```
/**
 * Property-Based-Testing-Starter mit jqwik
 * Praktische Beispiele für Business-Logic-Testing
 *
 * Von Dr. Cassian Holt - Java Fleet Systems Consulting
 * Basiert auf 50+ realen Property-Based-Testing-Implementierungen
 */

//
=========================================================================
=====
// DEPENDENCIES (pom.xml)
//
=========================================================================
=====
/*
<dependency>
    <groupId>net.jqwik</groupId>
    <artifactId>jqwik</artifactId>
    <version>1.8.2</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.assertj</groupId>
    <artifactId>assertj-core</artifactId>
    <version>3.24.2</version>
    <scope>test</scope>
</dependency>
*/

//
=========================================================================
=====
// 1. BASIC PROPERTY-BASED TESTING PATTERNS
//
=========================================================================
=====

import net.jqwik.api.*;
import net.jqwik.api.constraints.*;
import org.assertj.core.api.Assertions;
import static org.assertj.core.api.Assertions.*;

/**
 * Starter-Beispiele für Property-Based Testing
 * Diese Tests demonstrieren fundamentale Eigenschaften, die immer gelten sollten
 */
class PropertyBasedTestingStarter {

    //
=========================================================================
=
```

```
    // MATHEMATISCHE EIGENSCHAFTEN
    //
==========================================================================
=

    /**
     * Kommutativität: a + b = b + a
     * Eine der einfachsten und wichtigsten Eigenschaften
     */
    @Property
    void additionIsCommutative(@ForAll int a, @ForAll int b) {
        assertThat(a + b).isEqualTo(b + a);
    }

    /**
     * Assoziativität: (a + b) + c = a + (b + c)
     */
    @Property
    void additionIsAssociative(@ForAll int a, @ForAll int b, @ForAll int c) {
        assertThat((a + b) + c).isEqualTo(a + (b + c));
    }

    /**
     * Inverses Element: Für jede Operation sollte eine Umkehrung existieren
     */
    @Property
    void listReverseIsInverse(@ForAll List<Integer> list) {
        List<Integer> reversed = reverse(reverse(list));
        assertThat(reversed).isEqualTo(list);
    }

    private List<Integer> reverse(List<Integer> list) {
        List<Integer> result = new ArrayList<>(list);
        Collections.reverse(result);
        return result;
    }

    //
==========================================================================
=
    // BUSINESS-LOGIC EIGENSCHAFTEN
    //
==========================================================================
=

    /**
     * E-Commerce: Warenkorb-Eigenschaften
     * Warenkorb-Gesamtpreis sollte immer Summe der Einzelpreise sein
     */
    @Property
    void shoppingCartTotalShouldEqualSumOfItems(@ForAll("validShoppingCarts") ShoppingCart
cart) {
```

```
        BigDecimal expectedTotal = cart.getItems().stream()
            .map(Item::getPrice)
            .reduce(BigDecimal.ZERO, BigDecimal::add);

        assertThat(cart.getTotal()).isEqualTo(expectedTotal);
    }

    /**
     * Banking: Transfer-Eigenschaften
     * Geld verschwindet nie - Sender verliert genau das, was Empfänger erhält
     */
    @Property
    void moneyTransferConservesTotal(@ForAll("validAccounts") Account sender,
                                   @ForAll("validAccounts") Account receiver,
                                   @ForAll @Positive BigDecimal amount) {
        // Assumption: Sender hat genug Geld
        Assume.that(sender.getBalance().compareTo(amount) >= 0);

        BigDecimal initialTotal = sender.getBalance().add(receiver.getBalance());

        TransferService.transfer(sender, receiver, amount);

        BigDecimal finalTotal = sender.getBalance().add(receiver.getBalance());
        assertThat(finalTotal).isEqualTo(initialTotal);
    }

    //
===========================================================================
=
    // VALIDATION UND SECURITY EIGENSCHAFTEN
    //
===========================================================================
=

    /**
     * Input Validation: Gültige Emails bleiben nach Normalisierung gültig
     */
    @Property
    void validEmailsRemainValidAfterNormalization(@ForAll("validEmails") String email) {
        String normalized = EmailUtils.normalize(email);
        assertThat(EmailValidator.isValid(normalized)).isTrue();
