructor, we have saved the initial state of the object using the createSavepoint method. This method generates a memento object and requests the caretaker to manage the object. We have employed a lastUndoSavepoint variable, which is used to store the key name of the last saved memento in order to facilitate the undo operation.

The class offers three types of undo operations. The undo method without any parameters restores the last saved state, while the undo method with the savepoint name as a parameter restores the state saved with that specific savepoint name. The undoAll method instructs the caretaker to clear all the savepoints and reset it to the initial state (the state at the time of the object's creation).

```
packagecom.javacodegeeks.patterns.mementopattern;
publicclass Memento{
privatedoublex;
```

```
privatedoubley;
publicMemento(doublex,doubley){
this.x=x;
this.y=y;
}
publicdoublegetX(){
returnx;
}
publicdoublegetY(){
returny;
}
}
```

The Memento class is utilized to preserve the state of the Originator and is maintained by the caretaker. The class does not include any setter methods; it is solely employed to retrieve the state of the object.

```
packagecom.javacodegeeks.patterns.mementopattern;
importjava.util.HashMap;
importjava.util.Map;
publicclass CareTaker{

privatefinalMap<String,Memento>savepointStorage=newHashMap<String,Me
mento>()
;
publicvoidsaveMemento(Mementomemento,StringsavepointName){
System.out.println("Savingstate..."+savepointName);
savepointStorage.put(savepointName,memento);
}
publicMementogetMemento(StringsavepointName){
System.out.println("Undoat..."+savepointName);
returnsavepointStorage.get(savepointName);
}
publicvoidclearSavepoints(){
System.out.println("Clearingallsavepoints...");
savepointStorage.clear();
}
}
```

The above class is the caretaker class utilized for storing and providing the requested memento object. The class contains the saveMemento method, which is employed to save the memento object, the getMemento method, which is used to retrieve the requested memento object, and the clearSavepoints method, which serves to eliminate all save points and deletes all saved memento objects.

Now, let us test the example.

```
packagecom.javacodegeeks.patterns.mementopattern;
publicclass TestMementoPattern
{
publicstaticvoidmain(String[]args)
{
CareTakercareTaker=newCareTaker();
Originatororiginator=newOriginator(5,10,careTaker);
System.out.println("DefaultState:"+originator);
originator.setX(originator.getY()*51);
System.out.println("State:"+originator);
originator.createSavepoint("SAVE1");
originator.setY(originator.getX()/22);
System.out.println("State:"+originator);
originator.undo();
System.out.println("Stateafterundo:"+originator);
originator.setX(Math.pow(originator.getX(),3));
originator.createSavepoint("SAVE2");
System.out.println("State:"+originator);
originator.setY(originator.getX()-30);
originator.createSavepoint("SAVE3");
System.out.println("State:"+originator);
originator.setY(originator.getX()/22);
originator.createSavepoint("SAVE4");
System.out.println("State:"+originator);
originator.undo("SAVE2");
System.out.println("Retrievingat:"+originator);
originator.undoAll();
System.out.println("Stateafterundoall:"+originator);
}
```

```
}
```

The code provided above will yield the following output.

```
Savingstate...INITIAL
DefaultState: X:5.0,Y:10.0
State:X:510.0,Y:10.0
Savingstate...SAVE1
State:X:510.0,Y:23.181818181818183
Undoat...SAVE1
Stateafterundo:X:510.0,Y:10.0
Savingstate...SAVE2
State:X:1.32651E8,Y:10.0
Savingstate...SAVE3
State:X:1.32651E8,Y:1.3265097E8
Savingstate...SAVE4
State:X:1.32651E8,Y:6029590.909090909
Undoat...SAVE2
Retrievingat: X:1.32651E8,Y:10.0
Undoat...INITIAL
Clearingall savepoints...
Stateafterundoall:X:5.0,Y:10.0
```

In the code above, we have instantiated a CareTaker object and subsequently assigned it to an Originator object. We then set the values of x and y to 5 and 10, respectively. Following this, we performed some operations on x and saved the state of the object as "SAVE1".

After executing additional operations, we invoked the undo method to revert to the last state of the object, which is clearly demonstrated in the output. We then conducted further operations and saved the states of the object as "SAVE2, SAVE3, and SAVE4".

Subsequently, we instructed the Originator to restore the SAVE2 state and called the undoAll method, which reset the object to its initial state and removed all save points.

It is important to note that in the example above, the Originator is tasked with providing its memento to the caretaker. This is because we do not wish to assign this responsibility to the user. In our example, the user should only need to request the save point and the restoration of the object's state. In many instances, a caretaker operates independently of the originator through another class (as illustrated in the class diagram above).

6.4 Creational Patterns: Singleton, Factory, Builder

Singleton

The Singleton pattern is a creational design pattern that ensures a class has a single instance and offers a global access point to that instance. It is particularly useful for resources that need to be uniquely available throughout an application, such as logging systems, database connections, and configuration managers. The Singleton pattern guarantees that a class maintains only one instance while providing a global access point to it. This is beneficial for resources like configuration managers, loggers, or database connection pools, where a single instance is sufficient across the application.

Key components

- The implementation of the Singleton pattern generally requires a private constructor, a private static instance, and a public static method to retrieve the instance.
- Sample Java code: Bill Pugh Singleton
- The Bill Pugh method for implementing a singleton employs a static inner helper class to ensure thread safety without the need for synchronization. This approach is effective because the inner class is loaded only when getInstance() is invoked for the first time, and the JVM guarantees that a class is initialized only once, thus providing inherent thread safety and lazy initialization. The sample Java code and its output can be found in the referenced document.

Advantages of the Singleton pattern

The advantages include resource control, memory efficiency due to the creation of a single object, and the possibility of lazy initialization.

Disadvantages of the Singleton pattern

The potential drawbacks include challenges in testing, hidden dependencies, and a decrease in code modularity.

```
public class Singleton {
    // Private static instance of the class
    private static Singleton instance;

    // Private constructor to prevent direct instantiation
    private Singleton() {
        // Initialization logic if needed
    }

    // Public static method to provide global access to the instance
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton(); // Lazy instantiation
        }
        return instance;
    }

    public void showMessage() {
        System.out.println("Hello from the Singleton instance!");
    }
}
```

Factory

The Factory Method pattern establishes an interface for object creation while allowing subclasses to determine which class to instantiate. This approach defers the instantiation process to the subclasses, thereby fostering loose coupling.

The Factory pattern is categorized as a creational design pattern that facilitates the creation of objects without the necessity of specifying the exact class to be instantiated. It consolidates object creation within a single "factory" class, thereby alleviating the burden of object instantiation from the client code.
This pattern encourages loose coupling, enhancing the flexibility, reusability, and extensibility of the code. Rather than directly employing the new keyword, the client code solicits an object from the factory, which subsequently provides an instance of a concrete class that adheres to a shared interface.

Key components of the Factory pattern

- Product: An interface or abstract class that delineates the common methods for the objects intended for creation.
- Concrete Products: The specific classes that fulfill the Product interface. These represent the objects produced by the factory.
- Factory: A class that encompasses the factory method for generating and returning instances of the Product.
- Client: The code that utilizes the factory to obtain a product instance without requiring knowledge of the specific concrete product class.

Example: Vehicle Manufacturing

Consider an application that requires the creation of various types of vehicles. In the future, it may be necessary to introduce a new vehicle type. The Factory pattern can be employed to manage this process effectively.

1. Define the Product interface

This interface specifies the methods that all vehicle types are required to implement.

```
public interface Vehicle {
   void drive();
}
```

2. Create Concrete Product classes

These classes fulfill the Vehicle interface. An example includes classes for Car, Bike, and Truck, each of which implements the drive() method.

```
public class Car implements Vehicle {
   @Override
   public void drive() {
      System.out.println("Driving a Car");
   }
}
// Similar classes for Bike and Truck would be defined here
```

3. Create the Factory class

The factory class contains a method to create and return instances of concrete classes based on the provided input.

```
public class VehicleFactory {
   public Vehicle createVehicle(String vehicleType) {
      if (vehicleType == null) return null;
      if (vehicleType.equalsIgnoreCase("CAR")) return new Car();
      else if (vehicleType.equalsIgnoreCase("BIKE")) return new Bike();
      else if (vehicleType.equalsIgnoreCase("TRUCK")) return new Truck();
      return null;
   }
}
```

4. Use the factory in the Client code

The client utilizes the VehicleFactory to obtain vehicle objects without having a direct dependency on the concrete classes.

```
public class FactoryPatternDemo {
   public static void main(String[] args) {
      VehicleFactory factory = new VehicleFactory();
      Vehicle car = factory.createVehicle("CAR");
      if (car != null) car.drive(); // Output: Driving a Car
```

```
        // Similar usage for Bike and Truck
    }
}
```

Advantages of the Factory pattern

The Factory pattern provides numerous benefits:

- It separates clients from concrete classes, as the client interacts solely with the interface and the factory.
- It improves extensibility, enabling the addition of new vehicle types with minimal alterations to the existing code.
- It consolidates the creation logic, making management and modifications easier.

```
// Product interface
interface Product {
    void use();
}

// Concrete Products
class ConcreteProductA implements Product {
    @Override
    public void use() {
        System.out.println("Using ConcreteProductA");
    }
}

class ConcreteProductB implements Product {
    @Override
    public void use() {
        System.out.println("Using ConcreteProductB");
    }
}

// Creator interface or abstract class with a factory method
abstract class Creator {
```

```
    public abstract Product createProduct();

    public void doSomethingWithProduct() {
        Product product = createProduct();
        product.use();
    }
}

// Concrete Creators
class ConcreteCreatorA extends Creator {
    @Override
    public Product createProduct() {
        return new ConcreteProductA();
    }
}

class ConcreteCreatorB extends Creator {
    @Override
    public Product createProduct() {
        return new ConcreteProductB();
    }
}
```

Builder

The Builder pattern distinguishes the process of constructing a complex object from its representation, enabling the same construction method to yield various representations. This is especially beneficial when an object contains numerous optional parameters.

The Builder pattern facilitates the step-by-step construction of complex objects. It decouples the building logic from the object itself, allowing the same construction process to produce different variations. This pattern effectively addresses challenges such as an excessive number of constructors (the telescoping constructor anti-pattern) and aids in ensuring that objects are instantiated in a valid state.

Core components

The essential elements of the Builder pattern consist of:

- Product: The intricate object being constructed. In Java, this typically features a private constructor and final fields.
- Builder: A nested class, usually static, within the Product class that reflects the fields of the Product. It provides a fluent interface for setting parameters by returning itself.
- build() method: This method of the Builder creates and returns the final Product object.
- Director (Optional): This class may define the order of calls to the builder methods, which is advantageous for generating standard object configurations.

