Software Entwicklung 1

Jil Zerndt, Lucien Perret
January 2025

Examples

Adapter Pattern

Szenario: Altbestand an Drittanbieter-Bibliothek integrieren

```java
// Bestehende Schnittstelle
interface ModernPrinter {
    void printDocument(String content);
}

// Alte Drittanbieter-Klasse
class LegacyPrinter {
    public void print(String[] pages) {
        for(String page : pages) {
            System.out.println(page);
        }
    }
}

// Adapter
class PrinterAdapter implements ModernPrinter {
    private LegacyPrinter legacyPrinter;

    public PrinterAdapter(LegacyPrinter
        printer) {
        this.legacyPrinter = printer;
    }

    public void printDocument(String content) {
        String[] pages = content.split("\n");
        legacyPrinter.print(pages);
    }
}
```

Simple Factory

Szenario: Erzeugung von verschiedenen Datenbankverbindungen

```java
class DatabaseFactory {
    public static Database
        createDatabase(String type) {
        switch(type) {
            case "MySQL":
                return new MySQLDatabase();
            case "PostgreSQL":
                return new PostgreSQLDatabase();
            default:
                throw new
                    IllegalArgumentException("Unknow
                    DB type");
        }
    }
}

// Verwendung
Database db =
    DatabaseFactory.createDatabase("MySQL");
```

Singleton

Szenario: Globale Konfigurationsverwaltung

```java
public class Configuration {
    private static Configuration instance;
    private Map<String, String> config;

    private Configuration() {
        config = new HashMap<>();
    }

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

Dependency Injection

Szenario: Flexible Logger-Implementation

```java
interface Logger {
    void log(String message);
}

class FileLogger implements Logger {
    public void log(String message) {
        // Log to file
    }
}

class UserService {
    private final Logger logger;

    public UserService(Logger logger) { //
        Dependency Injection
        this.logger = logger;
    }
}
```

Proxy

Szenario: Verzögertes Laden eines großen Bildes

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

class RealImage implements Image {
    private String filename;

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

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

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

class ImageProxy implements Image {
    private RealImage realImage;
    private String filename;

    public ImageProxy(String filename) {
        this.filename = filename;
    }

    public void display() {
        if(realImage == null) {
            realImage = new RealImage(filename);
        }
        realImage.display();
    }
}
```

Chain of Responsibility

Szenario: Authentifizierungskette

```java
abstract class AuthHandler {
    protected AuthHandler next;

    public void setNext(AuthHandler next) {
        this.next = next;
    }

    public abstract boolean handle(String
        username, String password);
}

class LocalAuthHandler extends AuthHandler {
    public boolean handle(String username,
        String password) {
        if(checkLocalDB(username, password)) {
            return true;
        }
        return next != null ?
            next.handle(username, password) :
            false;
    }
}

class LDAPAuthHandler extends AuthHandler {
    public boolean handle(String username,
        String password) {
        if(checkLDAP(username, password)) {
            return true;
        }
        return next != null ?
            next.handle(username, password) :
            false;
    }
}
```

Decorator

Szenario: Dynamische Erweiterung eines Text-Editors

```java
interface TextComponent {
    String render();
}

class SimpleText implements TextComponent {
    private String text;

    public SimpleText(String text) {
        this.text = text;
    }

    public String render() {
        return text;
    }
}

class BoldDecorator implements TextComponent {
    private TextComponent component;

    public BoldDecorator(TextComponent
        component) {
        this.component = component;
    }

    public String render() {
        return "<b>" + component.render() +
            "</b>";
    }
}
```

Observer

Szenario: News-Benachrichtigungssystem

```java
interface NewsObserver {
    void update(String news);
}

class NewsAgency {
    private List<NewsObserver> observers = new
        ArrayList<>();

    public void addObserver(NewsObserver
        observer) {
        observers.add(observer);
    }

    public void notifyObservers(String news) {
        for(NewsObserver observer : observers) {
            observer.update(news);
        }
    }
}

class NewsChannel implements NewsObserver {
    private String name;

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

    public void update(String news) {
        System.out.println(name + " received: "
            + news);
    }
}
```