    }

    /**
     * Security: Passwort-Hashing ist deterministisch aber nicht umkehrbar
     */
    @Property
    void passwordHashingIsDeterministic(@ForAll @AlphaChars @StringLength(min = 8, max =
50) String password) {
        String hash1 = PasswordHasher.hash(password);
        String hash2 = PasswordHasher.hash(password);
```

```
        // Deterministisch: Gleiche Eingabe → Gleiche Ausgabe
        assertThat(hash1).isEqualTo(hash2);

        // Nicht umkehrbar: Hash sollte niemals Original-Password enthalten
        assertThat(hash1).doesNotContain(password);

        // Verifizierung sollte funktionieren
        assertThat(PasswordHasher.verify(password, hash1)).isTrue();
    }

    //
==============================================================================
=
    // PERFORMANCE UND SCALING EIGENSCHAFTEN
    //
==============================================================================
=

    /**
     * Performance: Algorithmus-Komplexität sollte erwartbar sein
     */
    @Property
    void sortingPerformanceScalesReasonably(@ForAll @Size(max = 1000) List<Integer> list) {
        long startTime = System.nanoTime();
        List<Integer> sorted = QuickSort.sort(new ArrayList<>(list));
        long duration = System.nanoTime() - startTime;

        // O(n log n) im Average Case - großzügige Obergrenze
        long expectedMaxDuration = list.size() * log2(list.size() + 1) * 1000; // nanoseconds

        assertThat(duration).isLessThan(expectedMaxDuration);
        assertThat(isSorted(sorted)).isTrue();
    }

    private boolean isSorted(List<Integer> list) {
        for (int i = 1; i < list.size(); i++) {
            if (list.get(i - 1) > list.get(i)) return false;
        }
        return true;
    }

    private long log2(long n) {
        return (long)(Math.log(n) / Math.log(2));
    }

    //
==============================================================================
=
    // DATA PROVIDERS - Realistische Test-Daten generieren
    //
==============================================================================
=
```

```
    @Provide
    Arbitrary<ShoppingCart> validShoppingCarts() {
        return Arbitraries.create(() -> {
            ShoppingCart cart = new ShoppingCart();
            // 1-10 Items im Warenkorb
            int itemCount = Arbitraries.integers().between(1, 10).sample();

            for (int i = 0; i < itemCount; i++) {
                Item item = validItems().sample();
                cart.addItem(item);
            }

            return cart;
        });
    }

    @Provide
    Arbitrary<Item> validItems() {
        return Combinators.combine(
            Arbitraries.strings().alpha().ofMinLength(1).ofMaxLength(50), // Name
            Arbitraries.bigDecimals()
                .between(BigDecimal.valueOf(0.01), BigDecimal.valueOf(999.99))
                .ofScale(2) // Preis mit 2 Dezimalstellen
        ).as(Item::new);
    }

    @Provide
    Arbitrary<Account> validAccounts() {
        return Combinators.combine(
            Arbitraries.strings().numeric().ofLength(10), // Account Number
            Arbitraries.bigDecimals()
                .between(BigDecimal.ZERO, BigDecimal.valueOf(100000))
                .ofScale(2) // Balance
        ).as(Account::new);
    }

    @Provide
    Arbitrary<String> validEmails() {
        return Combinators.combine(
            // Local part: 1-30 alphanumeric characters
            Arbitraries.strings().alpha().ofMinLength(1).ofMaxLength(30),
            // Domain: Realistic domains
            Arbitraries.of("gmail.com", "yahoo.com", "hotmail.com", "company.de", "test.org")
        ).as((local, domain) -> local + "@" + domain);
    }

    //
==============================================================================
=
    // ERWEITERTE PATTERNS
```

```
    //
================================================================================
=

    /**
     * Stateful Testing: Warenkorb-Operationen in beliebiger Reihenfolge
     */
    @Property
    void shoppingCartOperationsAreConsistent(@ForAll("cartOperations") List<CartOperation>
operations) {
        ShoppingCart cart = new ShoppingCart();
        BigDecimal runningTotal = BigDecimal.ZERO;

        for (CartOperation operation : operations) {
            switch (operation.getType()) {
