Design Patterns in Python e Java

Una Guida Introduttiva E Completa Ai Modelli Di Progettazione

IVO ROBERTIS

Elenco Dei Libri Dello Stesso Autore

- **Crea il tuo Gioco Roguelike con JavaScript: Guida Passo-Passo per Sviluppatori**

  https://amzn.eu/d/7oOX4no

- **Crea il tuo Gioco Platform con JavaScript: Guida Passo-Passo per Sviluppatori alla Fisica dei Giochi**

  https://amzn.eu/d/38cr4lE

- **Quaderno delle Password: Un Taccuino Privato e Organizer di Password con schede in Ordine Alfabetico A-Z per Tenere Traccia e Proteggere i tuoi Dati. Ideale per Genitori, Papà, Mamma, Nonne e Nonni**

  https://amzn.eu/d/aGLuvdV

Introduzione

Nel mondo dinamico e in costante evoluzione dello sviluppo software, la comprensione e l'applicazione dei design patterns rappresentano un aspetto fondamentale per ogni sviluppatore. Questo documento ha offerto una serie di esempi pratici in Python e Java, mirati a fornire una visione chiara e concreta dei vari design patterns. Dalle strutture architetturali come microservizi e cloud computing, ai pattern comportamentali e funzionali, abbiamo esplorato un ampio spettro di soluzioni progettuali, illustrate attraverso codice facilmente comprensibile.

È importante sottolineare che gli esempi forniti in questo documento servono esclusivamente come punti di partenza o idee di base per comprendere il funzionamento e l'applicazione dei design patterns nel mondo reale. Non sono intesi come soluzioni complete o pronte per la produzione, ma piuttosto come strumenti didattici per stimolare ulteriori ricerche e sperimentazioni.

Gli esempi presentati possono necessitare di adattamenti significativi per soddisfare requisiti specifici o per essere integrati in applicazioni software complesse. Inoltre, i concetti di design e architettura del software sono soggetti a contesti e requisiti in continua evoluzione, pertanto è fondamentale che ogni soluzione venga valutata e personalizzata in base alle esigenze specifiche del progetto.

Infine, si consiglia di approfondire ulteriormente attraverso risorse aggiuntive e di rimanere aggiornati sulle best practices e sulle tendenze emergenti nel campo dello sviluppo software.

Questa continua ricerca e apprendimento rappresenta il cuore pulsante della crescita professionale di ogni sviluppatore software.

Capitolo 2: Modelli Di Creazione

Esempi In Python

1. **Modello Singleton**: questo modello garantisce che una classe disponga di una sola istanza e fornisca un punto di accesso globale a tale istanza. In Python, puoi implementarlo usando una classe con un metodo di classe che controlla se esiste già un'istanza.

```
class Singleton:
    _instance = None

    @classmethod
    def getInstance(cls):
        if cls._instance is None:
            cls._instance = cls()
        return cls._instance

# Singleton
s1 = Singleton.getInstance()
s2 = Singleton.getInstance()
print(s1 == s2)  # Returns True, s1 and s2 are the same instance
```

2. **Factory Method Pattern**: questo modello fornisce un'interfaccia per la creazione di un oggetto, ma consente alle sottoclassi di modificare il tipo di oggetti che verranno creati. È possibile creare una classe base con un metodo factory e sottoclassi che implementano questo metodo.

```
class Button:
    def render(self):
        raise NotImplementedError

class WindowsButton(Button):
    def render(self):
        return "Rendering Windows button"

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

def get_button(platform):
    if platform == "Windows":
        return WindowsButton()
```

```
    elif platform == "MacOS":
        return MacOSButton()
    else:
        raise ValueError("Unknown platform")

# Factory Method
button = get_button("Windows")
print(button.render())
```

3. **Builder Pattern**: utilizzato per costruire un oggetto complesso passo dopo passo. È utile quando è necessario creare un oggetto con molte possibili configurazioni e caratteristiche. In Python, puoi implementarlo con una classe 'Builder' separata.

```
class Car:
    def __init__(self):
        self.wheels = 4
        self.engine = None

class CarBuilder:
    def __init__(self):
        self.car = Car()

    def add_engine(self, engine_type):
        self.car.engine = engine_type
        return self

    def get_car(self):
        return self.car

# Builder Pattern
builder = CarBuilder()
car = builder.add_engine("V8").get_car()
print(car.engine)  # Print "V8"
```

Esempi In Java

1. **Singleton Pattern**: In Java, il Singleton può essere implementato con un campo statico privato che tiene traccia dell'istanza e di un metodo statico per ottenerla.

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

    private Singleton() {}
```

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

// Singleton
Singleton s1 = Singleton.getInstance();
Singleton s2 = Singleton.getInstance();
System.out.println(s1 == s2); // Print "true", s1 e s2 are the same instance
```

2. **Factory Method Pattern**: questo modello può essere implementato in Java definendo un'interfaccia per la creazione di oggetti e lasciando che le sottoclassi implementino questi metodi.

```
interface Button {
    String render();
}

class WindowsButton implements Button {
    public String render() {
        return "Rendering Windows button";
    }
}

class MacOSButton implements Button {
    public String render() {
        return "Rendering MacOS button";
    }
}

class ButtonFactory {
    public static Button getButton(String platform) {
        if (platform.equals("Windows")) {
            return new WindowsButton();
        } else if (platform.equals("MacOS")) {
            return new MacOSButton();
        } else {
            throw new IllegalArgumentException("Unknown platform");
        }
    }
```

```
}

// Factory Method
Button button = ButtonFactory.getButton("Windows");
System.out.println(button.render());
```

3**. Builder Pattern**: In Java, il Builder Pattern viene spesso utilizzato per costruire oggetti complessi. All'interno della classe di prodotto viene creata una classe 'Builder' interna.

```
public class Car {
    private int wheels;
    private String engine;

    public static class Builder {
        private Car car;

        public Builder() {
            car = new Car();
        }

        public Builder setWheels(int wheels) {
            car.wheels = wheels;
            return this;
        }

        public Builder setEngine(String engine) {
            car.engine = engine;
            return this;
        }

        public Car build() {
            return car;
        }
    }
}

// Builder Pattern
Car car = new Car.Builder()
                .setEngine("V8")
                .setWheels(4)
                .build();
System.out.println(car);  // Print the Object Car
```

Capitolo 3: Modelli Strutturali

Esempi In Python

1**. Modello Adattatore**: utilizzato per convertire l'interfaccia di una classe in un'altra interfaccia prevista da un client. L'adapter consente di collaborare con classi che altrimenti sarebbero incompatibili a causa di interfacce incompatibili.

```
class EuropeanSocket:
    def voltage(self):
        return 230

class AmericanSocket:
    def voltage(self):
        return 120

class SocketAdapter:
    def __init__(self, socket):
        self.socket = socket

    def voltage(self):
        return self.socket.voltage()

# Adapter
eu_socket = EuropeanSocket()
socket_adapter = SocketAdapter(eu_socket)
print(socket_adapter.voltage()) # Print 230
```