Sample Java code

A sample Java implementation illustrates the Builder pattern for constructing Computer objects with both required and optional components. The Computer class employs a private constructor that accepts a builder. A ComputerBuilder class manages the step-by-step construction.

```
class Pizza {
   private String dough;
   private String sauce;
   private String topping;

   private Pizza(Builder builder) {
      this.dough = builder.dough;
      this.sauce = builder.sauce;
      this.topping = builder.topping;
   }

   public void describe() {
       System.out.println("Pizza with Dough: " + dough + ", Sauce: " + sauce + ",
Topping: " + topping);
   }

   // Builder Class
```

```
    public static class Builder {
        private String dough;
        private String sauce;
        private String topping;

        public Builder withDough(String dough) {
            this.dough = dough;
            return this;
        }

        public Builder withSauce(String sauce) {
            this.sauce = sauce;
            return this;
        }

        public Builder withTopping(String topping) {
            this.topping = topping;
            return this;
        }

        public Pizza build() {
            return new Pizza(this);
        }
    }
}
```

Advantages of the Builder pattern

- Enhances readability: The process of object creation is more transparent with named builder methods.
- Guarantees immutability: It facilitates the creation of immutable objects through a private constructor and the absence of public setters.
- Simplifies intricate construction: It effectively handles objects with numerous optional parameters and circumvents the telescoping constructor problem.
- Prevents inconsistent state: Objects are instantiated only after the builder has been completely configured.

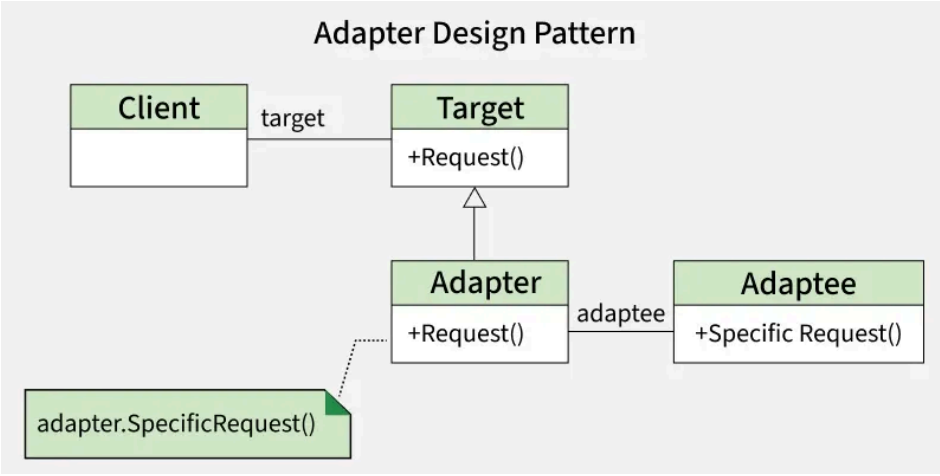
6.5 Structural Patterns: Adapter, Decorator, Composite

Adapter Design Pattern

The Adapter Design Pattern is a structural design pattern that serves as a bridge between two incompatible interfaces, enabling them to function together. This pattern is particularly beneficial for integrating legacy code or third-party libraries into a modern system.

To illustrate this concept, let us refer to the following diagram:

Diagram Explanation:



fig(6.5)

- The Client intends to utilize a Target interface (which invokes Request()).
- The Adaptee possesses valuable functionality; however, its method (SpecificRequest()) does not align with the Target interface.
- The Adapter functions as a bridge: it implements the Target interface (Request()), while internally invoking the Adaptee's SpecificRequest().
- This setup allows the Client to utilize the Adaptee without necessitating any alterations to its code.

Practical Application of the Adapter Design Pattern

The Adapter Design Pattern can be exemplified by a mobile adapter that aligns the power and voltage specifications of a device. In a similar vein, in software development, adapters are employed to facilitate interoperability between incompatible interfaces.

Software that operates with various file formats (such as CSV, JSON, XML) utilizes adapters to transform these formats into one that the application can process, thereby enhancing data interoperability.
Examples of adapters also include Device Drivers in Operating Systems, Database Connectors, and Language Converters (such as Chinese to English and English to Hindi).

In scenarios where there is a significant transition between new and old APIs, rather than rewriting the entire legacy code, an adapter can be employed to handle the conversion.

Advantages of the Adapter Design Pattern

The following are the advantages of the Adapter Design Pattern:

- Encourages code reuse without necessitating modifications.
- Maintains the focus of classes on core logic by isolating adaptation tasks.
- Facilitates support for multiple interfaces through interchangeable adapters.
- Decouples the system from specific implementations, simplifying modifications and replacements.

Disadvantages of the Adapter Design Pattern
Below are the disadvantages of the Adapter Design Pattern:

- Increases complexity, potentially making the code more difficult to understand.
- Causes a minor performance overhead due to additional indirection.
- The presence of multiple adapters can heighten maintenance efforts.

- There is a risk of excessive use for trivial modifications, resulting in unnecessary complexity.
- Managing numerous interfaces may necessitate several adapters, complicating the overall design.

Applications of the Adapter Design Pattern
The adapter design pattern can be utilized when:

- It facilitates communication between systems that are not compatible.
- It allows for the reuse of existing code or libraries without the need for rewriting.
- It streamlines the integration of new components, maintaining system flexibility.
- It centralizes changes related to compatibility, thereby simplifying and securing maintenance.

When to Avoid the Adapter Design Pattern?
The adapter design pattern should not be employed when:

- If the system is simple and all components are compatible, an adapter may be superfluous.
- Adapters can introduce a minor overhead, which could be a concern in environments sensitive to performance.
- When there are no compatibility issues with interfaces, utilizing an adapter may be redundant.
- For projects with a very brief lifespan, the effort required to implement an adapter may not be justified.

Elements of the Adapter Design Pattern
Below are the elements of the adapter design pattern:

- Target Interface: The interface anticipated by the client, outlining the operations it can perform.
- Adaptee: The existing class that possesses an incompatible interface requiring integration.
- Adapter: Implements the target interface and internally utilizes the adaptee, serving as a bridge.

- Client: Engages with the target interface, remaining unaware of the adapter or adaptee specifics.

Various Implementations of the Adapter Design Pattern

The Adapter Design Pattern can be utilized in multiple ways, contingent upon the programming language and the specific context. Below are the main implementations:

1. Class Adapter (Inheritance-based)

- In this method, the adapter class derives from both the target interface (the one anticipated by the client) and the adaptee (the existing class that requires adaptation).
- Programming languages that permit multiple inheritance, such as C++, are more inclined to adopt this technique.
- Conversely, in languages like Java and C#, which do not allow multiple inheritance, this method is less commonly employed.

2. Object Adapter (Composition-based)

- The object adapter utilizes composition rather than inheritance. In this implementation, the adapter contains an instance of the adaptee and fulfills the target interface.
- This method is more adaptable as it enables a single adapter to interact with multiple adaptees and avoids the complications associated with inheritance.
- The object adapter is frequently utilized in languages such as Java and C#.

3. Two-way Adapter

- A two-way adapter can operate as both a target and an adaptee, depending on which interface is being called.
- This type of adapter is particularly advantageous when two systems need to collaborate and require reciprocal adaptation.

4. Interface Adapter (Default Adapter)

- When only a limited number of methods from an interface are required, an interface adapter can be utilized.
- This is particularly beneficial in scenarios where the interface comprises numerous methods, and the adapter offers default implementations for those that are unnecessary.
- This method is often observed in languages like Java, where abstract classes or default method implementations in interfaces facilitate the implementation process.

How does the Adapter Design Pattern function?

The following outlines the operation of the adapter design pattern:

Step 1: The client begins a request by invoking a method on the adapter through the target interface.
Step 2: The adapter converts or translates the client's request into a format comprehensible to the adaptee by utilizing the adaptee's interface.
Step 3: The adaptee performs the actual task based on the request that has been translated by the adapter.
Step 4: The client obtains the results of the call, remaining oblivious to the existence of the adapter or the specific details pertaining to the adaptee.

Example of the Adapter Design Pattern

Let us explore the adapter design pattern with the help of an example:

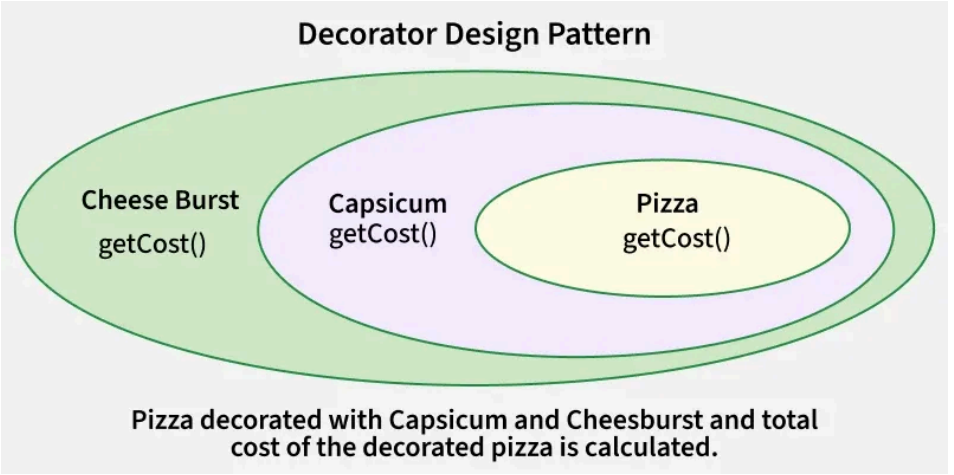
```
/* Adapter Design Pattern Example Code */

// Target Interface
interface Printer {
   void print();
}

// Adaptee
```

```
class LegacyPrinter {
    public void printDocument() {
        System.out.println("Legacy Printer is printing a document.");
    }
}

// Adapter
class PrinterAdapter implements Printer {
    private LegacyPrinter legacyPrinter = new LegacyPrinter();

    @Override
    public void print() {
        legacyPrinter.printDocument();
    }
}

// Client Code
public class Client {
    public static void clientCode(Printer printer) {
        printer.print();
    }

    public static void main(String[] args) {
        // Using the Adapter
        PrinterAdapter adapter = new PrinterAdapter();
        clientCode(adapter);
    }
}
```

Decorator Design Pattern

The Decorator Design Pattern is a structural pattern that allows for the dynamic addition of behavior to individual objects without altering other objects of the same class. It employs decorator classes to encapsulate concrete components, thereby enhancing functionality in a more flexible and reusable manner.

To illustrate this concept, let us refer to the following diagram:



fig(6.5)

The diagram depicts a Pizza (the base component) enveloped by Capsicum and subsequently Cheese Burst (the decorators). Each decorator contributes its own cost through the getCost() method, resulting in a final price calculated as Pizza + Capsicum + Cheese Burst. This exemplifies how the Decorator Pattern dynamically layers additional behavior.