                case ADD_ITEM:
                    cart.addItem(operation.getItem());
                    runningTotal = runningTotal.add(operation.getItem().getPrice());
                    break;
                case REMOVE_ITEM:
                    if (cart.containsItem(operation.getItem())) {
                        cart.removeItem(operation.getItem());
                        runningTotal = runningTotal.subtract(operation.getItem().getPrice());
                    }
                    break;
                case CLEAR:
                    cart.clear();
                    runningTotal = BigDecimal.ZERO;
                    break;
            }

            // Invariante: Cart Total ist immer korrekt
            assertThat(cart.getTotal()).isEqualTo(runningTotal);
        }
    }

    @Provide
    Arbitrary<List<CartOperation>> cartOperations() {
        return Arbitraries.sequences(
            Arbitraries.create(() -> {
                CartOperationType type = Arbitraries.of(CartOperationType.values()).sample();
                Item item = validItems().sample();
                return new CartOperation(type, item);
            })
        ).ofMinSize(1).ofMaxSize(20);
    }

    /**
     * Contract Testing: API-Eigenschaften zwischen Services
     */
    @Property
    void userServiceContractProperties(@ForAll("validUsers") User user) {
```

```
        // Idempotenz: Mehrfaches Speichern desselben Users sollte gleiche ID ergeben
        Long id1 = userService.save(user);
        Long id2 = userService.save(user); // Gleicher User
        assertThat(id1).isEqualTo(id2);

        // Referentielle Integrität: Gespeicherte User können geladen werden
        User retrieved = userService.findById(id1);
        assertThat(retrieved.getEmail()).isEqualTo(user.getEmail());

        // Konsistenz: Email-Updates werden korrekt persistiert
        String newEmail = "updated@example.com";
        userService.updateEmail(id1, newEmail);
        User updated = userService.findById(id1);
        assertThat(updated.getEmail()).isEqualTo(newEmail);
    }

    //
================================================================================
=
    // ERROR-HANDLING PROPERTIES
    //
================================================================================
=

    /**
     * Robustness: Service sollte bei ungültigen Inputs nicht crashen
     */
    @Property
    void paymentServiceHandlesInvalidInputsGracefully(@ForAll String invalidInput) {
        // Test mit allen möglichen Strings als Input
        assertThatCode(() -> {
            PaymentResult result = paymentService.processPayment(invalidInput);
            // Service sollte nicht crashen, sondern Error-Result zurückgeben
            if (result.isError()) {
                assertThat(result.getErrorMessage()).isNotBlank();
            }
        }).doesNotThrowAnyException();
    }

    /**
     * Boundary Testing: Service verhält sich an Grenzen korrekt
     */
    @Property
    void discountCalculationHandlesBoundaryValues(@ForAll @BigRange(min = "0.00", max =
"10000.00") BigDecimal amount,
                                                  @ForAll @IntRange(min = 0, max = 100) int discountPercent) {

        BigDecimal discount = discountService.calculateDiscount(amount, discountPercent);

        // Discount sollte nie negativ sein
        assertThat(discount).isGreaterThanOrEqualTo(BigDecimal.ZERO);
```

```
        // Discount sollte nie größer als Original-Amount sein
        assertThat(discount).isLessThanOrEqualTo(amount);

        // Bei 0% Discount sollte Discount 0 sein
        if (discountPercent == 0) {
            assertThat(discount).isEqualTo(BigDecimal.ZERO);
        }

        // Bei 100% Discount sollte Discount = Amount sein
        if (discountPercent == 100) {
            assertThat(discount).isEqualTo(amount);
        }
    }

    //
=============================================================================
=
    // INTEGRATION PROPERTIES
    //
=============================================================================
=

    /**
     * Database Integration: CRUD-Operationen sind konsistent
     */
    @Property
    void crudOperationsAreConsistent(@ForAll("validProducts") Product product) {
        // Create
        Long id = productRepository.save(product);