2. **Modello Composito**: consente di comporre gli oggetti in strutture ad albero per rappresentare gerarchie parte-intero. Il Composito consente di trattare i singoli oggetti e le composizioni di oggetti in modo uniforme.

```
class Component:
    def operation(self):
        pass

class Leaf(Component):
    def operation(self):
        return "Leaf"
```

```
class Composite(Component):
    def __init__(self):
        self._children = []

    def add(self, component):
        self._children.append(component)

    def operation(self):
        results = []
        for child in self._children:
            results.append(child.operation())
        return f"Branch({'+'.join(results)})"

# Composite
tree = Composite()
left_child = Composite()
left_child.add(Leaf())
left_child.add(Leaf())

right_child = Leaf()

tree.add(left_child)
tree.add(right_child)

print(tree.operation())  # Print "Branch(Leaf+Leaf+Leaf)”
```

3. **Modello Proxy**: fornisce un sostituto o un segnaposto per un altro oggetto per controllarne l'accesso. È utile per il lazy loading, il controllo degli accessi, la registrazione, ecc.

```
class RealSubject:

    def request(self):

        return "RealSubject: Handling request."


class Proxy:

    def __init__(self, real_subject):

        self._real_subject = real_subject
```

```
    def request(self):
        if self.check_access():
            return self._real_subject.request()
        else:
            return "Proxy: Access denied."

    def check_access(self):
        # Access control
        return True

# Proxy
real_subject = RealSubject()
proxy = Proxy(real_subject)
print(proxy.request())  # Print "RealSubject: Handling request."
```

Esempi In Java

1. **Modello dell'adattatore**: questo modello viene utilizzato per convertire l'interfaccia di una classe in un'altra interfaccia prevista dai client. Consente la collaborazione con classi che normalmente sarebbero incompatibili a causa di interfacce incompatibili.

```
interface EuropeanSocketInterface {
    int provideVoltage();
}

class EuropeanSocket implements EuropeanSocketInterface {
    public int provideVoltage() {
        return 230;
    }
}

interface USASocketInterface {
```

```
    int provideVoltage();
}

class SocketAdapter implements USASocketInterface {
    EuropeanSocketInterface socket;

    public SocketAdapter(EuropeanSocketInterface socket) {
        this.socket = socket;
    }

    public int provideVoltage() {
        return socket.provideVoltage();
    }
}

// Uso dell'Adapter
EuropeanSocketInterface socket = new EuropeanSocket();
USASocketInterface adapter = new SocketAdapter(socket);
System.out.println(adapter.provideVoltage());  // Print 230
```

2. **Modello Composito**: consente di trattare i singoli oggetti e le composizioni di oggetti in modo uniforme. Viene utilizzato per creare strutture ad albero che rappresentano gerarchie parte-intero.

```
interface Component {
    void operation();
}

class Leaf implements Component {
    public void operation() {
        System.out.println("Leaf");
    }
}

class Composite implements Component {
    private List<Component> children = new ArrayList<>();

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

    public void operation() {
        for (Component child : children) {
            child.operation();
        }
```

```
        }
}

// Composite
Composite root = new Composite();
root.add(new Leaf());
root.add(new Leaf());

Composite subTree = new Composite();
subTree.add(new Leaf());

root.add(subTree);

root.operation();  // Print "Leaf Leaf Leaf"
```

3. **Modello Proxy**: fornisce un sostituto o un segnaposto per un altro oggetto per controllarne l'accesso. Utilizzato per il controllo degli accessi, l'inizializzazione differita, la registrazione, ecc.

```
interface Subject {
        void request();
}

class RealSubject implements Subject {
        public void request() {
                System.out.println("RealSubject: Handling request.");
        }
}

class Proxy implements Subject {
        private RealSubject realSubject;

        public void request() {
                if (this.realSubject == null) {
                        this.realSubject = new RealSubject();
                }
                realSubject.request();
        }
}

// Proxy
Subject proxy = new Proxy();
proxy.request();  // Print "RealSubject: Handling request."
```

Capitolo 4: Modelli Comportamentali

Esempi In Python

1. **Modello Osservatore**: utilizzato per creare una relazione uno-a-molti tra gli oggetti, in modo che quando un oggetto cambia stato, tutti i suoi dipendenti vengono notificati e aggiornati automaticamente.

```
class Subject:
    def __init__(self):
        self._observers = []

    def attach(self, observer):
        self._observers.append(observer)

    def notify(self):
        for observer in self._observers:
            observer.update(self)

class ConcreteSubject(Subject):
    def __init__(self):
        super().__init__()
        self._state = None
    @property
    def state(self):
        return self._state

    @state.setter
    def state(self, value):
        self._state = value
        self.notify()

class Observer:
    def update(self, subject):
        pass

class ConcreteObserver(Observer):
    def update(self, subject):
        print(f"Observer: Reacted to the state change at {subject.state}")

# Uso dell'Observer Pattern
subject = ConcreteSubject()
observer_a = ConcreteObserver()
```

```
subject.attach(observer_a)
subject.state = "state1”
```

2. **Modello Strategy**: consente di definire una famiglia di algoritmi, incapsularli come oggetti e renderli intercambiabili. La strategia consente di variare l'algoritmo indipendentemente dai client che lo utilizzano.

```
from abc import ABC, abstractmethod

class Strategy(ABC):
    @abstractmethod
    def execute(self, data):
        pass

class ConcreteStrategyA(Strategy):
    def execute(self, data):
        return sorted(data)

class ConcreteStrategyB(Strategy):
    def execute(self, data):
        return reversed(sorted(data))

class Context:
    def __init__(self, strategy):
        self._strategy = strategy

    def set_strategy(self, strategy):
        self._strategy = strategy

    def execute_strategy(self, data):
        return self._strategy.execute(data)

# Strategy Pattern
data = [23, 5, 12, 8]
context = Context(ConcreteStrategyA())
print(list(context.execute_strategy(data)))

context.set_strategy(ConcreteStrategyB())
print(list(context.execute_strategy(data)))
```

3. **Modello di Comando**: incapsula una richiesta come oggetto, consentendo così ai client di essere parametrizzati con code, richieste e operazioni. È utile per operazioni come l'annullamento e la registrazione.

```
class Command(ABC):
    @abstractmethod
    def execute(self):
        pass

class ConcreteCommand(Command):
    def __init__(self, receiver):
        self._receiver = receiver

    def execute(self):
        self._receiver.action()

class Receiver:

    def action(self):
        print("Action executed.")

class Invoker:
    def __init__(self):
        self._command = None

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

    def execute_command(self):
        self._command.execute()

# Command Pattern
receiver = Receiver()
command = ConcreteCommand(receiver)
invoker = Invoker()
invoker.set_command(command)
invoker.execute_command()
```

Esempi In Java

1. **Modello Osservatore: utilizzato per creare una relazione uno-a-molti in cui più oggetti osservatore vengono notificati e aggiornati in risposta ai cambiamenti di stato in un oggetto soggetto.**

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

interface Observer {
	void update(String message);
}

class ConcreteObserver implements Observer {
	public void update(String message) {
		System.out.println("Observer received: " + message);
	}
}

class Subject {
	private List<Observer> observers = new ArrayList<>();
	private String state;

	public void attach(Observer observer) {
		observers.add(observer);
	}

	public void setState(String state) {
		this.state = state;
		notifyAllObservers();
	}
	private void notifyAllObservers() {
		for (Observer observer : observers) {
			observer.update(state);
		}
	}
}

// Observer Pattern
Subject subject = new Subject();
Observer observer = new ConcreteObserver();
subject.attach(observer);

subject.setState("New state");
```

2. **Modello Strategy**: consente di definire una famiglia di algoritmi, incapsularli come oggetti e renderli intercambiabili. Questo modello consente di variare l'algoritmo indipendentemente dai client che lo utilizzano.