Key Features of the Decorator Pattern

- Dynamic Behavior Addition: Responsibilities can be added or removed at runtime without altering the original object.
- Open-Closed Principle (OCP): Object behavior can be extended without modifying existing code.
- Composition over Inheritance: Multiple behaviors can be combined without establishing a rigid class hierarchy.
- Reusable Decorators: Identical decorators can be utilized across various objects.
- Flexible and Scalable: New features can be incorporated easily without affecting the existing code.

Real-World Examples

- Coffee Shop Application: Customers have the option to customize their coffee with add-ons such as milk, sugar, or whipped cream. Each add-on functions as a decorator that is dynamically applied to the coffee object.
- Video Streaming Platforms: Videos can feature subtitles, audio enhancements, multiple resolutions, or language options, all of which are added as decorators.
- Text Processing Applications: A plain text object can be wrapped with BoldDecorator, ItalicDecorator, or UnderlineDecorator to implement various formatting styles.
- Java I/O Library: Classes such as FileInputStream can be encapsulated by BufferedInputStream or DataInputStream to introduce additional functionality without modifying the core class.

Key Elements of the Decorator Design Pattern

- Component Interface: Establishes common operations for both components and decorators.
- Concrete Component: The fundamental object that provides basic functionality.
- Decorator: An abstract wrapper that contains a reference to a Component and enhances its behavior.
- Concrete Decorator: Specific decorators that augment the functionality of the component.

Example of the Decorator Design Pattern
Below is a problem statement to illustrate the Decorator Design Pattern:

Imagine we are developing a coffee shop application where customers can place orders for various types of coffee. Each coffee can include several optional add-ons such as milk, sugar, whipped cream, etc. Our goal is to create a system that allows us to dynamically incorporate these add-ons into a coffee order without altering the coffee classes themselves.

Employing the Decorator Pattern enables us to add optional features (add-ons) to coffee orders dynamically without changing the fundamental coffee classes. This approach enhances code flexibility, scalability, and maintainability, as new

add-ons can be seamlessly introduced and combined with different coffee order types.

Let us break down the code into components:

1. Component Interface (Coffee)

- This interface, Coffee, represents the component.
- It declares two methods, getDescription() and getCost(), which must be implemented by concrete components and decorators.

```
// Coffee.java
public interface Coffee {
   String getDescription();
   double getCost();
}
```

2. Concrete Component (PlainCoffee)

- PlainCoffee is a concrete class that implements the Coffee interface.
- It provides the description and cost of plain coffee by implementing the getDescription() and getCost() methods.

```
/*package whatever //do not write package name here */

import java.io.*;

class GFG {
   public static void main (String[] args) {
      System.out.println("GFG!");
   }
}
```

3. Decorator (CoffeeDecorator)

- CoffeeDecorator is an abstract class that implements the Coffee interface.
- It holds a reference to the decorated Coffee object.

- The methods getDescription() and getCost() are implemented to delegate to the decorated coffee object.

```
// PlainCoffee.java
public class PlainCoffee implements Coffee {
    @Override
    public String getDescription() {
        return "Plain Coffee";
    }

    @Override
    public double getCost() {
        return 2.0;
    }
}
```

4. Concrete Decorators (MilkDecorator, SugarDecorator)

- MilkDecorator and SugarDecorator are concrete decorators that extend CoffeeDecorator.
- They override getDescription() to append the respective decorator description.

```
public interface Coffee {
    String getDescription();
    double getCost();
}

public abstract class CoffeeDecorator implements Coffee {
    protected Coffee decoratedCoffee;
    public CoffeeDecorator(Coffee decoratedCoffee) {
        this.decoratedCoffee = decoratedCoffee;
    }
    public String getDescription() {
        return decoratedCoffee.getDescription();
    }
    public double getCost() {
        return decoratedCoffee.getCost();
```

```
    }
}

public class MilkDecorator extends CoffeeDecorator {
    public MilkDecorator(Coffee decoratedCoffee) {
        super(decoratedCoffee);
    }
    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription() + ", Milk";
    }
    @Override
    public double getCost() {
        return decoratedCoffee.getCost() + 0.5;
    }
}

public class SugarDecorator extends CoffeeDecorator {
    public SugarDecorator(Coffee decoratedCoffee) {
        super(decoratedCoffee);
    }
    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription() + ", Sugar";
    }
    @Override
    public double getCost() {
        return decoratedCoffee.getCost() + 0.2;
    }
}
```

The full code corresponding to the aforementioned problem statement is as follows:

Below is the complete code for the previously mentioned problem statement:

```
interface Coffee {
    String getDescription();
    double getCost();
}

class PlainCoffee implements Coffee {
    @Override
    public String getDescription() {
        return "Plain Coffee";
    }

    @Override
    public double getCost() {
        return 2.0;
    }
}

abstract class CoffeeDecorator implements Coffee {
    protected Coffee decoratedCoffee;

    public CoffeeDecorator(Coffee decoratedCoffee) {
        this.decoratedCoffee = decoratedCoffee;
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription();
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost();
    }
}

class MilkDecorator extends CoffeeDecorator {
    public MilkDecorator(Coffee decoratedCoffee) {
        super(decoratedCoffee);
```

```
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription() + ", Milk";
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost() + 0.5;
    }
}

class SugarDecorator extends CoffeeDecorator {
    public SugarDecorator(Coffee decoratedCoffee) {
        super(decoratedCoffee);
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription() + ", Sugar";
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost() + 0.2;
    }
}

public class Main {
    public static void main(String[] args) {
        // Plain Coffee
        Coffee coffee = new PlainCoffee();
        System.out.println("Description: " + coffee.getDescription());
        System.out.println("Cost: $" + coffee.getCost());

        // Coffee with Milk
        Coffee milkCoffee = new MilkDecorator(new PlainCoffee());
```

```
        System.out.println("\nDescription: " + milkCoffee.getDescription());
        System.out.println("Cost: $" + milkCoffee.getCost());

        // Coffee with Sugar and Milk
          Coffee sugarMilkCoffee = new SugarDecorator(new MilkDecorator(new
PlainCoffee()));
        System.out.println("\nDescription: " + sugarMilkCoffee.getDescription());
        System.out.println("Cost: $" + sugarMilkCoffee.getCost());
    }
}
```

Composite Design Pattern

The Composite Design Pattern is a structural design pattern that arranges objects into tree structures, enabling clients to handle both individual objects and groups of objects in a consistent manner.

As articulated by the Gang of Four, "Compose objects into a tree structure to illustrate part-whole hierarchies. The Composite allows clients to treat both individual objects and compositions of objects uniformly."

The fundamental idea is that one can manipulate a single instance of an object in the same way as one would manipulate a collection of such instances. The operations applicable to all composite objects typically share a relationship defined by the least common denominator.

Components of the Composite Design Pattern:

- Component – Establishes a shared interface that is implemented by both leaf objects and composite objects.
- Leaf – Denotes simple, indivisible objects that execute operations directly.
- Composite – Signifies complex objects that may contain children (either leaves or other composites) and delegate operations to them.
- Client – Interacts with all objects via the Component interface, treating both individual and composite objects in a uniform manner.

Example of the Composite Design Pattern in Java:

Consider a scenario where you are developing a project management system in which tasks can either be simple tasks or a collection of tasks (subtasks) that together form a larger task.

1. Task (Component)

- Represents the shared interface for both simple tasks and task lists.
- Defines methods such as getTitle(), setTitle(), and display().

```
// Component

public interface Task {
   String getTitle();
   void setTitle(String title);
   void display();
}
```

2. SimpleTask (Leaf)

- Represents an individual task characterized by a title.
- Implements the Task interface.

```
// Leaf

public class SimpleTask implements Task {
   private String title;

   public SimpleTask(String title) {
      this.title = title;
   }

   @Override
   public String getTitle() {
      return title;
   }

   @Override
   public void setTitle(String title) {
```

```
        this.title = title;
    }

    @Override
    public void display() {
        System.out.println("Simple Task: " + title);
    }
}
```

3. TaskList (Composite)

- Represents a collection of tasks, which may include both simple tasks and other task lists.
- Implements the Task interface while also maintaining a list of tasks (List<Task>).
- Defines methods for adding, removing, and displaying tasks.

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

// Composite

public class TaskList implements Task {
    private String title;
    private List<Task> tasks;

    public TaskList(String title) {
        this.title = title;
        this.tasks = new ArrayList<>();
    }

    @Override
    public String getTitle() {
        return title;
    }

    @Override
    public void setTitle(String title) {
```

```
        this.title = title;
    }

    public void addTask(Task task) {
        tasks.add(task);
    }

    public void removeTask(Task task) {
        tasks.remove(task);
    }

    @Override
    public void display() {
        System.out.println("Task List: " + title);
        for (Task task : tasks) {
            task.display();
        }
    }
}
```

4. TaskManagementApp (Client)

- Represents the application that employs the Composite Design Pattern to oversee tasks.
- It generates a combination of simple tasks and task lists, demonstrating how the Composite pattern facilitates the uniform treatment of both individual tasks and collections of tasks.

```
// Client

public class TaskManagementApp {
    public static void main(String[] args) {
        // Creating simple tasks
        Task simpleTask1 = new SimpleTask("Complete Coding");
        Task simpleTask2 = new SimpleTask("Write Documentation");

        // Creating a task list
        TaskList projectTasks = new TaskList("Project Tasks");
```

```
        projectTasks.addTask(simpleTask1);
        projectTasks.addTask(simpleTask2);

        // Nested task list
        TaskList phase1Tasks = new TaskList("Phase 1 Tasks");
        phase1Tasks.addTask(new SimpleTask("Design"));
        phase1Tasks.addTask(new SimpleTask("Implementation"));

        projectTasks.addTask(phase1Tasks);

        // Displaying tasks
        projectTasks.display();
    }
}
```

Complete code for the aforementioned example:

This code encompasses the Task, SimpleTask, TaskList, and TaskManagementApp classes. It illustrates the Composite Design Pattern utilized for structuring tasks within a project management system.

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

// Component
interface Task {
    String getTitle();
    void setTitle(String title);
    void display();
}

// Leaf
class SimpleTask implements Task {
    private String title;

    public SimpleTask(String title) {
        this.title = title;
```

```
    }

    @Override
    public String getTitle() {
        return title;
    }

    @Override
    public void setTitle(String title) {
        this.title = title;
    }

    @Override
    public void display() {
        System.out.println("Simple Task: " + title);
    }
}

// Composite
class TaskList implements Task {
    private String title;
    private List<Task> tasks;

    public TaskList(String title) {
        this.title = title;
        this.tasks = new ArrayList<>();
    }

    @Override
    public String getTitle() {
        return title;
    }

    @Override
    public void setTitle(String title) {
        this.title = title;
    }
```

```
    public void addTask(Task task) {
        tasks.add(task);
    }

    public void removeTask(Task task) {
        tasks.remove(task);
    }

    @Override
    public void display() {
        System.out.println("Task List: " + title);
        for (Task task : tasks) {
            task.display();
        }
    }
}

// Client
public class TaskManagementApp {
    public static void main(String[] args) {
        // Creating simple tasks
        Task simpleTask1 = new SimpleTask("Complete Coding");
        Task simpleTask2 = new SimpleTask("Write Documentation");

        // Creating a task list
        TaskList projectTasks = new TaskList("Project Tasks");
        projectTasks.addTask(simpleTask1);
        projectTasks.addTask(simpleTask2);

        // Nested task list
        TaskList phase1Tasks = new TaskList("Phase 1 Tasks");
        phase1Tasks.addTask(new SimpleTask("Design"));
        phase1Tasks.addTask(new SimpleTask("Implementation"));

        projectTasks.addTask(phase1Tasks);

        // Displaying tasks
        projectTasks.display();
```

```
    }
}
```

Why is the Composite Design Pattern necessary?