        assertThat(id).isNotNull();

        // Read
        Product retrieved = productRepository.findById(id);
        assertThat(retrieved.getName()).isEqualTo(product.getName());
        assertThat(retrieved.getPrice()).isEqualTo(product.getPrice());

        // Update
        BigDecimal newPrice = product.getPrice().add(BigDecimal.TEN);
        productRepository.updatePrice(id, newPrice);
        Product updated = productRepository.findById(id);
        assertThat(updated.getPrice()).isEqualTo(newPrice);

        // Delete
        productRepository.delete(id);
        assertThat(productRepository.exists(id)).isFalse();
    }

    @Provide
    Arbitrary<Product> validProducts() {
        return Combinators.combine(
            Arbitraries.strings().alpha().ofMinLength(1).ofMaxLength(100), // Name
```

```
            Arbitraries.bigDecimals().between(BigDecimal.ONE,
BigDecimal.valueOf(9999)).ofScale(2), // Price
            Arbitraries.of(ProductCategory.values()), // Category
            Arbitraries.integers().between(0, 1000) // Stock
        ).as(Product::new);
    }

    //
=======================================================================
=
    // DOMAIN-SPECIFIC PROPERTIES
    //
=======================================================================
=

    /**
     * E-Commerce Domain: Bestellungs-Workflow-Eigenschaften
     */
    @Property
    void orderWorkflowMaintainsInvariants(@ForAll("validOrders") Order order) {
        // Zustandsübergänge sollten logisch sein
        OrderStateMachine stateMachine = new OrderStateMachine(order);

        // CREATED -> PAID sollte immer möglich sein bei gültiger Zahlung
        if (order.getStatus() == OrderStatus.CREATED && order.getPayment() != null) {
            assertThat(stateMachine.canTransitionTo(OrderStatus.PAID)).isTrue();
        }

        // SHIPPED -> CREATED sollte niemals möglich sein
        if (order.getStatus() == OrderStatus.SHIPPED) {
            assertThat(stateMachine.canTransitionTo(OrderStatus.CREATED)).isFalse();
        }

        // CANCELLED Orders sollten keine weiteren Übergänge haben
        if (order.getStatus() == OrderStatus.CANCELLED) {
            for (OrderStatus status : OrderStatus.values()) {
                if (status != OrderStatus.CANCELLED) {
                    assertThat(stateMachine.canTransitionTo(status)).isFalse();
                }
            }
        }
    }

    /**
     * Financial Domain: Währungskonvertierung-Eigenschaften
     */
    @Property
    void currencyConversionProperties(@ForAll @BigRange(min = "0.01", max = "999999.99")
BigDecimal amount,
                                      @ForAll("validCurrencyPairs") CurrencyPair pair) {

        BigDecimal converted = currencyService.convert(amount, pair);
```

```java
        // Konvertierung sollte nie null oder negativ sein
        assertThat(converted).isNotNull().isPositive();

        // Rund-Trip-Konvertierung (mit Toleranz für Rundungsfehler)
        CurrencyPair reversePair = pair.reverse();
        BigDecimal roundTrip = currencyService.convert(converted, reversePair);

        // 0.1% Toleranz für Rundungsfehler
        BigDecimal tolerance = amount.multiply(BigDecimal.valueOf(0.001));
        BigDecimal difference = amount.subtract(roundTrip).abs();

        assertThat(difference).isLessThanOrEqualTo(tolerance);
    }

    @Provide
    Arbitrary<CurrencyPair> validCurrencyPairs() {
        List<Currency> currencies = Arrays.asList(
            Currency.USD, Currency.EUR, Currency.GBP, Currency.CHF, Currency.JPY
        );

        return Combinators.combine(
            Arbitraries.of(currencies),
            Arbitraries.of(currencies)
        ).as(CurrencyPair::new)
         .filter(pair -> !pair.getFrom().equals(pair.getTo())); // Keine Selbst-Konvertierung
    }

    //
===========================================================================
=
    // PERFORMANCE UND RESOURCE PROPERTIES
    //
===========================================================================
=

    /**
     * Memory Management: Operationen sollten kein Memory Leak haben
     */
    @Property
    void operationsDoNotLeakMemory(@ForAll @Size(max = 100) List<String> data) {
        // Baseline Memory
        System.gc();
        long baselineMemory = getUsedMemory();

        // Operation ausführen