```
interface Strategy {
	void execute();
}

class ConcreteStrategyA implements Strategy {
	public void execute() {
		System.out.println("Execution of strategy A");
	}
}

class ConcreteStrategyB implements Strategy {
	public void execute() {
		System.out.println("Execution of strategy A");
	}
}

class Context {
	private Strategy strategy;

	public Context(Strategy strategy) {
		this.strategy = strategy;
	}

	public void setStrategy(Strategy strategy) {
		this.strategy = strategy;
	}
	public void executeStrategy() {
		strategy.execute();
	}
}

// Strategy Pattern
Context context = new Context(new ConcreteStrategyA());
context.executeStrategy();

context.setStrategy(new ConcreteStrategyB());
context.executeStrategy();
```

3. **Modello di Comando**: incapsula una richiesta come oggetto, consentendo di parametrizzare i client con code, richieste e operazioni.

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

class ConcreteCommand implements Command {
	private Receiver receiver;

	public ConcreteCommand(Receiver receiver) {
		this.receiver = receiver;
	}

	public void execute() {
		receiver.action();
	}
}

class Receiver {
	public void action() {
		System.out.println("Action executed.");
	}
}

class Invoker {
	private Command command;

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

	public void executeCommand() {
		command.execute();
	}
}

// Command Pattern
Receiver receiver = new Receiver();
Command command = new ConcreteCommand(receiver);
Invoker invoker = new Invoker();

invoker.setCommand(command);
invoker.executeCommand();
```

Capitolo 5: Modelli Architetturali

Esempi In Python

1. **Modello MVC (Model-View-Controller):** questo modello separa l'applicazione in tre componenti principali: Model, View e Controller. Si tratta di un modello molto comune per lo sviluppo di applicazioni web.

```
# Model
class DataModel:
    def __init__(self):
        self.data = "Initial Data"

    def update_data(self, new_data):
        self.data = new_data

# View
class DataView:
    def display(self, data):
        print(f"Data: {data}")

# Controller
class DataController:
    def __init__(self, model, view):
        self.model = model
        self.view = view

    def update_model_data(self, new_data):
        self.model.update_data(new_data)

    def display_view(self):
        self.view.display(self.model.data)

# MVC Pattern
model = DataModel()
view = DataView()
controller = DataController(model, view)
```

```
controller.update_model_data("New data")
controller.display_view()
```

2. **Modello di pubblicazione-sottoscrizione**: in questo modello, i messaggi vengono inviati dagli editori ai sottoscrittori in modo asincrono. È utile per la comunicazione tra i diversi componenti di un sistema.

```
class Publisher:
    def __init__(self):
    self.subscribers = set()

    def subscribe(self, subscriber):
  self.subscribers.add(subscriber)

    def unsubscribe(self, subscriber):
  self.subscribers.discard(subscriber)

    def notify(self, message):
    for subscriber in self.subscribers:
     subscriber.update(message)

class Subscriber:
    def update(self, message):
    print(f"Message: {message}")

# Publish-Subscribe Pattern
publisher = Publisher()
subscriber = Subscriber()
publisher.subscribe(subscriber)

publisher.notify("Hello World!")
```

3. **Modello di architettura a strati**: questo modello organizza l'applicazione in livelli, ognuno dei quali ha responsabilità specifiche. Ad esempio, è possibile disporre di un livello di presentazione, di un livello della logica di business e di un livello di accesso ai dati.

```
# Data Access Layer
class DataAccessLayer:
    def get_data(self):
    return "Database data"

# Business Logic Layer
```

```
class BusinessLogicLayer:
        def __init__(self, dal):
        self.dal = dal

        def process_data(self):
        data = self.dal.get_data()
        return f"Data: {data}"

# Presentation Layer
class PresentationLayer:
        def __init__(self, bll):
        self.bll = bll

        def display_data(self):
        processed_data = self.bll.process_data()
        print(processed_data)

# Layered Architecture Pattern
dal = DataAccessLayer()
bll = BusinessLogicLayer(dal)
pl = PresentationLayer(bll)

pl.display_data()
```

Esempi In Java

1. **Model-View-Controller (MVC):** questo modello divide l'applicazione in tre componenti principali: il modello (Model), la vista (View) e il controller (Controller), ognuno con responsabilità specifiche.

```
        // Model
        class Model {
                private String data;

                public String getData() {
                        return data;
                }

                public void setData(String data) {
                        this.data = data;
                }
        }
```

```
// View
class View {
	public void printData(String data) {
		System.out.println("Data: " + data);
	}
}

// Controller
class Controller {
	private Model model;
	private View view;

	public Controller(Model model, View view) {
		this.model = model;
		this.view = view;
	}

	public void updateView() {
		view.printData(model.getData());
	}

	public void setData(String data) {
		model.setData(data);
	}
}

// MVC Pattern
Model model = new Model();
View view = new View();
Controller controller = new Controller(model, view);

controller.setData("MVC Example");
controller.updateView();
```

2. **Publish-Subscribe**: in questo modello, i messaggi vengono pubblicati da un oggetto e ricevuti da uno o più oggetti interessati (sottoscrittori), in modo asincrono.

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

// Publisher
class Publisher {
	private List<Subscriber> subscribers = new ArrayList<>();
```

```
    public void subscribe(Subscriber subscriber) {
        subscribers.add(subscriber);
    }

    public void unsubscribe(Subscriber subscriber) {
        subscribers.remove(subscriber);
    }

    public void publish(String message) {
        for (Subscriber subscriber : subscribers) {
            subscriber.update(message);
        }
    }
}

// Subscriber
interface Subscriber {
    void update(String message);
}

class ConcreteSubscriber implements Subscriber {
    public void update(String message) {
        System.out.println("Received: " + message);
    }
}

// Publish-Subscribe Pattern
Publisher publisher = new Publisher();
Subscriber subscriber = new ConcreteSubscriber();
publisher.subscribe(subscriber);

publisher.publish("Hello World!");
```

3**. Architettura a più livelli**: questo modello struttura l'applicazione in livelli logici, ognuno con responsabilità specifiche, ad esempio la presentazione, la logica di business e l'accesso ai dati.

```
// Data Access Layer
class DataAccessLayer {
    public String fetchData() {
        return "Dati";
    }
}
```

```
// Business Logic Layer
class BusinessLogicLayer {
	private DataAccessLayer dal;

	public BusinessLogicLayer(DataAccessLayer dal) {
		this.dal = dal;
	}

	public String processData() {
		String data = dal.fetchData();
		return "Elaborato: " + data;
	}
}

// Presentation Layer
class PresentationLayer {
	private BusinessLogicLayer bll;

	public PresentationLayer(BusinessLogicLayer bll) {
		this.bll = bll;
	}

	public void presentData() {
		String processedData = bll.processData();
		System.out.println(processedData);
	}
}

// Layered Architecture Pattern
DataAccessLayer dal = new DataAccessLayer();
BusinessLogicLayer bll = new BusinessLogicLayer(dal);
PresentationLayer pl = new PresentationLayer(bll);

pl.presentData();
```

Capitolo 6: Modelli Di Concorrenza

Esempi In Python

1. **Blocchi**: i blocchi vengono utilizzati per sincronizzare l'accesso alle risorse condivise tra thread diversi, impedendo race condition.