The Composite Design Pattern was developed to tackle particular issues associated with the representation and manipulation of hierarchical structures in a consistent manner.

Here are several points that underscore the necessity of the Composite Design Pattern:

- Uniform Interface – Treats both individual and composite objects identically, thereby simplifying client code.
- Hierarchical Structures – Particularly suitable for tree-like part-whole relationships.
- Flexibility & Scalability – Facilitates dynamic composition, making it easy to extend or modify structures.
- Common Operations – Establishes shared operations at the component level, which reduces duplication.
- Client Simplification – Offers a cohesive method for efficiently managing complex structures.

Application of the Composite Design Pattern

The Composite Pattern enables uniform handling of both single components and groups, thereby simplifying design and enhancing efficiency.

- It allows clients to interact with individual components and collections in a consistent manner.
- It diminishes redundant logic in managing primitives versus composites.
- It reduces resource consumption by limiting unnecessary objects.
- It promotes resource sharing for improved scalability and performance.

When should the Composite Design Pattern be avoided?

The Composite Design Pattern complicates the restriction of component types within a composite. Therefore, it should not be employed when there is no intention to represent a complete or partial hierarchy of objects.

- The Composite Design Pattern can lead to overly generalized designs.
- It complicates the restriction of components within a composite.
- At times, it may be necessary for a composite to contain only specific components. With the Composite, one cannot depend on the type system to enforce those constraints.
- Instead, run-time checks will be required.

PRACTISE SET

MULTIPLE CHOICES QUESTION

1. Which best describes the purpose of the Memento Design Pattern?
A. To define a family of algorithms and make them interchangeable
B. To capture and restore an object's internal state without exposing its implementation
C. To provide a unified interface to a set of interfaces in a subsystem
D. To notify dependent objects automatically when one object changes state

2. In the Memento pattern, which class is responsible for saving and restoring the state?
A. Originator
B. Caretaker
C. Memento
D. Observer

3. Which of the following is *not* a Creational Design Pattern?
A. Singleton
B. Factory
C. Builder
D. Adapter

4. What is the main purpose of the Singleton pattern?
A. To create multiple instances of a class
B. To ensure only one instance of a class exists globally
C. To encapsulate object creation logic
D. To hide the complexity of object creation

5. The Adapter pattern is primarily used to:
A. Add new functionality to an existing object dynamically
B. Allow incompatible interfaces to work together
C. Represent part-whole hierarchies
D. Maintain state history of objects

6. Which Structural Design Pattern allows you to treat individual objects and compositions uniformly?

A. Composite
B. Decorator
C. Proxy
D. Facade

7. The Observer pattern belongs to which category of design patterns?

A. Creational
B. Structural
C. Behavioral
D. Reactive

8. The Strategy pattern allows:

A. Dynamic switching of algorithms at runtime
B. Restoring previous states of an object
C. Wrapping functionality dynamically
D. Creating only one instance of a class

9. In Reactive Programming, what does the term "Reactive Streams" refer to?

A. Streams that store data persistently
B. Streams that support backpressure for asynchronous data flow
C. Streams that execute sequentially
D. Streams that manage synchronous operations

10. Which of the following is an example of an *anti-pattern* in Java?

A. Using dependency injection
B. God Object (an object that knows too much or does too much)
C. Applying composition over inheritance
D. Using the Strategy pattern

BRACE YOURSELF

1.Explain the importance of design patterns in software development. How do design patterns contribute to writing maintainable, reusable, and scalable Java applications?

2.Define the Memento Design Pattern. Describe its main components (Originator, Memento, and Caretaker) and explain how it supports the principle of encapsulation while enabling state restoration.

3.Discuss the steps involved in implementing the Memento pattern in Java. Provide a conceptual example demonstrating how an application (like a text editor or game) can use this pattern to implement “undo” functionality.

4.Describe the Singleton Design Pattern and explain how it ensures that only one instance of a class exists. What are some common pitfalls of Singleton implementation in Java, and how can they be avoided (e.g., thread safety, reflection attacks)?

5.Compare the Factory Method Pattern and the Builder Pattern. How do they differ in terms of object creation and flexibility? Provide an example scenario where each pattern would be most appropriate.

6.Explain the purpose of Structural Design Patterns. Discuss how the Adapter, Decorator, and Composite patterns help manage complex class structures. Provide real-world examples or Java use cases for each.

7.Discuss how Behavioral Design Patterns manage communication and responsibility between objects. Explain the key idea behind Observer, Strategy, and Command patterns, and describe a situation where each would be useful in Java programming.

8.Define Reactive Programming and explain how Event-Driven Architecture and Reactive Streams support responsiveness and scalability. How do these patterns differ from traditional imperative programming models in Java?

9.What are anti-patterns in software design? Describe at least two common Java anti-patterns (e.g., God Object, Spaghetti Code, Premature Optimization) and explain their negative effects on code quality and maintainability.

10.Discuss how multiple design patterns can be combined in a single project to solve complex problems. Provide an example where a combination of patterns (e.g., Factory + Singleton + Observer) was or could be used effectively in a real-world Java application.

Chapter 7: Integration

7.1 Spring Boot Integration: REST APIs & Microservices

The integration of Spring Boot with REST APIs and microservices enhances the efficiency of developing contemporary, distributed applications.

Spring Boot and REST APIs:

- Spring Boot facilitates the development of RESTful APIs by offering:

- Auto-configuration: It automatically sets up common components such as embedded web servers (Tomcat, Jetty, Undertow), allowing applications to run without the need for external server configurations.

- Spring Web Starter: This dependency includes essential libraries for creating web applications, such as Spring MVC and Jackson for processing JSON.

- Annotations: Annotations like @RestController, @RequestMapping, @GetMapping, @PostMapping, etc., enable developers to specify API endpoints, associate HTTP methods with controller methods, and manage request/response bodies with minimal boilerplate code.

- Embedded Servers: Executing the application as a standalone JAR with an embedded server simplifies deployment and removes the necessity for separate web server installations.

Spring Boot and Microservices:

- Spring Boot is an ideal framework for constructing microservices because of its:

- Rapid Development: Its opinionated defaults and auto-configuration speed up the creation of individual services, allowing teams to concentrate on business logic.

- Standalone Applications: Each microservice can be packaged as a separate Spring Boot application, making it straightforward to deploy and manage.

- Embedded Servers: The capability to include a server within each service simplifies deployment and lessens operational overhead.

- Integration with Spring Cloud: Spring Boot integrates effortlessly with Spring Cloud projects, which provide solutions for common microservice issues such as service discovery, configuration management, load balancing, and circuit breakers.

- Modularity: Spring Boot encourages a modular design, where each microservice is dedicated to a specific business function, resulting in improved organization and easier maintenance.

Integration of REST APIs and Microservices Utilizing Spring Boot:

- In a microservices architecture, services built on Spring Boot primarily communicate with one another through REST APIs.

- Service-to-Service Communication: Microservices provide their functionalities as RESTful endpoints, allowing other services to consume these APIs for interaction and data exchange.

- API Gateway: An API Gateway, often realized with Spring Cloud Gateway or Netflix Zuul, can be positioned in front of the microservices, serving as a unified entry point for external clients and directing requests to the corresponding backend services.

- Data Exchange: REST APIs enable data exchange between services using standard formats such as JSON or XML, thereby ensuring interoperability.

Key elements of Spring Boot integration for REST APIs and Microservices:

Streamlined REST API Development: Spring Boot, which is based on the Spring framework, offers annotations such as @RestController,

@RequestMapping, @GetMapping, @PostMapping, etc., to facilitate the definition of RESTful endpoints and the management of HTTP requests. It automates a significant portion of the configuration, enabling developers to concentrate on business logic.

Embedded Servers: Spring Boot incorporates embedded servers like Tomcat, Jetty, or Undertow, which removes the necessity for external server deployments. This simplifies the packaging and deployment of REST APIs as self-contained executable JAR files, making it ideal for microservices.

Auto-configuration: The auto-configuration feature of Spring Boot intelligently sets up various components based on the dependencies available in the project. This minimizes boilerplate code and enhances the setup of features such as database connections, security, and web-related configurations for REST APIs.

Support for Microservices Architecture: Spring Boot is widely regarded as a preferred option for developing microservices due to its lightweight design, rapid startup times, and straightforward deployment. It integrates seamlessly with Spring Cloud, offering tools for service discovery, load balancing, circuit breakers, and other vital microservices patterns.

API Integration with Other Services: Spring Boot applications can effortlessly consume external REST APIs using tools like RestTemplate (though now deprecated in favor of WebClient for reactive programming or Feign Client for declarative REST clients). This facilitates communication between microservices within an ecosystem or with external third-party services.

Production-Ready Features: Spring Boot provides features such as externalized configuration, health checks, metrics, and monitoring capabilities (via Spring Boot Actuator), which are essential for managing and operating REST APIs and microservices in production settings.

Dependency Management: Spring Boot Starters streamline the management of dependencies by offering carefully selected collections of dependencies tailored for typical use cases, such as spring-boot-starter-web, which is utilized for creating web applications and REST APIs.

In summary, Spring Boot's opinionated methodology, automatic configuration, and cohesive features render it a highly efficient and effective framework for the development and integration of REST APIs within a microservices architecture.