        DataProcessor processor = new DataProcessor();
        List<ProcessedData> results = processor.processAll(data);

        // Results verwenden um Compiler-Optimierung zu verhindern
        assertThat(results).hasSize(data.size());
```

```
        // Cleanup
        results.clear();
        processor.cleanup();
        System.gc();

        // Memory sollte ungefähr auf Baseline zurück sein (10% Toleranz)
        long finalMemory = getUsedMemory();
        long memoryIncrease = finalMemory - baselineMemory;
        long tolerance = baselineMemory / 10; // 10% Toleranz

        assertThat(memoryIncrease).isLessThanOrEqualTo(tolerance);
    }

    private long getUsedMemory() {
        Runtime runtime = Runtime.getRuntime();
        return runtime.totalMemory() - runtime.freeMemory();
    }

    /**
     * Concurrency: Thread-Safety-Eigenschaften
     */
    @Property
    void sharedResourceIsThreadSafe(@ForAll @Size(min = 2, max = 10) List<String> operations)
throws InterruptedException {
        ThreadSafeCounter counter = new ThreadSafeCounter();
        ExecutorService executor = Executors.newFixedThreadPool(operations.size());
        List<Future<?>> futures = new ArrayList<>();

        // Parallele Operations ausführen
        for (String operation : operations) {
            Future<?> future = executor.submit(() -> {
                if (operation.startsWith("increment")) {
                    counter.increment();
                } else {
                    counter.decrement();
                }
            });
            futures.add(future);
        }

        // Warten bis alle fertig sind
        for (Future<?> future : futures) {
            future.get(); // Kann InterruptedException werfen
        }

        executor.shutdown();

        // Counter sollte consistent sein
        long expectedValue = operations.stream()
            .mapToLong(op -> op.startsWith("increment") ? 1 : -1)
            .sum();
```

```
        assertThat(counter.getValue()).isEqualTo(expectedValue);
    }

    //
=================================================================================
=
    // REGRESSION UND GOLDEN MASTER PROPERTIES
    //
=================================================================================
=

    /**
     * Golden Master: Komplexe Berechnungen sollten konsistent bleiben
     */
    @Property
    void complexCalculationResultsAreStable(@ForAll("realWorldScenarios") BusinessScenario
scenario) {
        // Berechnung ausführen
        ComplexBusinessCalculation calculation = new ComplexBusinessCalculation();
        BusinessResult result = calculation.calculate(scenario);

        // Golden Master Vergleich (hier vereinfacht)
        String resultHash = createResultHash(result);

        // In echten Tests würdest du ApprovalTests oder ähnliches verwenden
        // ApprovalTests.verify(result);

        // Grundlegende Konsistenz-Checks
        assertThat(result).isNotNull();
        assertThat(result.isValid()).isTrue();

        // Gleiche Eingabe sollte gleiche Ausgabe produzieren (Determinismus)
        BusinessResult secondResult = calculation.calculate(scenario);
        assertThat(createResultHash(secondResult)).isEqualTo(resultHash);
    }

    private String createResultHash(BusinessResult result) {
        // Vereinfachter Hash für Demonstration
        return String.valueOf(result.hashCode());
    }

    @Provide
    Arbitrary<BusinessScenario> realWorldScenarios() {
        return Combinators.combine(
            validCustomers(),
            validProducts().list().ofMinSize(1).ofMaxSize(10),
            Arbitraries.of(Season.values()),
            Arbitraries.booleans() // Premium customer
        ).as(BusinessScenario::new);
    }

    @Provide
```

```
    Arbitrary<Customer> validCustomers() {
        return Combinators.combine(
            Arbitraries.strings().alpha().ofMinLength(2).ofMaxLength(50), // Name
            validEmails(), // Email
            Arbitraries.integers().between(18, 100), // Age
            Arbitraries.of(CustomerType.values()) // Type
        ).as(Customer::new);
    }

    //
==========================================================================
=
    // UTILITY METHODS UND HELPER
    //
==========================================================================
=