```
import threading

class Counter:
    def __init__(self):
        self.value = 0
        self.lock = threading.Lock()

    def increment(self):
        with self.lock:
            self.value += 1

counter = Counter()

def increment_counter():
    for _ in range(1000):
        counter.increment()

threads = [threading.Thread(target=increment_counter) for _ in range(10)]

for thread in threads:
    thread.start()

for thread in threads:
    thread.join()

print(f" Final value of the counter: {counter.value}")
```

2. **Modello produttore-consumatore**: in questo modello, i produttori generano dati e li inseriscono in una coda, mentre i consumatori prelevano i dati dalla coda per elaborarli.

```
import queue
import threading
import time

def producer(q):
    for i in range(5):
        print(f'Produced item {i}')
        q.put(i)
        time.sleep(1)
```

```
def consumer(q):
        while True:
                item = q.get()
                if item is None:
                        break
                print(f'Consumed item {item}')

q = queue.Queue()
t1 = threading.Thread(target=producer, args=(q,))
t2 = threading.Thread(target=consumer, args=(q,))

t1.start()
t2.start()

t1.join()
q.put(None)  # Termination signal for the consumer
t2.join()
```

3. **ThreadPool**: un pool di thread è una raccolta di thread preallocati che possono essere utilizzati per eseguire attività. Python fornisce un modulo 'concurrent.futures' per lavorare con i pool di thread.

```
from concurrent.futures import ThreadPoolExecutor

import time



def task(n):

        print(f'Execution of the task {n}')

        time.sleep(2)

        return n



with ThreadPoolExecutor(max_workers=3) as executor:

        futures = [executor.submit(task, i) for i in range(5)]



for future in futures:
```

```
print(f' The task has returned {future.result()}')
```

Esempi in Java

1. **Blocchi**: utilizzare "ReentrantLock" per gestire l'accesso sincronizzato a una risorsa condivisa.

```
import java.util.concurrent.locks.ReentrantLock;

class Counter {
	private int c = 0;
	private final ReentrantLock l = new ReentrantLock();

	public void increment() {
		l.lock();
		try {
			c++;
		} finally {
			l.unlock();
		}
	}

	public int getCount() {
		return c;
	}
}
```

Ora è possibile definire una classe denominata "Main" con un metodo "main", che funge da punto di ingresso del programma. All'interno del metodo "main", creare un'istanza della classe "Counter" denominata "c". Creare quindi due thread separati, "thread1" e "thread2". Ogni thread esegue un ciclo per incrementare l'oggetto "Counter" condiviso "c" di 100 volte all'interno di un blocco sincronizzato, garantendo un accesso simultaneo sicuro. Dopo aver avviato entrambi i thread utilizzando "start()" il programma attende che terminino la loro esecuzione utilizzando "join()". Infine, stampa il valore dell'oggetto "Counter" chiamando il metodo "getCount()" e lo visualizza come "Counter: [value]". Lo scopo di questo codice è dimostrare il multithreading in Java e come gestire la sincronizzazione per evitare race condition quando più thread accedono a risorse condivise.

2. **Modello produttore-consumatore:** implementare il modello produttore-consumatore utilizzando 'BlockingQueue'.

```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

class Producer implements Runnable {
	private final BlockingQueue<Integer> queue;
```

```
	Producer(BlockingQueue<Integer> q) {
		queue = q;
	}

	public void run() {
		try {
			for (int i = 0; i < 10; i++) {
				queue.put(i);
				System.out.println("q: " + i);
			}
		} catch (InterruptedException ex) {
			Thread.currentThread().interrupt();
		}
	}
}

class Consumer implements Runnable {
	private final BlockingQueue<Integer> queue;

	Consumer(BlockingQueue<Integer> q) {
		queue = q;
	}

	public void run() {
		try {
			while (true) {
				System.out.println("Consumed " + queue.take());
			}
		} catch (InterruptedException ex) {
			Thread.currentThread().interrupt();
		}
	}
}

public class Main {
	public static void main(String[] args) {
		BlockingQueue<Integer> queue = new ArrayBlockingQueue<>(10);

		Producer p = new Producer(queue);
		Consumer c = new Consumer(queue);

		new Thread(p).start();
		new Thread(c).start();
	}
}
```

3. **ThreadPool**: utilizzare 'ExecutorService' per gestire un pool di thread.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class Main {
	public static void main(String[] args) {
		ExecutorService executor = Executors.newFixedThreadPool(2);

		for (int t = 0; t < 10; t++) {
			int taskId = t;
			executor.submit(() -> {
				System.out.println("Task executed " + taskId
					+ " in " +
Thread.currentThread().getName());
			});
		}

		executor.shutdown();
		try {
			executor.awaitTermination(1, TimeUnit.MINUTES);
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
	}
}
```

Capitolo 7: Modelli Specifici Di Dominio

Esempi In Python

1. **Modello di repository**: questo modello è comune nelle applicazioni con una gestione intensiva dei dati. Consente di separare la logica di business dalla logica di accesso ai dati, fornendo un'astrazione per l'accesso ai dati.

```
class DataModel:
	def __init__(self, data):
```

```
        self.data = data

class Repository:
    def __init__(self):
        self.datasource = {}

    def add(self, key, value):
        self.datasource[key] = value

    def get(self, key):
        return self.datasource.get(key)

# Repository Pattern
repo = Repository()
repo.add("key1", DataModel("value1"))
model = repo.get("key1")
print(model.data)
```

2. **Modello del livello di servizio**: questo modello organizza la logica di business in un livello di servizio separato, facilitando la gestione delle operazioni aziendali.

```
class DataService:
    def __init__(self, repository):
        self.repository = repository

    def processData(self, key):
        data = self.repository.get(key)
        return data.data.upper()  # Example

# Service Layer Pattern
repo = Repository()
repo.add("key1", DataModel("value1"))
service = DataService(repo)

processed_data = service.processData("key1")
print(processed_data)
```

3. **Modello di record attivo**: in questo modello, gli oggetti incorporano sia i dati che i comportamenti relativi al database. È utile negli scenari in cui i modelli di dati devono interagire direttamente con l'archiviazione.

```
class ActiveRecord:
	def __init__(self, data):
		self.data = data

	def save(self):
		print(f"Saving {self.data}")

	def update(self, new_data):
		self.data = new_data
		print(f"Updating to {self.data}")

# Active Record Pattern
record = ActiveRecord("Initial Data")
record.save()
record.update("Updated Data")
record.save()
```

Esempi In Java

1. **Modello di repository**: utilizzato per astrarre la logica di accesso ai dati dal resto dell'applicazione. Questo modello è utile nelle applicazioni con una gestione significativa dei dati.