Example

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication
public class MyMicroserviceApplication {

    public static void main(String[] args) {
        SpringApplication.run(MyMicroserviceApplication.class, args);
    }
}

@RestController
class HelloWorldController {

    @GetMapping("/hello")
    public String sayHello() {
        return "Hello from my microservice!";
    }
}
```

7.2 Database Connect: JDBC, JPA, Hibernate

Java Database Connectivity (JDBC)

Java Database Connectivity (JDBC) is an API that facilitates interaction between Java applications and relational databases. It offers a standardized approach for connecting to a database, executing SQL queries, and obtaining results.

The procedure for establishing a JDBC connection and engaging with a database generally includes the following steps:

- Import the JDBC Package: Incorporate the required java.sql package into your Java code.

- Load or Register the Driver: The JDBC driver serves as a conduit between the Java application and the specific database. While earlier versions of JDBC necessitated explicit driver loading, JDBC 4.0 and subsequent versions typically manage this automatically if the driver JAR is included in the classpath.

- Establish a Connection: Utilize the DriverManager.getConnection() method with the correct JDBC URL, username, and password to create a connection to the database. The JDBC URL delineates the database type, location, and other connection specifics.

```
Connection con =
DriverManager.getConnection("jdbc:mysql://localhost:3306/mydatabase",
"username", "password");
```

- Create a Statement: Generate a Statement object from the Connection to execute SQL queries. For dynamic queries or to mitigate SQL injection risks, PreparedStatement is frequently the preferred choice.

```
Statement stmt = con.createStatement();
```

- Execute SQL Query: Carry out SQL queries using methods such as executeQuery() for SELECT statements or executeUpdate() for INSERT, UPDATE, or DELETE operations.

```
ResultSet rs = stmt.executeQuery("SELECT * FROM employees");
// Or for updates:
// int rowsAffected = stmt.executeUpdate("INSERT INTO employees (name)
VALUES ('John Doe')");
```

- Process Results (if applicable): If a SELECT query was executed, handle the ResultSet to extract the data.

```
while (rs.next()) {
   System.out.println(rs.getString("name"));
}
```

- Close Resources: Ensure to close the ResultSet, Statement, and Connection objects within a finally block to free database resources and avert resource leaks.

```
if (rs != null) rs.close();
if (stmt != null) stmt.close();
if (con != null) con.close();
```

- **Example:**

```
import java.sql.*;


public class JdbcExample {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/mydb";
        String user = "root";
        String password = "password";


         try (Connection conn = DriverManager.getConnection(url, user,
password);
              Statement stmt = conn.createStatement();
                   ResultSet rs = stmt.executeQuery("SELECT * FROM
users")) {


              while (rs.next()) {
                  System.out.println("User ID: " + rs.getInt("id") + ", Name:
" + rs.getString("name"));
              }
         } catch (SQLException e) {
```

```
            e.printStackTrace();
        }
    }
}
```

JPA (Java Persistence API)

Establishing a connection to a database through JPA (Java Persistence API) primarily requires the configuration of the persistence unit and the acquisition of an EntityManager instance. Below is a detailed explanation of the procedure:

1. Dependencies:

- Initially, confirm that your project includes the essential dependencies. This generally encompasses:
- A JPA provider (for instance, Hibernate or EclipseLink).
- The JDBC driver corresponds to your specific database (such as MySQL Connector/J or PostgreSQL JDBC Driver).
- If you are utilizing Spring Boot, you may want to add spring-boot-starter-data-jpa along with the appropriate database connector.

2. Configuration of persistence.xml:

- JPA depends on a persistence.xml file, typically found in META-INF/persistence.xml, to outline persistence units. This file delineates:
- Persistence Unit Name: A distinct identifier for the persistence unit.
- JPA Provider: The class associated with your selected JPA implementation.
- Data Source Properties: Details for database connection, including URL, username, and password.
- Entity Classes: A list of classes that correspond to your database tables.

Example of a persistence.xml (simplified for illustration):

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_2.xsd"
         version="2.2">
                                <persistence-unit          name="myPersistenceUnit"
transaction-type="RESOURCE_LOCAL">
      <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
      <properties>
                                 <property    name="jakarta.persistence.jdbc.driver"
value="com.mysql.cj.jdbc.Driver"/>
                                   <property    name="jakarta.persistence.jdbc.url"
value="jdbc:mysql://localhost:3306/mydatabase"/>
         <property name="jakarta.persistence.jdbc.user" value="root"/>
                              <property    name="jakarta.persistence.jdbc.password"
value="password"/>
         <property name="hibernate.hbm2ddl.auto" value="update"/>
         <property name="hibernate.show_sql" value="true"/>
      </properties>
   </persistence-unit>
</persistence>
```

3. Acquiring an EntityManager:

In order to engage with the database, it is necessary to have an instance of EntityManager. This instance is derived from an EntityManagerFactory.

```
import jakarta.persistence.EntityManager;
import jakarta.persistence.EntityManagerFactory;
import jakarta.persistence.Persistence;

public class JpaConnectionExample {
   public static void main(String[] args) {
                                          EntityManagerFactory      emf      =
Persistence.createEntityManagerFactory("myPersistenceUnit");
      EntityManager em = emf.createEntityManager();
```

```
        try {
            // Perform database operations using the EntityManager
            // ...
        } finally {
            em.close();
            emf.close();
        }
    }
}
```

The method Persistence.createEntityManagerFactory("myPersistenceUnit") initializes an EntityManagerFactory according to the settings defined in persistence.xml for the designated persistence unit.

The call emf.createEntityManager() generates an instance of EntityManager, which serves as a link to the database, enabling operations such as persisting, merging, removing, and locating entities.

It is crucial to ensure that both EntityManager and EntityManagerFactory instances are appropriately closed (using em.close() and emf.close()) to free up resources. Typically, the EntityManagerFactory is instantiated once and utilized across the application, whereas EntityManager instances are created and terminated for each transaction or request.

```
@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    // ... getters and setters
}

// In a service layer
EntityManager em = entityManagerFactory.createEntityManager();
em.getTransaction().begin();
```

```
User user = new User("John Doe");
em.persist(user); // Saves the user object to the database
em.getTransaction().commit();
```

Hibernate

Establishing a database connection through Hibernate requires several essential steps and configuration components:

Add Dependencies: Incorporate the required Hibernate and database-specific JDBC driver dependencies into your project's build configuration (for instance, pom.xml for Maven).

```
<dependencies>
    <!-- Hibernate Core -->
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>{hibernate-version}</version>
    </dependency>
    <!-- Database-specific JDBC Driver (e.g., MySQL) -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>{mysql-connector-version}</version>
    </dependency>
</dependencies>
```

Configure Hibernate: Generate a Hibernate configuration file, usually named hibernate.cfg.xml, within your application's classpath (for example, src/main/resources). This file outlines the database connection properties along with other Hibernate configurations.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
```

```
    <session-factory>
        <!-- Database connection properties -->
                                                    <property
name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver</property
>
                                                    <property
name="hibernate.connection.url">jdbc:mysql://localhost:3306/your_database_n
ame</property>
                                                    <property
name="hibernate.connection.username">your_username</property>
                                                    <property
name="hibernate.connection.password">your_password</property>

        <!-- Hibernate Dialect (specific to your database) -->
                                                    <property
name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>

        <!-- Optional: Show SQL queries in console -->
        <property name="hibernate.show_sql">true</property>
        <property name="hibernate.format_sql">true</property>

          <!-- Optional: Automatic schema generation (for development/testing)
-->
        <property name="hibernate.hbm2ddl.auto">update</property>

        <!-- Mapping files or annotated classes -->
        <!-- <mapping class="com.example.YourEntityClass"/> -->
      </session-factory>
  </hibernate-configuration>
```

Create SessionFactory: In your Java application, utilize the Configuration object to load the hibernate.cfg.xml file and construct a SessionFactory. The SessionFactory is a thread-safe entity that oversees Session instances.

```
  import org.hibernate.SessionFactory;
  import org.hibernate.cfg.Configuration;
```

```
public class HibernateUtil {
    private static final SessionFactory sessionFactory;

    static {
        try {
                                                    sessionFactory = new
Configuration().configure().buildSessionFactory();
        } catch (Throwable ex) {
            System.err.println("Initial SessionFactory creation failed." + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }
}
```

Obtain Session and Perform Operations: Retrieve a Session from the SessionFactory to engage with the database. A Session signifies a singular unit of work and is not thread-safe.

```
import org.hibernate.Session;
import org.hibernate.Transaction;

public class Main {
    public static void main(String[] args) {
        Session session = null;
        Transaction transaction = null;
        try {
            session = HibernateUtil.getSessionFactory().openSession();
            transaction = session.beginTransaction();

            // Perform database operations (e.g., save, retrieve, update, delete)
            // Example: session.save(new YourEntity());

            transaction.commit();
```

```
        } catch (Exception e) {
            if (transaction != null) {
                transaction.rollback();
            }
            e.printStackTrace();
        } finally {
            if (session != null) {
                session.close();
            }
        }
    }
}
```

- Definition: Hibernate is a widely-used, open-source ORM framework that implements the JPA specification. It provides additional functionalities that extend beyond the JPA standard.

- Functionality: Hibernate adheres to the JPA interfaces and offers the foundational mechanism for mapping Java objects to relational database tables. It manages SQL generation, caching, and various other ORM capabilities. You have the option to utilize Hibernate directly (native Hibernate) or as the JPA provider.

- Relationship with JPA: Hibernate serves as an implementation of JPA. This indicates that you can develop your application using JPA annotations and APIs, subsequently selecting Hibernate as the underlying ORM provider. This enhances the portability of your application, allowing for a theoretical transition to another JPA implementation (such as EclipseLink) with minimal code alterations.

7.3 Messaging Systems: Kafka, RabbitMQ

Kafka:

Apache Kafka serves as a distributed streaming platform that is frequently utilized as a high-throughput, fault-tolerant messaging system. In Java applications, Kafka clients facilitate interaction with a Kafka cluster, enabling applications to produce and consume messages.

Key Components in Kafka with Java:

- Producers: Java applications responsible for sending messages (records) to Kafka topics.

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092"); // Kafka broker address
                                                    props.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
                                                  props.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");

Producer<String, String> producer = new KafkaProducer<>(props);
producer.send(new ProducerRecord<>("my_topic", "key", "value"));
producer.close();
```

- Consumers: Java applications that subscribe to Kafka topics to receive messages.