    /**
     * Hilfsmethode: Test-Data-Konsistenz prüfen
     */
    private void assertValidTestData(Object data) {
        assertThat(data).isNotNull();
        // Weitere generische Validierungen je nach Domain
    }

    /**
     * Hilfsmethode: Performance-Messungen
     */
    private void assertReasonablePerformance(Runnable operation, Duration maxDuration) {
        Instant start = Instant.now();
        operation.run();
        Instant end = Instant.now();

        Duration actualDuration = Duration.between(start, end);
        assertThat(actualDuration).isLessThan(maxDuration);
    }

    /**
     * Hilfsmethode: Floating-Point-Vergleiche mit Toleranz
     */
    private void assertApproximatelyEqual(BigDecimal actual, BigDecimal expected, BigDecimal
tolerance) {
        BigDecimal difference = actual.subtract(expected).abs();
        assertThat(difference).isLessThanOrEqualTo(tolerance);
    }
}

//
==========================================================================
=====
// SUPPORTING CLASSES (normalerweise in separaten Dateien)
```

```
// ==============================================================================

// Business Domain Classes
class ShoppingCart {
    private List<Item> items = new ArrayList<>();

    public void addItem(Item item) { items.add(item); }
    public void removeItem(Item item) { items.remove(item); }
    public void clear() { items.clear(); }
    public boolean containsItem(Item item) { return items.contains(item); }
    public List<Item> getItems() { return new ArrayList<>(items); }

    public BigDecimal getTotal() {
        return items.stream()
            .map(Item::getPrice)
            .reduce(BigDecimal.ZERO, BigDecimal::add);
    }
}

class Item {
    private final String name;
    private final BigDecimal price;

    public Item(String name, BigDecimal price) {
        this.name = name;
        this.price = price;
    }

    public String getName() { return name; }
    public BigDecimal getPrice() { return price; }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Item)) return false;
        Item item = (Item) o;
        return Objects.equals(name, item.name) && Objects.equals(price, item.price);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, price);
    }
}

class Account {
    private final String accountNumber;
    private BigDecimal balance;

    public Account(String accountNumber, BigDecimal balance) {
```

```
        this.accountNumber = accountNumber;
        this.balance = balance;
    }

    public String getAccountNumber() { return accountNumber; }
    public BigDecimal getBalance() { return balance; }

    public void debit(BigDecimal amount) {
        this.balance = this.balance.subtract(amount);
    }

    public void credit(BigDecimal amount) {
        this.balance = this.balance.add(amount);
    }
}

// Service Classes
class TransferService {
    public static void transfer(Account from, Account to, BigDecimal amount) {
        from.debit(amount);
        to.credit(amount);
    }
}

class EmailUtils {
    public static String normalize(String email) {
        return email.toLowerCase().trim();
    }
}

class EmailValidator {
    public static boolean isValid(String email) {
        return email != null && email.contains("@") && email.contains(".");
    }
}

class PasswordHasher {
    public static String hash(String password) {
        // Vereinfachter Hash für Demo
        return "hash_" + password.hashCode();
    }

    public static boolean verify(String password, String hash) {
        return hash.equals(hash(password));
    }
}

// Enum Classes
enum CartOperationType { ADD_ITEM, REMOVE_ITEM, CLEAR }
enum CustomerType { REGULAR, VIP, STUDENT }
enum ProductCategory { ELECTRONICS, CLOTHING, BOOKS, HOME }
enum OrderStatus { CREATED, PAID, SHIPPED, DELIVERED, CANCELLED }
```

```java
enum Currency { USD, EUR, GBP, CHF, JPY }
enum Season { SPRING, SUMMER, AUTUMN, WINTER }

// Operation Classes
class CartOperation {
    private final CartOperationType type;
    private final Item item;

    public CartOperation(CartOperationType type, Item item) {
        this.type = type;
        this.item = item;
    }

    public CartOperationType getType() { return type; }
    public Item getItem() { return item; }
}

// Complex Domain Classes
class Product {
    private final String name;