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

class DataModel {
	private String data;

	public DataModel(String data) {
		this.data = data;
	}

	public String getData() {
		return data;
	}
}

interface Repository {
	void add(String key, DataModel model);
	DataModel get(String key);
```

```
}

class DataRepository implements Repository {
	private Map<String, DataModel> dataSource = new HashMap<>();

	@Override
	public void add(String key, DataModel model) {
		dataSource.put(key, model);
	}

	@Override
	public DataModel get(String key) {
		return dataSource.get(key);
	}
}

public class Main {
	public static void main(String[] args) {
		Repository repository = new DataRepository();
		repository.add("key1", new DataModel("value1"));

		DataModel model = repository.get("key1");
		System.out.println(model.getData());
	}
}
```

2**. Modello del livello di servizio**: questo modello organizza la logica di business in un livello di servizio separato, semplificando la gestione delle operazioni aziendali.

```
class BusinessService {
	private Repository repository;

	public BusinessService(Repository repository) {
		this.repository = repository;
	}

	public String processData(String key) {
		DataModel model = repository.get(key);
		return model.getData().toUpperCase();  // Example
	}
}

public class Main {
	public static void main(String[] args) {
```

```
        Repository repository = new DataRepository();
        repository.add("key1", new DataModel("value1"));

        BusinessService service = new BusinessService(repository);
        String processedData = service.processData("key1");
        System.out.println(processedData);
    }
}
```

3. **Active Record Pattern**: In questo pattern, gli oggetti combinano sia i dati che il comportamento relativo alla loro memorizzazione, tipico dell'ORM (Object-Relational Mapping).

```
class ActiveRecord {
    private String data;

    public ActiveRecord(String data) {
        this.data = data;
    }

    public void save() {
        System.out.println("Saving: " + this.data);
    }

    public void update(String newData) {
        this.data = newData;
        System.out.println("Updated to: " + this.data);
    }
}

public class Main {
    public static void main(String[] args) {
        ActiveRecord record = new ActiveRecord("Initial data");
        record.save();
        record.update("Data updated");
        record.save();
    }
}
```

Capitolo 8: Modelli Funzionali

Esempi In Python

1. **Currying**: questo modello trasforma una funzione che accetta più argomenti in una sequenza di funzioni che accettano un singolo argomento.

```
def curry_add(a):
    def add_b(b):
        return a + b
    return add_b

# Currying
add_five = curry_add(5)
print(add_five(10))  # Print 15
```

2. **Funzione di ordine superiore**: queste funzioni accettano altre funzioni come argomenti o restituiscono funzioni come risultato.

```
def apply_twice(func, value):
    return func(func(value))

def square(x):
    return x * x

# Higher-Order Function
result = apply_twice(square, 3)
print(result)  # Print 81 (3^4)
```

3. **Valutazione lazy**: questo modello ritarda la valutazione di un'espressione fino a quando non è strettamente necessario. È possibile utilizzare i generatori in Python per questo scopo.

```
def lazy_range(n):
    i = 0
    while i < n:
        yield i
        i += 1

# Lazy evaluation
for num in lazy_range(5):
```

```
        print(num)
```

4. **Funzioni pure**: queste funzioni non hanno effetti collaterali e restituiscono gli stessi risultati se chiamate con gli stessi argomenti.

```
def pure_function(a, b):
        return a * b + b

print(pure_function(2, 3))  # Print always 9
```

5. **Map-Reduce**: questo modello viene utilizzato per l'elaborazione di raccolte di dati. La funzione 'map' applica una funzione a ogni elemento di una lista, mentre 'reduce' aggrega i risultati.

```
from functools import reduce

data = [1, 2, 3, 4, 5]
mapped_data = map(lambda x: x * 2, data)  # Dounbles each element
reduced_data = reduce(lambda x, y: x + y, mapped_data)  # Adds everything

print(list(mapped_data))  # [2, 4, 6, 8, 10]
print(reduced_data)  # 30
```

Esempi In Java

1. **Currying**: scomposizione di una funzione che accetta più argomenti in una serie di funzioni che accettano un argomento.

```
import java.util.function.Function;

public class CurryingExample {
        public static void main(String[] args) {
        Function<Integer, Function<Integer, Integer>> curryAdd = a -> b -> a + b;

                Function<Integer, Integer> addFive = curryAdd.apply(5);
                int result = addFive.apply(10);  // 15
                System.out.println(result);
        }
}
```

2. **Funzioni di ordine superiore**: funzioni che accettano altre funzioni come argomenti o restituiscono funzioni come risultato.

```
import java.util.function.Function;

public class HigherOrderFunctionExample {

	public static void main(String[] args) {

		Function<Function<Integer, Integer>, Function<Integer, Integer>> applyTwice
= f -> v -> f.apply(f.apply(v));

		Function<Integer, Integer> square = x -> x * x;

		int result = applyTwice.apply(square).apply(3);  // 81

		System.out.println(result);

	}

}
```

3. **Lazy Evaluation**: posticipare la valutazione di un'espressione fino a quando il suo valore non è necessario. In Java, i flussi forniscono un modo per ottenere una valutazione lazy.

```
import java.util.stream.IntStream;

public class LazyEvaluationExample {
	public static void main(String[] args) {
		IntStream stream = IntStream.range(1, 6).map(x -> {
			System.out.println("Doubling " + x);
			return x * 2;
		});

		System.out.println("Stream created, not evaluated yet.");
		int sum = stream.sum();
		System.out.println("Sum: " + sum);
	}
}
```

4. **Funzioni pure**: funzioni che non hanno effetti collaterali e restituiscono sempre lo stesso risultato per gli stessi input.

```
public class PureFunctionExample {
	public static int pureFunction(int a, int b) {
		return a * b + b;
	}

	public static void main(String[] args) {
		int result = pureFunction(2, 3);  // Always returns 9
		System.out.println(result);
	}
}
```

5. **Map-Reduce**: un modello utilizzato per elaborare raccolte di dati. L'operazione 'map' applica una funzione a ogni elemento di una raccolta e 'reduce' aggrega i risultati.

```
import java.util.Arrays;
import java.util.List;
import java.util.function.BinaryOperator;
import java.util.function.Function;

public class MapReduceExample {
	public static void main(String[] args) {
		List<Integer> data = Arrays.asList(1, 2, 3, 4, 5);

		Function<Integer, Integer> mapper = x -> x * 2;
		BinaryOperator<Integer> reducer = Integer::sum;

		int result = data.stream()
					.map(mapper)
					.reduce(0, reducer);

		System.out.println(result);  // 30
	}
}
```

Capitolo 9: Argomenti Avanzati Nei Design Patterns

Esempi In Python

1. **Microservizi**: l'implementazione di un'architettura di microservizi in Python comporta la creazione di più servizi distribuibili in modo indipendente che comunicano tra loro, spesso tramite HTTP utilizzando API RESTful. Ogni servizio viene in genere eseguito nel proprio processo e si concentra su una specifica funzionalità aziendale. Ecco un semplice esempio per illustrare il concetto:

Supponiamo di creare un'applicazione di e-commerce con due microservizi: uno per la gestione delle informazioni degli utenti e un altro per la gestione dei dettagli del prodotto.

Microservizio 1: Servizio utente

Questo servizio gestirà le informazioni dell'utente. Useremo Flask, un framework Web leggero, per esporre un'API per le operazioni utente.

Innanzitutto, installa Flask:

pip install flask

Crea un file 'user_service.py':

```
from flask import Flask, jsonify, request
app = Flask(__name__)