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("group.id", "my_consumer_group"); // Consumer group ID
                                                  props.put("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
                                                props.put("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");

Consumer<String, String> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Collections.singletonList("my_topic"));

while (true) {
                        ConsumerRecords<String, String> records =
consumer.poll(Duration.ofMillis(100));
    for (ConsumerRecord<String, String> record : records) {
            System.out.printf("offset = %d, key = %s, value = %s%n",
record.offset(), record.key(), record.value());
    }
```

```
}
// consumer.close(); // Close when done
```

- Topics: Categories or feeds where records are published. Topics are segmented into partitions to enhance scalability and parallelism.
- Brokers: Kafka servers that store messages and manage requests from both producers and consumers.
- Consumer Groups: A collection of consumers that share a common group.id and work together to consume messages from one or more topics. Each message within a partition is delivered to only one consumer in a group.
- Architecture: A distributed streaming platform that functions as a commit log, employing a "dumb broker/smart consumer" model.
- Use cases: Real-time analytics, log aggregation, and the development of event-driven architectures.
- Performance: Exceptional throughput, capable of processing millions of messages each second.
- Message handling: Retains messages in a log with a policy-driven retention period, enabling consumers to re-read messages.
- Routing: Lacks built-in intelligent routing; consumers obtain all messages from a topic and can subsequently filter them, or developers may implement custom logic.
- Scalability: Extremely scalable in a horizontal manner.
- Message priority: Does not accommodate message priorities.

Advantages of using Kafka in Java Messaging:

- High Throughput: Engineered to efficiently manage large volumes of data.
- Scalability: Can be scaled horizontally by incorporating additional brokers and partitions.
- Fault Tolerance: Data is replicated across brokers, guaranteeing availability even if certain components fail.
- Durability: Messages are stored on disk for a configurable retention period, permitting replay and historical analysis.
- Real-time Processing: Supports real-time data streaming and event-driven architectures.

RabbitMQ

RabbitMQ is a message broker software that is open-source and implements the Advanced Message Queuing Protocol (AMQP), enabling asynchronous communication among various components of a distributed system. In the Java programming language, RabbitMQ can be utilized to create robust and scalable messaging solutions.

Key Concepts:

- Producer: This refers to an application that transmits messages to RabbitMQ.
- Consumer: This is an application that receives and processes messages from RabbitMQ.
- Exchange: This component receives messages from producers and directs them to queues according to specific rules (such as exchange types including direct, topic, fanout, and headers).
- Queue: This is where messages are held until they are consumed by a consumer.
- Binding: This defines the connection between an exchange and a queue, determining which messages from an exchange are sent to a specific queue.

Using RabbitMQ in Java:

- Dependency: It is necessary to include the RabbitMQ Java client library in your project (for instance, by using Maven or Gradle).

```
<!-- Maven -->
<dependency>
    <groupId>com.rabbitmq</groupId>
    <artifactId>amqp-client</artifactId>
    <version>5.20.0</version> <!-- Use a recent version -->
</dependency>
```

- Connection and Channel: You must establish a connection to the RabbitMQ server and create a channel, which is utilized for sending and receiving messages.

```
ConnectionFactory factory = new ConnectionFactory();
factory.setHost("localhost"); // Or your RabbitMQ host
Connection connection = factory.newConnection();
Channel channel = connection.createChannel();
```

- Declaring Exchanges and Queues: You need to declare the exchanges and queues that will be used for message routing and storage.

```
channel.exchangeDeclare("my_exchange", "fanout"); // Example: fanout exchange
channel.queueDeclare("my_queue", true, false, false, null); // Example: durable queue
channel.queueBind("my_queue", "my_exchange", ""); // Example: bind queue to exchange
```

- Sending Messages (Producer): Messages are published to an exchange.

```
DeliverCallback deliverCallback = (consumerTag, delivery) -> {
    String message = new String(delivery.getBody(), "UTF-8");
    System.out.println("Received: '" + message + "'");
        channel.basicAck(delivery.getEnvelope().getDeliveryTag(), false); // Acknowledge message
};
channel.basicConsume("my_queue", false, deliverCallback, consumerTag -> {});
```

Receiving Messages (Consumer): A consumer is created to listen for messages on a designated queue.

Benefits:

- Asynchronous Communication: This decouples the components responsible for sending and receiving messages.

- Reliability: Features such as message acknowledgments and persistence guarantee message delivery.

- Scalability: It supports clustering to ensure high availability and load balancing.

- Flexibility: There are various exchange types and routing options available for different messaging patterns.

- Architecture: A conventional message broker that operates on a "smart broker/dumb consumer" paradigm, featuring adaptable routing based on message characteristics.

- Use cases: Task scheduling, communication between microservices, request-response messaging, and job queues.

- Performance: Exhibits lower throughput compared to Kafka, typically managing thousands of messages each second.

- Message handling: Brokers are responsible for message delivery, and messages are usually removed following successful acknowledgment by a consumer.

- Routing: Facilitates flexible routing through various exchange types (direct, topic, fanout, header) to direct messages to designated queues.

- Scalability: Capable of horizontal scaling but encounters performance constraints under extremely high loads.

- Message priority: Accommodates message priorities, enabling high-priority messages to be processed first.

7.4 Containerization: Docker, Jib & Kubernetes Basics

Docker

Docker serves as a containerization platform that enables the packaging of your application along with all its dependencies into containers, ensuring that

your application operates smoothly across various environments, whether for development, testing, or production purposes. It is a tool specifically designed to facilitate the creation, deployment, and execution of applications through the use of containers.

```
FROM node:18-alpine
WORKDIR /app
COPY package.json .
RUN npm install
COPY . .
CMD ["npm", "start"]
```

Recognized as the foremost software container platform globally, Docker was introduced in 2013 by a company initially named Dotcloud, Inc., which later changed its name to Docker, Inc. The platform is developed using the Go programming language. Despite being launched only six years ago, Docker has already seen a significant transition from virtual machines (VMs) to its containerization approach within various communities. It is tailored to provide advantages for both developers and system administrators, thereby integrating into numerous DevOps toolchains. Developers can focus on coding without the need to concern themselves with the testing and production environments. Similarly, system administrators can manage infrastructure effortlessly, as Docker allows for easy scaling of systems up or down. Docker plays a crucial role during the deployment phase of the software development lifecycle.

The architecture of Docker comprises the Docker client, the Docker Daemon operating on the Docker Host, and the Docker Hub repository. Docker employs a client-server architecture where the client interacts with the Docker Daemon on the Docker Host through a combination of REST APIs, Socket IO, and TCP. To build a Docker image, the client issues a build command to the Docker Daemon, which then constructs an image based on the provided inputs and stores it in the Docker registry. If creating an image is not desired, one can simply execute the pull command from the client, prompting the Docker Daemon to retrieve the image from Docker Hub. Finally, to run the image, the client must execute the run command, which will generate the container.

Components of Docker

The primary components of Docker consist of Docker clients and servers, Docker images, Dockerfile, Docker Registries, and Docker containers. These components are elaborated upon in the following section:

Docker Clients and Servers - Docker operates on a client-server architecture. The Docker Daemon/Server encompasses all containers. The Docker Daemon/Server receives requests from the Docker client via CLI or REST APIs and processes these requests accordingly. The Docker client and Daemon may reside on the same host or on different hosts.

Advantages of Docker -

Docker has gained significant popularity in recent times due to the numerous benefits offered by Docker containers. The primary advantages of Docker include:

- Speed - Docker containers operate at a much faster speed compared to virtual machines. The time taken to create a container is minimal since they are compact and lightweight. Consequently, development, testing, and deployment processes can be expedited as containers are small in size. Once built, containers can be swiftly pushed for testing and subsequently transitioned to the production environment.

- Portability - Applications developed within Docker containers exhibit exceptional portability. These applications can be effortlessly relocated as a single unit, maintaining consistent performance regardless of their location.

- Scalability - Docker possesses the capability to be deployed across multiple physical servers, data servers, and cloud platforms. It is also compatible with all Linux machines. Containers can be seamlessly transferred from a cloud environment to a local host and vice versa at an impressive speed.

- Density - Docker optimizes the utilization of available resources more effectively as it does not rely on a hypervisor. This efficiency allows for a greater number of containers to be operated on a single host compared

to virtual machines. Docker containers deliver superior performance due to their high density and the absence of resource overhead wastage.

There are three categories of networks in Docker -

- Bridged network: When a new Docker container is instantiated without the –network argument, Docker automatically connects the container to the bridge network. In bridged networks, all containers on a single host can communicate via their IP addresses. A Bridge network is established when the scope of Docker hosts is limited to one, meaning all containers operate on a single host. To create a network that spans multiple Docker hosts, an overlay network is required.

- Host network: When a new Docker container is created using the –network=host argument, it integrates the container into the host network stack where the Docker daemon operates. All interfaces of the host become accessible from the container assigned to the host network

- None network: When a new Docker container is created with the –network=none argument, it places the Docker container in its own network stack. Consequently, in this none network, no IP addresses are allocated to the container, preventing them from communicating with one another.

Jib

Jib is an open-source tool created by Google that streamlines the containerization of Java applications. It operates as a collection of plugins for widely-used build tools such as Maven and Gradle, enabling developers to create optimized, OCI-compliant container images for their Java applications without the necessity of a Docker daemon or Dockerfile.

Here are the primary features and advantages of Jib:

- Dockerfile-less Containerization: Jib removes the requirement to write and manage Dockerfiles, thereby simplifying the containerization process for Java applications. It intelligently builds the container image based on the structure of your Maven or Gradle project.

- No Docker Daemon Required: Jib can generate container images and push them directly to a container registry (such as Docker Hub or Google Container Registry) without needing a locally running Docker daemon. This enhances the development workflow and eliminates a dependency.

- Optimized Image Layers: Jib automatically optimizes the layering of your container image, distinguishing dependencies from application code. This results in smaller image sizes and quicker rebuilds, as only the modified layers need to be refreshed.

- Fast Development Cycles: By integrating directly into your build system, Jib facilitates rapid iteration and testing of containerized applications, making image building a seamless component of the development process.

- Reproducible Builds: Jib guarantees that container images are built in a reproducible manner, meaning that the same source code will reliably yield the same image, irrespective of the build environment.

- Easy Integration: Jib easily integrates into existing Maven and Gradle projects by simply adding the Jib plugin to your pom.xml or build.gradle file. You can then set the target image and other configurations.

- Supports JVM Languages: Although primarily intended for Java, Jib can also be utilized to containerize applications developed in other JVM languages such as Kotlin.

Benefits of Jib:

- No requirement for a Dockerfile.
- Accelerated build times, particularly in CI/CD workflows.
- Enhanced security by removing the necessity to install Docker on the build environment.

Kubernetes

Kubernetes, commonly referred to as K8s, is an open-source platform designed for container orchestration, facilitating the automation of deployment, scaling, and management of containerized applications. It offers a comprehensive framework for executing and overseeing containerized workloads across a cluster of machines.

Key features of Kubernetes in relation to containerization:

- Container Orchestration: While containerization technologies such as Docker encapsulate applications and their dependencies into portable containers, Kubernetes oversees the lifecycle of these containers on a large scale. It automates various tasks, including the deployment, scaling, updating, and recovery of applications across numerous containers and hosts.

- Cluster Management: Kubernetes functions on a cluster of nodes, which can be either physical or virtual machines. It comprises a control plane (master node) that governs the cluster and worker nodes that execute the actual containerized applications.

- Automated Operations: Kubernetes automates a multitude of operational tasks, which include:

- Deployment: Implementing containerized applications in accordance with specified configurations.

- Scaling: Automatically adjusting the scale of applications based on demand and resource usage.

- Self-healing: Identifying and substituting failed containers to maintain application availability.

- Load Balancing: Allocating network traffic among multiple instances of an application.

- Cloud-Native Applications: Kubernetes is especially advantageous for cloud-native applications developed using a microservices architecture,

allowing for effective management and scaling of these distributed systems.

- Portability: Kubernetes facilitates workload portability across diverse environments, including on-premises data centers, public clouds (such as AWS, Google Cloud, Azure), and hybrid cloud configurations.

- Ecosystem: Kubernetes boasts a vast and dynamic ecosystem, featuring numerous tools and integrations for monitoring, logging, security, and other facets of container management.