    private final BigDecimal price;
    private final ProductCategory category;
    private final int stock;

    public Product(String name, BigDecimal price, ProductCategory category, int stock) {
        this.name = name;
        this.price = price;
        this.category = category;
        this.stock = stock;
    }

    // Getters
    public String getName() { return name; }
    public BigDecimal getPrice() { return price; }
    public ProductCategory getCategory() { return category; }
    public int getStock() { return stock; }
}

class Order {
    private final Long id;
    private OrderStatus status;
    private final Payment payment;

    public Order(Long id, OrderStatus status, Payment payment) {
        this.id = id;
        this.status = status;
        this.payment = payment;
    }

    public Long getId() { return id; }
    public OrderStatus getStatus() { return status; }
    public Payment getPayment() { return payment; }
```

```
    public void setStatus(OrderStatus status) { this.status = status; }
}

class Payment {
    private final BigDecimal amount;
    private final String method;

    public Payment(BigDecimal amount, String method) {
        this.amount = amount;
        this.method = method;
    }

    public BigDecimal getAmount() { return amount; }
    public String getMethod() { return method; }
}

class Customer {
    private final String name;
    private final String email;
    private final int age;
    private final CustomerType type;

    public Customer(String name, String email, int age, CustomerType type) {
        this.name = name;
        this.email = email;
        this.age = age;
        this.type = type;
    }

    public String getName() { return name; }
    public String getEmail() { return email; }
    public int getAge() { return age; }
    public CustomerType getType() { return type; }
}

class User {
    private final String email;
    private final String name;

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

    public String getEmail() { return email; }
    public String getName() { return name; }
}

// Complex Business Classes
class OrderStateMachine {
    private final Order order;
```

```
    public OrderStateMachine(Order order) {
        this.order = order;
    }

    public boolean canTransitionTo(OrderStatus newStatus) {
        // Vereinfachte State-Machine-Logik
        OrderStatus current = order.getStatus();

        switch (current) {
            case CREATED:
                return newStatus == OrderStatus.PAID || newStatus == OrderStatus.CANCELLED;
            case PAID:
                return newStatus == OrderStatus.SHIPPED || newStatus == OrderStatus.CANCELLED;
            case SHIPPED:
                return newStatus == OrderStatus.DELIVERED;
            case DELIVERED:
            case CANCELLED:
                return false;
            default:
                return false;
        }
    }
}

class CurrencyPair {
    private final Currency from;
    private final Currency to;

    public CurrencyPair(Currency from, Currency to) {
        this.from = from;
        this.to = to;
    }

    public Currency getFrom() { return from; }
    public Currency getTo() { return to; }

    public CurrencyPair reverse() {
        return new CurrencyPair(to, from);
    }
}

class BusinessScenario {
    private final Customer customer;
    private final List<Product> products;
    private final Season season;
    private final boolean isPremium;

    public BusinessScenario(Customer customer, List<Product> products, Season season, boolean
isPremium) {
        this.customer = customer;
        this.products = products;
        this.season = season;
```

```
        this.isPremium = isPremium;
    }

    public Customer getCustomer() { return customer; }
    public List<Product> getProducts() { return products; }
    public Season getSeason() { return season; }
    public boolean isPremium() { return isPremium; }
}

class BusinessResult {
    private final boolean valid;
    private final BigDecimal value;

    public BusinessResult(boolean valid, BigDecimal value) {
        this.valid = valid;
        this.value = value;
    }

    public boolean isValid() { return valid; }
    public BigDecimal getValue() { return value; }

    @Override
    public int hashCode() {