# In-memory database
users = {
    1: {"name": "Pamela", "age": 25},
    2: {"name": "Tony", "age": 30}
}
@app.route('/user/<int:user_id>', methods=['GET'])
def get_user(user_id):
    user = users.get(user_id)
```

```
    if not user:
    return jsonify({'error': 'User not found'}), 404
    return jsonify(user)

if __name__ == '__main__':
    app.run(port=5000)
```

Questo servizio fornisce un endpoint per recuperare le informazioni utente in base all'ID utente.

Microservizio 2: Servizio del prodotto

Questo servizio gestisce i dettagli del prodotto. Crea un'altra app Flask in un file separato 'product_service.py':

```
from flask import Flask, jsonify

app = Flask(__name__)

# In-memory database
products = {
    1: {"name": "Pen", "price": 2},
    2: {"name": "Pencil", "price": 1}
}

@app.route('/product/<int:product_id>', methods=['GET'])
def get_product(product_id):
    product = products.get(product_id)
    if not product:
    return jsonify({'error': 'Product not found'}), 404
    return jsonify(product)

if __name__ == '__main__':
    app.run(port=6000)
```

Questo servizio fornisce un endpoint per recuperare i dettagli del prodotto in base all'ID prodotto.

Esecuzione dei servizi

Esegui ogni file Python in un terminale diverso per avviare i servizi:

```
python user_service.py
```

```
python product_service.py
```

A questo punto, sono disponibili due microservizi in esecuzione su porte diverse: il servizio utente sulla porta 5000 e il servizio prodotto sulla porta 6000. È possibile accedervi in modo indipendente tramite i rispettivi endpoint.

Nota: questo è un esempio di base. In uno scenario reale, è possibile aggiungere altre funzionalità, ad esempio la connettività del database, l'autenticazione, la registrazione, la gestione degli errori e i meccanismi di comunicazione, ad esempio le code di messaggi per la comunicazione asincrona. Inoltre, per una configurazione di produzione, è probabile che questi servizi vengano containerizzati usando strumenti come Docker e orchestrati con Kubernetes o un sistema simile.

2. **Cloud Computing**: L'implementazione di un'architettura di cloud computing in Python comporta in genere l'interazione con servizi cloud come AWS, Azure o Google Cloud. Ciò include spesso l'uso di SDK forniti da questi provider di servizi cloud per gestire risorse come istanze di calcolo, archiviazione, database e altro ancora.

Ecco un esempio di base di come è possibile utilizzare Python per interagire con i servizi AWS, come EC2 (per la gestione dei server virtuali) e S3 (per lo storage). Utilizzeremo la libreria "boto3", che funge da SDK AWS per Python.

Nota: prima di eseguire questi esempi, è necessario configurare le credenziali AWS sul computer, in genere tramite l'interfaccia a riga di comando di AWS.

Innanzitutto, installa 'boto3':

```
pip install boto3
```

Esempio 1: interazione con Amazon S3

Questo esempio mostra come creare un bucket S3, caricare un file ed elencare i file nel bucket.

```
import boto3

# Initialize a session using Amazon S3
s3 = boto3.resource('s3')

# Create a new S3 bucket
bucket_name = "my-bucket-name"
s3.create_bucket(Bucket=bucket_name)
# Upload a new file
filename = "example.txt"
bucket = s3.Bucket(bucket_name)
bucket.upload_file(filename, filename)

# List files in the bucket
for object in bucket.objects.all():
    print(object.key)
```

Esempio 2: gestione delle istanze EC2

Questo esempio illustra l'avvio e l'arresto di un'istanza EC2.

```
import boto3

# Initialize a session using EC2
ec2 = boto3.resource('ec2')

# Start an EC2 instance
instance_id = 'i-1234567890'
instance = ec2.Instance(instance_id)
instance.start()

print(f"Instance {instance_id} has been started.")

# Stop an EC2 instance
instance.stop()
```

```
print(f"Instance {instance_id} has been stopped.")
```

Questi esempi sono piuttosto semplici e hanno lo scopo di illustrare l'approccio generale. In pratica, le applicazioni cloud spesso comportano interazioni più complesse, come la gestione della sicurezza (ad esempio, i ruoli IAM), la rete (ad esempio, la configurazione VPC), i servizi di database e altro ancora. Inoltre, è possibile utilizzare servizi nativi del cloud come AWS Lambda per l'elaborazione serverless o Amazon RDS per i database gestiti.

Nota sulla sicurezza: prestare sempre attenzione alle risorse cloud. Evitare di incorporare le credenziali direttamente nel codice. Utilizza i ruoli e le policy IAM per gestire l'accesso in modo sicuro e segui sempre le best practice per la sicurezza del cloud.

3. **Modello reattivo e reattivo, osservatore per la reattività**: questo modello è utile per gli scenari in cui un oggetto deve essere in grado di notificare ad altri oggetti le modifiche di stato. Si tratta di un modello fondamentale per la programmazione basata su eventi.

```
class Observer:
    def update(self, message):
        pass

class Subject:
    def __init__(self):
        self.observers = []

    def register(self, observer):
        self.observers.append(observer)

    def notify_all(self, message):
        for observer in self.observers:
            observer.update(message)

class ConcreteObserver(Observer):
    def update(self, message):
        print(f"Received: {message}")

# Example usage
subject = Subject()
observer = ConcreteObserver()
subject.register(observer)
```

```
subject.notify_all("Hello, Observer Pattern!")
```

4**. Reattivo e reattivo, Event Loop con Asyncio per la programmazione reattiva**: La libreria "asyncio" di Python è perfetta per scrivere codice asincrono, che è una pietra miliare della programmazione reattiva.

```
import asyncio

async def reactive_task():
    while True:
        print("Reacting to an event")
        await asyncio.sleep(1)  # Simulate an async operation

async def main():
    task = asyncio.create_task(reactive_task())
    await asyncio.sleep(5)  # Run for 5 seconds
    task.cancel()

asyncio.run(main())
```

5**. Reattivo e reattivo, utilizzando RxPy per estensioni reattive**: ReactiveX, o Rx, è una libreria per la composizione di programmi asincroni e basati su eventi utilizzando sequenze osservabili. 'RxPy' è la sua implementazione Python.

Innanzitutto, installa 'RxPy':

```
pip install rx
```

Ora puoi scrivere qualcosa come:

```
import rx
import rx.operators as ops
from rx.scheduler import NewThreadScheduler

def reactive_stream(observer, scheduler):
    observer.on_next("Reactive Programming in Python")
    observer.on_completed()
```