Strategy

Szenario: Verschiedene Zahlungsmethoden

```java
interface PaymentStrategy {
    void pay(int amount);
}

class CreditCardPayment implements
    PaymentStrategy {
    private String cardNumber;

    public void pay(int amount) {
        System.out.println("Paid " + amount + "
            using Credit Card");
    }
}

class PayPalPayment implements PaymentStrategy {
    private String email;

    public void pay(int amount) {
        System.out.println("Paid " + amount + "
            using PayPal");
    }
}
```

Strategy Pattern Implementation

```java
public interface SortStrategy {
    void sort(List<String> data);
}

public class QuickSort implements SortStrategy {
    public void sort(List<String> data) {
        // Implementierung
    }
}

public class Context {
    private SortStrategy strategy;

    public void setStrategy(SortStrategy strategy) {
        this.strategy = strategy;
    }

    public void executeStrategy(List<String> data) {
        strategy.sort(data);
    }
}
```

Composite

Szenario: Dateisystem-Struktur

```java
interface FileSystemComponent {
    void list(String prefix);
}

class File implements FileSystemComponent {
    private String name;

    public void list(String prefix) {
        System.out.println(prefix + name);
    }
}

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

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

    public void list(String prefix) {
        System.out.println(prefix + name);
        for(FileSystemComponent child :
            children) {
            child.list(prefix + "  ");
        }
    }
}
```

State

Szenario: Verkaufsautomat

```java
interface VendingMachineState {
    void insertCoin();
    void ejectCoin();
    void selectProduct();
    void dispense();
}

class HasCoinState implements
    VendingMachineState {
    private VendingMachine machine;

    public void selectProduct() {
        System.out.println("Product selected");
        machine.setState(machine.getSoldState());
    }

    public void insertCoin() {
        System.out.println("Already have coin");
    }
}

class VendingMachine {
    private VendingMachineState currentState;

    public void setState(VendingMachineState
        state) {
        this.currentState = state;
    }

    public void insertCoin() {
        currentState.insertCoin();
    }
}
```

Visitor

Szenario: Dokumentstruktur mit verschiedenen Operationen

```java
interface DocumentElement {
    void accept(Visitor visitor);
}

interface Visitor {
    void visit(Paragraph paragraph);
    void visit(Heading heading);
}

class HTMLVisitor implements Visitor {
    public void visit(Paragraph p) {
        System.out.println("<p>" + p.getText()
            + "</p>");
    }

    public void visit(Heading h) {
        System.out.println("<h1>" + h.getText()
            + "</h1>");
    }
}
```

Facade

Szenario: Vereinfachte Multimedia-Bibliothek

```java
class MultimediaFacade {
    private AudioSystem audio;
    private VideoSystem video;
    private SubtitleSystem subtitles;

    public void playMovie(String movie) {
        audio.initialize();
        video.initialize();
        subtitles.load(movie);
        video.play(movie);
        audio.play();
    }
}
```

Abstract Factory

Szenario: GUI-Elemente für verschiedene Betriebssysteme

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

class WindowsFactory implements GUIFactory {
    public Button createButton() {
        return new WindowsButton();
    }

    public Checkbox createCheckbox() {
        return new WindowsCheckbox();
    }
}

class MacFactory implements GUIFactory {
    public Button createButton() {
        return new MacButton();
    }

    public Checkbox createCheckbox() {
        return new MacCheckbox();
    }
}
```

Factory Method Implementation

Aufgabe: Implementieren Sie eine Factory für verschiedene Dokumenttypen (PDF, Word, Text)

Lösung:

```java
// Interface fuer Produkte
interface Document {
    void open();
    void save();
}

// Konkrete Produkte
class PdfDocument implements Document {
    public void open() { /* ... */ }
    public void save() { /* ... */ }
}

// Factory Method Pattern
abstract class DocumentCreator {
    abstract Document createDocument();

    // Template Method
    final void processDocument() {
        Document doc = createDocument();
        doc.open();
        doc.save();
    }
}

// Konkrete Factory
class PdfDocumentCreator extends
    DocumentCreator {
    Document createDocument() {
        return new PdfDocument();
    }
}
```

Observer Pattern Implementation

Aufgabe: Implementieren Sie ein Benachrichtigungssystem für Aktienkurse

Lösung:

```java
interface StockObserver {
    void update(String stock, double price);
}

class StockMarket {
    private List<StockObserver> observers = new
        ArrayList<>();

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

    public void notifyObservers(String stock,
        double price) {
        for(StockObserver observer : observers)
            {
            observer.update(stock, price);
        }
    }
}

class StockDisplay implements StockObserver {
    public void update(String stock, double
        price) {
        System.out.println("Stock: " + stock +
                          " updated to " +
                             price);
    }
}
```

Extract Method Refactoring

Vorher:

```java
void printOwing() {
    printBanner();

    // calculate outstanding
    double outstanding = 0.0;
    for (Order order : orders) {
        outstanding += order.getAmount();
    }

    // print details
    System.out.println("name: " + name);
    System.out.println("amount: " + outstanding);
}
```

Nachher:

```java
void printOwing() {
    printBanner();
    double outstanding = calculateOutstanding();
    printDetails(outstanding);
}

double calculateOutstanding() {
    double result = 0.0;
    for (Order order : orders) {
        result += order.getAmount();
    }
    return result;
}

void printDetails(double outstanding) {
    System.out.println("name: " + name);
    System.out.println("amount: " + outstanding);
}
```

Unit Test

Zu testende Klasse:

```java
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }
}
```

Test:

```java
@Test
public class CalculatorTest {
    private Calculator calc;

    @Before
    public void setup() {
        calc = new Calculator();
    }

    @Test
    public void testAdd() {
        assertEquals(4, calc.add(2, 2));
        assertEquals(0, calc.add(-2, 2));
        assertEquals(-4, calc.add(-2, -2));
    }
}
```

BDD Test

Feature File:

```
Feature: Calculator Addition
  Scenario: Add two positive numbers
    Given I have a calculator
    When I add 2 and 2
    Then the result should be 4

  Scenario: Add positive and negative numbers
    Given I have a calculator
    When I add -2 and 2
    Then the result should be 0
```

Step Definitions:

```java
public class CalculatorSteps {
    private Calculator calc;
    private int result;

    @Given("I have a calculator")
    public void createCalculator() {
        calc = new Calculator();
    }

    @When("I add {int} and {int}")
    public void addNumbers(int a, int b) {
        result = calc.add(a, b);
    }

    @Then("the result should be {int}")
    public void checkResult(int expected) {
        assertEquals(expected, result);
    }
}
```

Client-Server Implementation

Aufgabe: Implementieren Sie einen einfachen Echo-Server mit Java.

Lösung:

```java
// Server
public class EchoServer {
    public static void main(String[] args) {
        try (ServerSocket server = new
             ServerSocket(8080)) {
            while (true) {
                Socket client = server.accept();
                new Thread(() ->
                      handleClient(client)).start();
            }
        }
    }

    private static void handleClient(Socket client) {
        try (
            BufferedReader in = new BufferedReader(
                new
                      InputStreamReader(client.getInputStrea
            PrintWriter out = new PrintWriter(
                client.getOutputStream(), true)
        ) {
            String line;
            while ((line = in.readLine()) != null) {
                out.println("Echo: " + line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

// Client
public class EchoClient {
    public static void main(String[] args) {
        try (
            Socket socket = new Socket("localhost",
                 8080);
            PrintWriter out = new PrintWriter(
                socket.getOutputStream(), true);
            BufferedReader in = new BufferedReader(
                new
                      InputStreamReader(socket.getInputStrea
        ) {
            out.println("Hello Server!");
            System.out.println(in.readLine());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Publish-Subscribe Pattern

Aufgabe: Implementieren Sie ein einfaches Event-System.