```
# deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: java-app-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: java-app
  template:
    metadata:
      labels:
        app: java-app
    spec:
      containers:
      - name: java-app-container
        image: your-registry/your-image:latest
        ports:
        - containerPort: 8080
---
# service.yaml
apiVersion: v1
kind: Service
metadata:
  name: java-app-service
spec:
```

```
  selector:
    app: java-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
  type: LoadBalancer # Or ClusterIP for internal access
```

Advantages of Kubernetes for Java Applications:

- Scalability: Seamlessly adjust the scale of Java applications according to demand.
- High Availability: Kubernetes automatically manages failures by restarting or rescheduling pods.
- Automated Deployments: Streamline the deployment and updates of Java applications.
- Resource Management: Effectively oversee CPU and memory resources for Java applications.
- Service Discovery: Offers integrated service discovery for interaction among various microservices.

7.5 CI/CD Pipelines: GitHub Actions, Jenkins

GitHub CI/CD pipeline

A GitHub CI/CD pipeline, primarily executed through GitHub Actions, streamlines the software development lifecycle from code integration to deployment. This methodology merges Continuous Integration (CI) with Continuous Delivery/Deployment (CD).

Continuous Integration (CI):

- Automated Builds and Tests: When developers commit code changes to a repository, CI automatically initiates builds and executes tests (including unit tests, integration tests, etc.) to verify that the new code integrates smoothly with the existing codebase and does not introduce regressions.

- Early Feedback: Developers obtain prompt feedback regarding the quality and functionality of their code, enabling them to resolve issues swiftly.

Continuous Delivery/Deployment (CD):

- Automated Release Process: Following successful CI, CD automates the preparation and delivery of the application for deployment. This may include packaging the application, generating deployment artifacts, and even automatically deploying to a staging or production environment.

- Faster Releases: CD facilitates more frequent and dependable releases of new features and bug fixes, minimizing manual effort and the risk of human error.

How GitHub Actions support CI/CD:

- Workflows: CI/CD workflows are defined in YAML files located in the .github/workflows directory of your repository. These workflows outline a sequence of jobs and steps that need to be executed.

- Events: Various GitHub events can trigger workflows, including branch pushes, pull requests, scheduled times, or even manual initiation.

- Jobs and Steps: A workflow is made up of one or more jobs, each of which operates on a virtual machine or container (runner). Jobs consist of individual steps that may involve actions such as checking out code, executing scripts, building Docker images, or deploying to cloud services.

- Actions: GitHub Actions utilize a marketplace of pre-existing actions or enable the creation of custom actions to carry out specific tasks within your workflow steps.

- Secrets: Sensitive data, such as API keys or credentials, can be securely stored as secrets in your GitHub repository and accessed by your workflows.

Example

```
name: CI/CD Pipeline

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

jobs:
  build-and-test:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Set up Node.js
        uses: actions/setup-node@v4
        with:
          node-version: '20'

      - name: Install dependencies
        run: npm install

      - name: Run tests
        run: npm test
```

```
deploy:
  runs-on: ubuntu-latest
  needs: build-and-test # This job depends on the success of 'build-and-test'
  if: github.ref == 'refs/heads/main' # Only deploy if pushing to main branch
  steps:
    - name: Checkout code
      uses: actions/checkout@v4

    - name: Deploy to production
      run: |
        # Your deployment commands here (e.g., using a cloud provider CLI)
        echo "Deploying application..."
```

Jenkins CI/CD pipeline

A Jenkins CI/CD pipeline automates the various stages of the software delivery process, from code commit to deployment. It facilitates continuous integration (CI) and continuous delivery/deployment (CD) by defining the entire workflow as code within a Jenkinsfile.

Key Concepts and Stages of a Jenkins CI/CD Pipeline:

Continuous Integration (CI):

- Source Code Management (SCM): The pipeline typically starts by fetching code from a version control system (e.g., Git) upon every commit.

- Build: The fetched code is then compiled and built into an executable artifact.

- Unit Testing: Automated unit tests are executed to ensure individual components function correctly.

Continuous Delivery/Deployment (CD):

- Integration Testing: After successful unit tests, the built artifact undergoes integration testing to verify interactions between different components.

- Quality Assurance (QA) / User Acceptance Testing (UAT): The application may be deployed to a staging environment for further testing by QA teams or end-users.

- Deployment: Upon successful completion of all tests, the application is automatically deployed to production or a designated environment.

Defining a Jenkins Pipeline:

Jenkins pipelines are defined using a Groovy-based Domain Specific Language (DSL) within a Jenkinsfile. This file is committed to the project's source code repository, enabling "Pipeline-as-Code" and ensuring version control and collaboration for the delivery process itself.

```
pipeline {
  agent any // Specifies where the pipeline will run

  stages {
    stage('Build') {
      steps {
        // Commands for building the application
        sh 'mvn clean install'
      }
    }
    stage('Test') {
      steps {
        // Commands for running tests
        sh 'mvn test'
      }
    }
    stage('Deploy') {
      steps {
        // Commands for deploying the application
        sh 'ansible-playbook deploy.yml'
```

```
            }
        }
    }
    post { // Actions to perform after stages, regardless of success/failure
        always {
            echo 'Pipeline finished.'
        }
        success {
            echo 'Deployment successful!'
        }
        failure {
            echo 'Deployment failed!'
        }
    }
}
```

Benefits of Jenkins CI/CD Pipelines:

- Automation: Automates repetitive tasks, reducing manual effort and errors.
- Faster Feedback: Provides quick feedback on code changes, enabling early detection of issues.
- Improved Quality: Ensures consistent testing and deployment processes, leading to higher quality software.
- Increased Efficiency: Streamlines the software delivery lifecycle, accelerating time-to-market.
- Visibility: Offers clear visibility into the progress and status of the delivery process.

Key Differences:

- Hosting: GitHub Actions provides managed runners, whereas Jenkins is generally self-hosted.

- Configuration: GitHub Actions employs YAML for workflows, in contrast to Jenkins which utilizes Groovy DSL for pipelines.

- Integration: GitHub Actions is closely integrated with GitHub, while Jenkins allows for wider integration through its plugin ecosystem.

- Learning Curve: GitHub Actions is frequently regarded as more accessible for newcomers because of its YAML syntax and managed environment, whereas Jenkins may present a more challenging learning curve due to its comprehensive features and plugin management.

7.6 Cloud & API Integration: AWS SDK, OpenAI, Stripe

The AWS SDK (Software Development Kit) is a suite of tools designed to facilitate the integration of applications with Amazon Web Services (AWS) cloud offerings. It includes libraries, code examples, documentation, and tools across various programming languages to engage with AWS APIs.

Key features of utilizing the AWS SDK for cloud and API integration include:

- Simplified API Interaction: The SDK simplifies the complexities associated with direct interactions with AWS service APIs, such as authentication, request signing, and error management. This enables developers to concentrate on application logic instead of intricate API details.

- Language-Specific Libraries: AWS provides SDKs for widely-used programming languages such as Python (Boto3), Java, JavaScript, .NET, Go, PHP, Ruby, and C++. This allows developers to utilize their

preferred programming language to create applications that utilize AWS services.

- Access to AWS Services: The SDK offers methods and classes for interacting with a diverse array of AWS services, including Amazon S3 (storage), Amazon EC2 (compute), Amazon DynamoDB (database), AWS Lambda (serverless functions), Amazon SQS (messaging), and many others.

- Integration with Development Environments: The SDK integrates effortlessly with various development environments, simplifying the process of building, testing, and deploying applications that connect with AWS.

- Authentication and Authorization: The SDK manages the secure authentication and authorization processes necessary for accessing AWS resources, typically employing AWS credentials or IAM roles.

- Error Handling and Retries: It includes built-in features for managing common API errors and implementing retry logic, thereby enhancing the resilience of applications.

Setting Up the Environment:
Ensure that a Java Development Kit (JDK) is properly installed.
Incorporate the AWS SDK for Java as a dependency within your project (for instance, by utilizing Maven or Gradle).

AWS Credentials Management:

- The SDK necessitates AWS credentials (Access Key ID and Secret Access Key) for authentication with AWS services.

- These credentials can be set up in multiple ways, such as through environment variables, shared credentials files located at ~/.aws/credentials, or IAM roles assigned to EC2 instances.

Interacting with AWS Services:

- The SDK offers client objects for each AWS service (for example, S3Client for Amazon S3 and DynamoDbClient for Amazon DynamoDB)

- These client objects provide methods that correspond to the API operations of the service (such as putObject for S3 and getItem for DynamoDB).

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.s3.S3Client;
import software.amazon.awssdk.services.s3.model.CreateBucketRequest;
import software.amazon.awssdk.services.s3.model.PutObjectRequest;
import java.io.File;

public class S3IntegrationExample {

   public static void main(String[] args) {
      // Create an S3 client
      S3Client s3Client = S3Client.builder()
            .region(Region.US_EAST_1) // Specify your desired region
            .build();

      String bucketName = "my-unique-java-sdk-bucket";
      String key = "my-file.txt";
        String filePath = "path/to/my-local-file.txt"; // Replace with your local file
path

      try {
         // Create an S3 bucket

s3Client.createBucket(CreateBucketRequest.builder().bucket(bucketName).buil
d());
          System.out.println("Bucket created: " + bucketName);

          // Upload a file to the bucket

s3Client.putObject(PutObjectRequest.builder().bucket(bucketName).key(key).b
uild(),
```

```
                new File(filePath).toPath());
            System.out.println("File uploaded: " + key);

        } catch (Exception e) {
            System.err.println("Error interacting with S3: " + e.getMessage());
        } finally {
            // Close the client when done
            s3Client.close();
        }
    }
}
```

OpenAI's API

Integrating OpenAI's API into Java code for cloud applications generally entails the following steps:

Acquire an OpenAI API Key:

- Register or log into the OpenAI platform.
- Access the API Keys section within your account dashboard.
- Create a new secret key and ensure it is stored securely, as it will not be retrievable after its initial creation.