```
source = rx.create(reactive_stream)
source.pipe(ops.map(lambda s: s.upper())).subscribe(
        on_next=lambda value: print("Received:", value),
        on_completed=lambda: print("Done!"),
        scheduler=NewThreadScheduler()
)
```

Ognuno di questi esempi dimostra un aspetto diverso della programmazione reattiva e reattiva in Python. Il modello Observer consente la velocità di risposta alle modifiche dello stato, 'asyncio' abilita operazioni asincrone non bloccanti e RxPy fornisce strumenti per l'elaborazione di flussi reattivi.

6**. Modello di pipeline AI/ML: In AI e ML**, il modello di pipeline viene comunemente utilizzato per semplificare e organizzare il processo di preparazione dei dati, estrazione di funzionalità, addestramento del modello e previsione. Questo modello aiuta a gestire la sequenza delle fasi di lavorazione in modo efficiente e modulare. Creiamo un esempio in Python usando scikit-learn, una popolare libreria di machine learning:

Innanzitutto, installa scikit-learn:

pip install scikit-learn

Esempio: creazione di una pipeline ML per la classificazione

Questo esempio illustra la creazione di una semplice pipeline di Machine Learning per un'attività di classificazione usando scikit-learn.

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.ensemble import RandomForestClassifier
from sklearn.pipeline import Pipeline
from sklearn.metrics import classification_report

# Load the dataset
```

```
data = load_iris()
X, y = data.data, data.target

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Define the processing steps
# Step 1: Standardize the data
# Step 2: Apply PCA
# Step 3: Train a Random Forest Classifier
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('pca', PCA(n_components=2)),
    ('classifier', RandomForestClassifier())
])

# Train the pipeline on the training data
pipeline.fit(X_train, y_train)

# Make predictions on the test set
predictions = pipeline.predict(X_test)

# Evaluate the model
print(classification_report(y_test, predictions))
```

In questo esempio:

1. Caricato il set di dati Iris, che è un set di dati classico nell'apprendimento automatico.

2. Suddividere i dati in set di training e di test.

3. È stata creata una pipeline con tre passaggi: ridimensionamento dei dati, riduzione della dimensionalità con PCA e training di un classificatore di foreste casuali.

4. Eseguire il training della pipeline sui dati di training e valutarla sui dati di test.

Questo modello è molto efficace per garantire che la sequenza di trasformazioni e il training del modello vengano eseguiti in modo coerente, sia durante il training che durante l'esecuzione di stime. Semplifica inoltre il processo di sperimentazione di diverse fasi di pre-elaborazione e algoritmi.

Esempi In Java

1. **Microservizi**: la creazione di un'architettura di microservizi in Java comporta in genere lo sviluppo di servizi separati, ciascuno responsabile di una funzionalità o di una capacità aziendale distinta, e consente loro di comunicare spesso tramite API RESTful. Ecco un semplice esempio di utilizzo di Spring Boot, un framework popolare per la creazione di microservizi in Java.

Nota: In questo esempio si presuppone una certa familiarità con Spring Boot. In caso contrario, è possibile fare riferimento alla documentazione di Spring Boot per la configurazione e le nozioni di base.

Innanzitutto, dovrai aggiungere la dipendenza web iniziale di Spring Boot al 'pom.xml' del tuo progetto:

```
<dependencies>
        <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
</dependencies>
```

Esempio 1: Servizio utente

Questo microservizio gestirà le operazioni correlate all'utente.

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

@SpringBootApplication
public class UserServiceApplication {

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

    @RestController
    class UserController {
```

```
    @GetMapping("/user/{userId}")
        public User getUser(@PathVariable String userId) {
        // In real-world applications, you'd fetch user data from a database
        return new User(userId, "User 1");
        }
        }

        class User {
        private String id;
        private String name;

        public User(String id, String name) {
        this.id = id;
        this.name = name;
        }

        // Getters and setters...
        }
}
```

Esempio 2: Servizio del prodotto

Questo microservizio gestirà le funzionalità relative al prodotto.

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

@SpringBootApplication
public class ProductServiceApplication {

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

    @RestController
    class ProductController {

@GetMapping("/product/{productId}")
    public Product getProduct(@PathVariable String productId) {
    // In real-world applications, you'd fetch product data from a database
    return new Product(productId, "Widget", 9.99);
    }
```

```
    }

    class Product {
    private String id;
    private String name;
    private double price;

    public Product(String id, String name, double price) {
    this.id = id;
    this.name = name;
    this.price = price;
    }

    // Getters and setters...
    }
}
```

Ognuna di queste classi Java rappresenta un'applicazione Spring Boot separata e insieme formano un'architettura di microservizi di base. Possono essere eseguiti in modo indipendente su porte diverse e comunicare tra loro in base alle esigenze, in genere tramite HTTP/REST.

In uno scenario reale, è possibile migliorare questi servizi con funzionalità aggiuntive come la connettività del database, la sicurezza (ad esempio, OAuth, JWT), la registrazione, la gestione della configurazione, l'individuazione dei servizi (ad esempio, Eureka), i gateway API (ad esempio, Zuul) e altre funzionalità native del cloud. È anche possibile containerizzarli usando Docker e gestirli con Kubernetes per l'orchestrazione negli ambienti di produzione.

2. **Cloud Computing**: L'implementazione di un'architettura di cloud computing in Java spesso comporta l'integrazione con provider di servizi cloud come AWS, Azure o Google Cloud. Ciò include in genere l'uso dei rispettivi SDK per gestire risorse come istanze di calcolo, archiviazione, database e così via. Di seguito sono riportati esempi di base che illustrano come interagire con i servizi AWS utilizzando l'SDK AWS per Java.

Nota: prima di eseguire questi esempi, assicurati di aver installato e configurato l'SDK AWS per Java con le tue credenziali AWS.

Innanzitutto, aggiungi l'SDK Java AWS al tuo "pom.xml" se utilizzi Maven:

```
<dependencies>
  <dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-java-sdk</artifactId>
    <version> 2.23.0</version> <!-- Use the latest version -->
  </dependency>
</dependencies>
```

Esempio 1: interazione con Amazon S3

Questo esempio mostra come creare un bucket Amazon S3, caricare un file ed elencare i file nel bucket.

```
import com.amazonaws.auth.AWSStaticCredentialsProvider;
import com.amazonaws.auth.BasicAWSCredentials;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.services.s3.model.Bucket;
import com.amazonaws.services.s3.model.ObjectListing;
import com.amazonaws.services.s3.model.PutObjectRequest;

import java.io.File;

public class S3Example {
   public static void main(String[] args) {
      BasicAWSCredentials awsCreds = new BasicAWSCredentials("YOUR_ACCESS_KEY",
"YOUR_SECRET_KEY");
      AmazonS3 s3Client = AmazonS3ClientBuilder.standard()
                  .withCredentials(new AWSStaticCredentialsProvider(awsCreds))
                  .withRegion("us-west-2")
                  .build();

      String bucketName = "my-new-bucket";
      s3Client.createBucket(bucketName);

      String fileName = "path/to/file.txt";
      s3Client.putObject(new PutObjectRequest(bucketName, "file.txt", new File(fileName)));

      ObjectListing objectListing = s3Client.listObjects(bucketName);
      objectListing.getObjectSummaries().forEach((objectSummary) -> {
         System.out.println(objectSummary.getKey());
      });
   }
}
```

Esempio 2: gestione delle istanze EC2

Questo esempio illustra l'avvio e l'arresto di un'istanza EC2.

```
import com.amazonaws.auth.AWSStaticCredentialsProvider;
import com.amazonaws.auth.BasicAWSCredentials;
import com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.StartInstancesRequest;
```

```
import com.amazonaws.services.ec2.model.StopInstancesRequest;

public class EC2Example {
   public static void main(String[] args) {
      BasicAWSCredentials awsCreds = new BasicAWSCredentials("ACCESSKEY", "
SECRETKEY");
      AmazonEC2 ec2Client = AmazonEC2ClientBuilder.standard()
                  .withCredentials(new AWSStaticCredentialsProvider(awsCreds))
                  .withRegion("us-west-2")
                  .build();

      String instanceId = "i-1234567890";

      // Start the instance
      StartInstancesRequest startInstancesRequest = new StartInstancesRequest()
         .withInstanceIds(instanceId);
      ec2Client.startInstances(startInstancesRequest);

      // Stop the instance
      StopInstancesRequest stopInstancesRequest = new StopInstancesRequest()
         .withInstanceIds(instanceId);
      ec2Client.stopInstances(stopInstancesRequest);
   }
}
```

L'ID dell'istanza EC2 e i nomi dei bucket S3 sono segnaposto e devono essere sostituiti con valori effettivi rilevanti per l'ambiente AWS.