        return Objects.hash(valid, value);
    }
}

// Service Implementations (Stubs für Demo)
class QuickSort {
    public static List<Integer> sort(List<Integer> list) {
        Collections.sort(list);
        return list;
    }
}

class DiscountService {
    public BigDecimal calculateDiscount(BigDecimal amount, int percentage) {
        return amount.multiply(BigDecimal.valueOf(percentage)).divide(BigDecimal.valueOf(100));
    }
}

class CurrencyService {
    public BigDecimal convert(BigDecimal amount, CurrencyPair pair) {
        // Vereinfachte Konvertierung für Demo
        // In Realität würde hier ein echter Exchange-Rate-Service verwendet
        return amount.multiply(getExchangeRate(pair));
    }

    private BigDecimal getExchangeRate(CurrencyPair pair) {
        // Mock Exchange Rates
        if (pair.getFrom() == Currency.EUR && pair.getTo() == Currency.USD) {
            return new BigDecimal("1.10");
```

```java
        }
        if (pair.getFrom() == Currency.USD && pair.getTo() == Currency.EUR) {
            return new BigDecimal("0.91");
        }
        return BigDecimal.ONE; // Fallback
    }
}

class DataProcessor {
    public List<ProcessedData> processAll(List<String> data) {
        return data.stream()
            .map(ProcessedData::new)
            .collect(Collectors.toList());
    }

    public void cleanup() {
        // Cleanup resources
    }
}

class ProcessedData {
    private final String data;

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

    public String getData() { return data; }
}

class ThreadSafeCounter {
    private final AtomicLong value = new AtomicLong(0);

    public void increment() { value.incrementAndGet(); }
    public void decrement() { value.decrementAndGet(); }
    public long getValue() { return value.get(); }
}

class ComplexBusinessCalculation {
    public BusinessResult calculate(BusinessScenario scenario) {
        // Vereinfachte komplexe Berechnung
        BigDecimal value = scenario.getProducts().stream()
            .map(Product::getPrice)
            .reduce(BigDecimal.ZERO, BigDecimal::add);

        if (scenario.isPremium()) {
            value = value.multiply(new BigDecimal("0.9")); // 10% Rabatt
        }

        return new BusinessResult(true, value);
    }
}
```

```
// Repository Interfaces (für Demo als Stubs)
class ProductRepository {
    private static Map<Long, Product> storage = new ConcurrentHashMap<>();
    private static AtomicLong idGenerator = new AtomicLong(1);

    public Long save(Product product) {
        Long id = idGenerator.getAndIncrement();
        storage.put(id, product);
        return id;
    }

    public Product findById(Long id) {
        return storage.get(id);
    }

    public void updatePrice(Long id, BigDecimal newPrice) {
        Product product = storage.get(id);
        if (product != null) {
            Product updated = new Product(product.getName(), newPrice, product.getCategory(),
product.getStock());
            storage.put(id, updated);
        }
    }

    public void delete(Long id) {
        storage.remove(id);
    }

    public boolean exists(Long id) {
        return storage.containsKey(id);
    }
}

// Service Stubs
class UserService {
    public Long save(User user) { return 1L; }
    public User findById(Long id) { return new User("test@example.com", "Test User"); }
    public void updateEmail(Long id, String email) { }
}

class PaymentService {
    public PaymentResult processPayment(String input) {
        // Vereinfachte Validierung
        if (input == null || input.trim().isEmpty()) {
            return PaymentResult.error("Invalid input");
        }
        return PaymentResult.success();
    }
}

class PaymentResult {
```

```
    private final boolean success;
    private final String errorMessage;

    private PaymentResult(boolean success, String errorMessage) {
        this.success = success;
        this.errorMessage = errorMessage;
    }

    public static PaymentResult success() {
        return new PaymentResult(true, null);
    }

    public static PaymentResult error(String message) {
        return new PaymentResult(false, message);
    }

    public boolean isSuccess() { return success; }
    public boolean isError() { return !success; }
    public String getErrorMessage() { return errorMessage; }
}
```