Select an Integration Method:

- Direct HTTP Client: You may utilize standard Java HTTP clients (such as java.net.HttpURLConnection or libraries like Apache HttpClient or OkHttp) to send direct HTTP requests to OpenAI's API endpoints. This process includes constructing JSON request bodies, setting the necessary headers (including your API key for authentication), and interpreting JSON responses.

- OpenAI Java Client Library: OpenAI offers an official Java client library that streamlines interaction with the API by abstracting the details of the HTTP requests.

- Spring AI (for Spring Boot Applications): If you are creating a Spring Boot application, Spring AI provides a comprehensive framework for integrating with various AI models, including OpenAI, offering auto-configuration and simplified interaction through its ChatClient API.

Execute the Integration (Example using OpenAI Java Client Library):

```
import com.openai.OpenAIClient;
import com.openai.models.ChatCompletionRequest;
import com.openai.models.ChatMessage;
import com.openai.models.ChatCompletionResponse;
import com.openai.models.ChatCompletionChoice;
import com.openai.models.MessageRole;
import com.openai.models.Model;

import java.util.Collections;

public class OpenAIIntegrationExample {

    public static void main(String[] args) {
        // Replace with your actual OpenAI API key
        String apiKey = System.getenv("OPENAI_API_KEY");
        if (apiKey == null || apiKey.isEmpty()) {
            System.err.println("OPENAI_API_KEY environment variable not set.");
            return;
        }

        OpenAIClient client = new OpenAIClient(apiKey);

        try {
            ChatMessage userMessage = new ChatMessage(MessageRole.USER, "Suggest a recipe for dinner.");

            ChatCompletionRequest request = ChatCompletionRequest.builder()
                .model(Model.GPT_3_5_TURBO) // Or another desired model
                .messages(Collections.singletonList(userMessage))
```

```
                .build();

        ChatCompletionResponse response = client.chat().create(request);

        for (ChatCompletionChoice choice : response.getChoices()) {
                                        System.out.println("AI Response: " +
choice.getMessage().getContent());
        }

    } catch (Exception e) {
                System.err.println("Error interacting with OpenAI API: " +
e.getMessage());
        e.printStackTrace();
    }
  }
}
```

Manage Authentication and Error Handling:

- Safeguard your API key securely (for instance, by utilizing environment variables).
- Establish comprehensive error handling to address possible API errors or network complications.

Cloud Deployment Considerations:

- When deploying within a cloud environment (such as Azure App Service or AWS Elastic Beanstalk), verify that your application can securely access the API key and that network settings permit communication with OpenAI's endpoints.
- It is advisable to consider employing managed identities or secret management services offered by your cloud provider to enhance security.

Integrating Stripe

Integrating Stripe into a Java application for cloud and API integration requires several essential steps:

1. Setup and Dependencies:

- Create a Stripe Account: Acquire your secret and publishable API keys from the Stripe Dashboard.
- Add Stripe Java Dependency: Incorporate the Stripe Java library into your project's pom.xml (Maven) or build.gradle (Gradle) file:

```
<dependency>
  <groupId>com.stripe</groupId>
  <artifactId>stripe-java</artifactId>
    <version>LATEST_VERSION</version> <!-- Replace with the latest version -->
</dependency>
```

2. Initialize Stripe API Key:

Programmatically set your secret API key in your application, preferably within a service or configuration class, ensuring it is securely loaded (for instance, from environment variables or a properties file).

```
import com.stripe.Stripe;

// ...
 Stripe.apiKey = "sk_test_YOUR_SECRET_KEY"; // Use your actual secret key
```

3. Client-Side Integration (Stripe.js):

To securely manage sensitive payment information (such as credit card details), utilize Stripe.js on the client side to tokenize card data and transmit only a token to your Java backend. This prevents direct handling of PCI-sensitive data on your server.

4. Server-Side Processing (Java):

- Create a Charge Request POJO: Define a Java object to encapsulate payment details received from the client, including the Stripe token generated by Stripe.js.
- Implement Payment Logic: Develop a service or controller method to manage the payment processing:
- Receive the Stripe token and other payment details from the client.
- Utilize the Stripe Java library to create a Charge or PaymentIntent object using the received token and amount.
- Process the response from Stripe, including success or failure.

```
import com.stripe.model.Charge;
import com.stripe.exception.StripeException;
import java.util.HashMap;
import java.util.Map;

// ...
 public Charge createCharge(String stripeToken, int amount, String currency) throws StripeException {
    Map<String, Object> chargeParams = new HashMap<>();
    chargeParams.put("amount", amount);
    chargeParams.put("currency", currency);
        chargeParams.put("source", stripeToken); // Token obtained from client-side
    chargeParams.put("description", "Example Charge");

    return Charge.create(chargeParams);
}
```

5. Handle Webhooks (Optional but Recommended):

Establish webhook endpoints in your Java application to receive asynchronous notifications from Stripe regarding events such as successful payments, refunds, or subscription changes. This ensures your system remains synchronized with Stripe's state.

6. Error Handling and Security:

- Implement comprehensive error handling for Stripe API calls.

- Avoid hardcoding sensitive API keys directly in your code.
- Ensure secure communication (HTTPS) between your client and server, as well as between your server and Stripe.

PRACTISE SET

MULTIPLE CHOICES QUESTION

1. Which annotation in Spring Boot is used to define a REST controller?
A. @Service
B. @Controller
C. @RestController
D. @Component

2. What is the primary purpose of building microservices with Spring Boot?
A. To combine all features into one large application
B. To create lightweight, independently deployable services
C. To eliminate the need for databases
D. To improve front-end design

3. Which JDBC component is responsible for executing SQL statements?
A. Connection
B. ResultSet
C. Statement
D. DriverManager

4. What does JPA (Java Persistence API) primarily provide?
A. A framework for sending messages between microservices
B. A specification for object-relational mapping (ORM)
C. A system for managing threads
D. A container for running Spring Boot applications

5. Which of the following is true about Hibernate?
A. It is a messaging system for Java applications
B. It is a Java-based ORM framework that implements JPA
C. It only supports NoSQL databases
D. It is used for building Docker images

6. In Apache Kafka, what is a *topic*?

A. A thread that sends messages
B. A named channel for storing and organizing messages
C. A consumer of messages
D. A database table

7. RabbitMQ uses which messaging protocol by default?
A. MQTT
B. HTTP
C. AMQP (Advanced Message Queuing Protocol)
D. JMS (Java Message Service)

8. Which command is used to build a Docker image from a Dockerfile?
A. docker build
B. docker run
C. docker compose
D. docker exec

9. What is the main purpose of GitHub Actions in CI/CD pipelines?
A. To host web applications
B. To automate testing, building, and deployment workflows
C. To manage container orchestration
D. To create REST APIs

10. In cloud integration, what is the purpose of using the AWS SDK in Java?
A. To directly compile Java applications
B. To interact programmatically with AWS services like S3 and EC2
C. To run containers on Docker
D. To configure Kubernetes clusters

BRACE YOURSELF

1.Explain how Spring Boot simplifies the development of RESTful APIs. Include in your answer how annotations like @RestController, @RequestMapping, and @Autowired work together to create and manage microservices.

2.Describe the advantages of using a microservices architecture in a Spring Boot application. How does it differ from a monolithic architecture in terms of scalability, deployment, and maintenance?

3.Discuss the process of connecting a Spring Boot application to a relational database using JDBC. Include steps such as configuring the database driver, establishing a connection, executing queries, and handling exceptions.

4.Differentiate between JDBC, JPA, and Hibernate. Explain how Hibernate implements the JPA specification and how it helps in object-relational mapping (ORM) to simplify database operations in Java applications.

5.Explain how Apache Kafka is used in microservices architecture for message streaming. Describe the roles of producers, consumers, brokers, and topics, and how Kafka ensures message reliability and fault tolerance.

6.Discuss the working principles of RabbitMQ and its AMQP (Advanced Message Queuing Protocol). How does RabbitMQ manage message queues, exchanges, and routing keys in a distributed system?

7.Describe how Docker and Google Jib are used to containerize Spring Boot applications. Explain how Kubernetes helps in orchestrating containers, scaling applications, and managing deployments in production environments.

8.Explain the concept of Continuous Integration and Continuous Deployment (CI/CD). Compare how GitHub Actions and Jenkins automate build, test, and deployment processes for Java microservices.

9.Discuss how Java developers can use the AWS SDK to integrate Spring Boot applications with cloud services such as Amazon S3, EC2, or RDS. Provide an

example of how a file upload or database connection can be implemented using the SDK.

10.Describe how external APIs like OpenAI and Stripe can be integrated into a Spring Boot project. Explain how to securely manage API keys, handle authentication, and consume REST APIs using tools like RestTemplate or WebClient.

CONCLUSION

CONCLUSION

The journey commenced with an introduction to Design Patterns, where we discovered that these patterns serve as time-honored solutions to recurring software design challenges. By grasping the essence of design patterns, their purposes, and the criteria for selecting the appropriate one, developers can enhance code readability, minimize complexity, and foster collaboration among teams. The classification into Creational, Structural, and Behavioral patterns offers a clear framework for addressing problems in software architecture.

Modern Java development benefits from a robust ecosystem of tools.

We compared integrated development environments (IDEs) such as IntelliJ IDEA and VS Code, examined build tools like Maven, Gradle, and JBang, and assessed testing frameworks including Mockito and TestNG.

Additionally, we recognized the significance of monitoring (utilizing JFR, VisualVM, and Micrometer), upholding code quality (with SonarQube, SpotBugs, and OWASP tools), and utilizing AI-driven coding assistants like GitHub Copilot and Amazon CodeWhisperer to enhance productivity.

The Command Pattern demonstrated how to encapsulate operations as objects, facilitating undo/redo functionality, task queuing, and improved control over execution logic.

The Memento Pattern expanded on this by preserving an object's state, enabling safe rollback and recovery — crucial for implementing undo mechanisms and checkpoints in software.

Chapter 6 unified multiple patterns under a single framework — encompassing Singleton, Factory, Builder, Decorator, Composite, Observer, Strategy, and Command — illustrating how they collectively address various architectural concerns.

We also explored Reactive and Event-Driven Patterns, which empower systems to be responsive, resilient, and scalable in real-time environments, and discussed Anti-Patterns to emphasize common pitfalls to avoid in Java development. Finally, Chapter 7 connected theory to practice by investigating Spring Boot

integration, database connectivity (through JDBC, JPA, and Hibernate), and messaging systems such as Kafka and RabbitMQ.

We learned how Docker, Jib, and Kubernetes provide portability and scalability in deployment.

_____ALL THE BEST_____