Ricorda che, per gli ambienti di produzione, è consigliabile utilizzare metodi più sicuri per gestire le credenziali AWS, come i ruoli IAM, invece di codificarle nell'applicazione. Inoltre, questi esempi sono semplicistici; Le applicazioni del mondo reale potrebbero comportare interazioni e considerazioni più complesse, come la gestione degli errori, la sicurezza e la gestione efficiente delle risorse.

3. **Responsive and Reactive, Observer Pattern for Responsive Design**: This pattern is fundamental in event-driven programming where an object (the subject) notifies other objects (observers) of changes in its state.

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

interface Observer {
        void update(String message);
}

class ConcreteObserver implements Observer {
```

```
    public void update(String message) {
        System.out.println("Received: " + message);
    }
}

class Subject {
    private List<Observer> observers = new ArrayList<>();

    public void attach(Observer observer) {
        observers.add(observer);
    }

    public void notifyAllObservers(String message) {
        for (Observer observer : observers) {
            observer.update(message);
        }
    }
}

public class ObserverPatternDemo {
    public static void main(String[] args) {
        Subject subject = new Subject();
        Observer observer = new ConcreteObserver();
        subject.attach(observer);

        subject.notifyAllObservers("Hello, Observer Pattern!");
    }
}
```

4. **Reattivo e reattivo, CompletableFuture per la programmazione reattiva**: 'CompletableFuture' in Java 8+ consente di scrivere codice asincrono non bloccante. È una base per la programmazione reattiva.

```
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;

public class CompletableFutureDemo {
    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        CompletableFuture<String> completableFuture =
CompletableFuture.supplyAsync(() -> {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
```

```
                }
                return "Result of the asynchronous computation";
            });

            String result = completableFuture.get(); // Blocks until the future is complete
            System.out.println(result);
        }
    }
```

5. **API Responsive and Reactive, Reactive Streams (Java 9+):** questa API fornisce uno standard per l'elaborazione asincrona dei flussi con backpressure non bloccante.

```
    import java.util.concurrent.Flow.*;
    import java.util.concurrent.SubmissionPublisher;

    public class ReactiveStreamDemo {
        public static void main(String[] args) throws InterruptedException {
            try (SubmissionPublisher<String> publisher = new SubmissionPublisher<>())
    {
                publisher.subscribe(new Subscriber<>() {
                    public void onSubscribe(Subscription subscription) {
                        subscription.request(1);
                    }

                    public void onNext(String item) {
                        System.out.println("Received: " + item);
                    }

                    public void onError(Throwable throwable) {
                        throwable.printStackTrace();
                    }

                    public void onComplete() {
                        System.out.println("Completed");
                    }
                });

                publisher.submit("Hello, Reactive Streams!");
                Thread.sleep(1000); // Waiting for the subscriber to process the data
            }
        }
    }
```

Questi esempi illustrano l'implementazione di design patterns reattivi e reattivi in Java. Il modello Observer è ideale per gli scenari in cui le modifiche in un oggetto devono essere propagate ad altri. 'CompletableFuture' e l'API Reactive Streams forniscono le basi per la creazione di applicazioni non bloccanti, asincrone e reattive.

6. **Pipeline AI/ML: l'implementazione di un** modello di pipeline per AI e ML in Java comporta in genere la creazione di una sequenza di fasi di elaborazione dei dati, in cui l'output di un passaggio è l'input del successivo. Ciò può essere particolarmente utile per le attività di Machine Learning in cui sono presenti una serie di trasformazioni e una fase di training/previsione del modello. Si consideri un esempio che usa la popolare libreria di machine learning MLlib di Apache Spark in Java.

Nota: nell'esempio seguente si presuppone che Apache Spark e la relativa libreria MLlib siano configurati. Il processo di configurazione per questi strumenti esula dall'ambito di questo esempio.

Innanzitutto, è necessario aggiungere la dipendenza per MLlib di Spark al Maven 'pom.xml':

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-mllib_3</artifactId>
  <version>3.3.4</version> <!-- Use the latest version -->
</dependency>
```

Esempio: pipeline ML per la classificazione

In questo esempio verrà creata una pipeline semplice per un'attività di classificazione:

```
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.ml.Pipeline;
import org.apache.spark.ml.PipelineModel;
import org.apache.spark.ml.PipelineStage;
import org.apache.spark.ml.classification.LogisticRegression;
import org.apache.spark.ml.feature.HashingTF;
import org.apache.spark.ml.feature.Tokenizer;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SparkSession;

public class MlPipelineExample {
  public static void main(String[] args) {
    // Initialize Spark
    SparkConf conf = new SparkConf().setAppName("ML Pipeline
Example").setMaster("local");
    JavaSparkContext sc = new JavaSparkContext(conf);
```

```
        SparkSession spark = SparkSession.builder().sparkContext(sc.sc()).getOrCreate();

        // Sample data
        Dataset<Row> trainingData = ... // Load or create training data

        // Configure a tokenizer, hashingTF, and lr
        Tokenizer tokenizer = new Tokenizer().setInputCol("text").setOutputCol("words");
        HashingTF hashingTF = new
    HashingTF().setNumFeatures(1000).setInputCol("words").setOutputCol("features");
        LogisticRegression lr = new LogisticRegression().setMaxIter(10).setRegParam(0.001);

        Pipeline pipeline = new Pipeline().setStages(new PipelineStage[]{tokenizer, hashingTF,
    lr});

        // Fit the pipeline to training data
        PipelineModel model = pipeline.fit(trainingData);

        // Now you can use the model to predict on new data
        Dataset<Row> testData = ... // Load or create test data
        Dataset<Row> predictions = model.transform(testData);

        // Evaluate predictions, etc.

        // Stop Spark
        spark.stop();
    }
}
```

In questo esempio:

- Abbiamo configurato un'applicazione Spark di base.

- Definiamo una pipeline con tre fasi: un 'Tokenizer' per dividere il testo in parole, un 'HashingTF' per convertire la sequenza di parole in un vettore di funzionalità e un 'LogisticRegression' per la classificazione.

- Usiamo la pipeline per adattare un modello ai dati di training.

- Successivamente, utilizziamo il modello per generare previsioni su nuovi dati.

Questo modello di pipeline è particolarmente potente nei contesti di Machine Learning in quanto consente la composizione pulita di diverse fasi di elaborazione e analisi dei dati. Ogni passaggio è chiaramente definito e la pipeline nel suo complesso può essere facilmente compresa, modificata ed estesa.