Lösung:

```java
public class EventBus {
    private Map<String, List<EventHandler>> handlers =
         new HashMap<>();

    public void subscribe(String event, EventHandler
         handler) {
        handlers.computeIfAbsent(event, k -> new
             ArrayList<>())
                .add(handler);
    }

    public void publish(String event, String data) {
        if (handlers.containsKey(event)) {
            handlers.get(event)
                   .forEach(handler ->
                        handler.handle(data));
        }
    }
}

interface EventHandler {
    void handle(String data);
}

// Verwendung
EventBus bus = new EventBus();
bus.subscribe("userLogin", data ->
    System.out.println("User logged in: " + data));
bus.publish("userLogin", "john_doe");
```

JDBC Basisbeispiel

```java
import java.sql.*;

public class DbTest {
    public static void main(String[] args)
            throws SQLException {
        // Verbindung aufbauen
        Connection con = DriverManager.getConnection(
            "jdbc:postgresql://test.zhaw.ch/testdb",
            "user", "password");

        // Statement erstellen und ausfuehren
        Statement stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery(
            "SELECT * FROM test ORDER BY name");

        // Ergebnisse verarbeiten
        while (rs.next()) {
            System.out.println(
                "Name: " + rs.getString("name"));
        }

        // Aufraeumen
        rs.close();
        stmt.close();
        con.close();
    }
}
```

DAO Implementation

```java
public interface ArticleDAO {
    void insert(Article item);
    void update(Article item);
    void delete(Article item);
    Article findById(int id);
    Collection<Article> findAll();
    Collection<Article> findByName(String name);
}

public class Article {
    private long id;
    private String name;
    private float price;

    // Getter/Setter
}

public class JdbcArticleDAO implements
    ArticleDAO {
    private Connection conn;

    public void insert(Article item) {
        PreparedStatement stmt =
            conn.prepareStatement(
                "INSERT INTO articles (name, price)
                    VALUES (?, ?)");
        stmt.setString(1, item.getName());
        stmt.setFloat(2, item.getPrice());
        stmt.executeUpdate();
    }
    // weitere Implementierungen
}
```

Parent-Child Beziehung mit JPA

```java
@Entity
public class Department {
    @Id @GeneratedValue
    private Long id;

    private String name;

    @OneToMany(mappedBy = "department")
    private List<Employee> employees;
}

@Entity
public class Employee {
    @Id @GeneratedValue
    private Long id;

    @ManyToOne
    @JoinColumn(name = "department_id")
    private Department department;

    private String name;
    private double salary;
}
```

Spring Data Repository

```java
@Repository
public interface SaleRepository
        extends CrudRepository<Sale, String> {

    List<Sale> findOrderByDateTime();

    List<Sale> findByDateTime(
        final LocalDateTime dateTime);
}

@Service
public class ProcessSaleHandler {
    private final ProductDescriptionRepository catalog;
    private final SaleRepository saleRepository;

    @Transactional
    public void endSale() {
        assert(currentSale != null
            && !currentSale.isComplete());
        this.currentSale.becomeComplete();
        this.saleRepository.save(currentSale);
    }
}
```

Abstract Factory: POS Terminal

```java
public interface IJavaPOSDevicesFactory {
    CashDrawer getNewCashDrawer();
    CoinDispenser getNewCoinDispenser();
    // weitere Methoden
}

public class IBMJavaPOSDevicesFactory
        implements IJavaPOSDevicesFactory {
    public CashDrawer getNewCashDrawer() {
        return new com.ibm.pos.jpos.CashDrawer();
    }
    // weitere Implementierungen
}
```

Command: Persistenz

```java
public interface ICommand {
    void execute();
    void undo();
}

public class DBUpdateCommand implements ICommand {
    private PersistentObject object;

    public void execute() {
        // Update in Datenbank
    }

    public void undo() {
        // Aenderung rueckgaengig machen
    }
}
```

Template Method: GUI Framework

```java
public abstract class GUIComponent {
    // Template Method
    public final void update() {
        clearBackground();
        repaint(); // Hook Method
    }

    protected abstract void repaint();
}

public class MyButton extends GUIComponent {
    protected void repaint() {
        // Button-spezifische Implementation
    }
}
```

Spring Data Repository

```java
@Repository
public interface UserRepository
        extends JpaRepository<User, Long> {
    // Methode wird automatisch implementiert
    List<User> findByLastNameOrderByFirstNameAsc(
        String lastName);

    // SQL-Query via Annotation
    @Query("SELECT u FROM User u WHERE u.active =
         true")
    List<User> findActiveUsers();
}
```