Software Entwicklung 1

Jil Zerndt, Lucien Perret
January 2025

Softwarearchitektur und Design

Modulkonzept und Architekturmuster

Clean Architecture Implementation

```java
// Enterprise Business Rules (Entity)
public class Order {
    private List<OrderItem> items;

    public Money calculateTotal() {
        return items.stream()
                    .map(OrderItem::getSubtotal)
                    .reduce(Money.ZERO,
                        Money::add);
    }
}

// Application Business Rules (Use Case)
public class CreateOrderUseCase {
    private OrderRepository repository;

    public OrderId execute(CreateOrderCommand
        cmd) {
        Order order = new Order(cmd.getItems());
        validateOrder(order);
        return repository.save(order);
    }
}

// Interface Adapters
public class OrderController {
    private CreateOrderUseCase useCase;

    public OrderResponse
        createOrder(OrderRequest req) {
        CreateOrderCommand cmd =
            mapToCommand(req);
        OrderId id = useCase.execute(cmd);
        return new OrderResponse(id);
    }
}
```

Design Patterns und UML-Modellierung

GRASP in der Praxis

```java
// Information Expert: Order kennt seine Details
public class Order {
    private List<OrderLine> lines;

    public Money calculateTotal() {
        return lines.stream()
                    .map(OrderLine::getSubtotal)
                    .reduce(Money.ZERO,
                        Money::add);
    }
}

// Controller: Koordiniert Use Case
public class OrderController {
    private OrderService service;

    public OrderResponse
        createOrder(OrderRequest request) {
        // Koordination der Verarbeitung
        Order order =
            service.createOrder(request);
        return OrderResponse.from(order);
    }
}

// Protected Variations: Abstraktion von
    Implementierung
public interface PaymentGateway {
    PaymentResult process(Money amount);
}

public class StripePaymentGateway implements
    PaymentGateway {
    public PaymentResult process(Money amount) {
        // Stripe-spezifische Implementierung
    }
}
```

Architekturstile und Patterns

Schichtenarchitektur (Layered Architecture)

Strukturierung eines Systems in horizontale Schichten:

Typische Schichten:

- Präsentationsschicht (UI)
- Anwendungsschicht (Application)
- Geschäftslogikschicht (Domain)
- Datenzugriffsschicht (Persistence)

Regeln:

- Kommunikation nur mit angrenzenden Schichten
- Abhängigkeiten nur nach unten
- Jede Schicht kapselt ihre Implementierung

Schichtenarchitektur Implementation

Beispiel einer typischen Schichtenstruktur:

```java
// Presentation Layer
public class CustomerController {
    private CustomerService service;

    public CustomerDTO getCustomer(String id) {
        Customer customer = service.findById(id);
        return CustomerDTO.from(customer);
    }
}

// Application Layer
public class CustomerService {
    private CustomerRepository repository;

    public Customer findById(String id) {
        validateId(id);
        return repository.findById(id)
            .orElseThrow(CustomerNotFoundException::new);
    }
}

// Domain Layer
public class Customer {
    private CustomerId id;
    private String name;
    private Address address;

    public void updateAddress(Address newAddress) {
        validateAddress(newAddress);
        this.address = newAddress;
    }
}

// Persistence Layer
public class CustomerRepository {
    private JpaRepository<Customer, CustomerId>
        jpaRepo;

    public Optional<Customer> findById(String id) {
        return jpaRepo.findById(new CustomerId(id));
    }
}
```

Client-Server Architektur

Verteilung von Funktionalitäten zwischen Client und Server:

Charakteristiken:

- Klare Trennung von Zuständigkeiten
- Zentralisierte Ressourcenverwaltung
- Skalierbarkeit durch Server-Erweiterung
- Verschiedene Client-Typen möglich

Varianten:

- Thin Client: Minimale Client-Logik
- Rich Client: Komplexe Client-Funktionalität
- Web Client: Browser-basiert
- Mobile Client: Für mobile Geräte optimiert

Architektur-Evaluation: Performance

Szenario: Online-Shop während Black Friday

Analyse:

- **Last-Annahmen:**
  - 10.000 gleichzeitige Nutzer
  - 1.000 Bestellungen pro Minute
  - 100.000 Produktaufrufe pro Minute
- **Architektur-Maßnahmen:**
  - Caching-Strategie für Produkte
  - Load Balancing für Anfragen
  - Asynchrone Bestellverarbeitung
  - Datenbank-Replikation
- **Monitoring:**
  - Response-Zeiten
  - Server-Auslastung
  - Cache-Hit-Rate
  - Fehlerraten

```java
// Performance-optimierte Produktabfrage
@Cacheable(value = "products")
public ProductDTO getProduct(String id) {
    ProductDTO product = cache.get(id);
    if (product == null) {
        product = repository.findById(id)
                            .map(this::toDTO)
                            .orElseThrow();
        cache.put(id, product);
    }
    return product;
}
```

Architekturmuster (Patterns)

Architekturmuster Übersicht

Grundlegende Architekturmuster für verteilte Systeme:

- **Layered Pattern:** Schichtenarchitektur
- **Client-Server Pattern:** Verteilte Dienste
- **Master-Slave Pattern:** Verteilte Verarbeitung
- **Pipe-Filter Pattern:** Datenstromverarbeitung
- **Broker Pattern:** Vermittler zwischen Endpunkten
- **Event-Bus Pattern:** Nachrichtenverteilung
- **MVC Pattern:** Trennung von Daten und Darstellung

Clean Architecture

Prinzipien nach Robert C. Martin:

- **Unabhängigkeit von Frameworks**
  - Framework als Tool, nicht als Einschränkung
  - Geschäftslogik unabhängig von UI/DB
- **Testbarkeit**
  - Business Rules ohne externe Systeme testbar
  - Keine DB/UI für Tests notwendig
- **Unabhängigkeit von UI**
  - UI austauschbar ohne Business Logic Änderung
  - Web, Desktop, Mobile möglich
- **Unabhängigkeit von Datenbank**
  - DB-System austauschbar
  - Business Rules unabhängig von Datenpersistenz

Schichten von außen nach innen:

1. Frameworks & Drivers (UI, DB, External Interfaces)
2. Interface Adapters (Controllers, Presenters)
3. Application Business Rules (Use Cases)
4. Enterprise Business Rules (Entities)

Layered Pattern

Struktur:

```java
// Presentation Layer
public class CustomerUI {
    private CustomerService service;
    public void showCustomerDetails(int id) {
        Customer customer = service.getCustomer(id);
        // display logic
    }
}

// Business Layer
public class CustomerService {
    private CustomerRepository repository;
    public Customer getCustomer(int id) {
        return repository.findById(id);
    }
}

// Data Layer
public class CustomerRepository {
    public Customer findById(int id) {
        // database access
        return customer;
    }
}
```

Vorteile:

- Klare Trennung der Verantwortlichkeiten
- Austauschbarkeit einzelner Schichten
- Einfachere Wartung und Tests

Clean Architecture Implementation

Strukturbeispiel für einen Online-Shop:

```java
// Enterprise Business Rules (Entities)
public class Order {
    private List<OrderItem> items;
    private OrderStatus status;

    public Money calculateTotal() {
        return items.stream()
                    .map(OrderItem::getSubtotal)
                    .reduce(Money.ZERO, Money::add);
    }
}

// Application Business Rules (Use Cases)
public class CreateOrderUseCase {
    private OrderRepository repository;
    private PaymentGateway paymentGateway;

    public OrderId execute(CreateOrderCommand command) {
        Order order = new Order(command.getItems());
        PaymentResult result = paymentGateway.process(
            order.calculateTotal());
        if (result.isSuccessful()) {
            return repository.save(order);
        }
        throw new PaymentFailedException();
    }
}

// Interface Adapters
public class OrderController {
    private CreateOrderUseCase createOrderUseCase;

    public OrderResponse createOrder(OrderRequest request) {
        CreateOrderCommand command = mapToCommand(request);
        OrderId id = createOrderUseCase.execute(command);
        return new OrderResponse(id);
    }
}
```

Microservices Architektur

Grundprinzipien:

- Unabhängig deploybare Services
- Lose Kopplung
- Eigene Datenhaltung pro Service
- REST/Message-basierte Kommunikation

Vorteile:

- Bessere Skalierbarkeit
- Unabhängige Entwicklung
- Technologiefreiheit
- Robustheit

Herausforderungen:

- Verteilte Transaktionen
- Service Discovery
- Datenkonvergenz
- Monitoring

Microservice Design

Service für Benutzerprofile:

```java
@RestController
@RequestMapping("/api/users")
public class UserProfileController {
    private final UserService userService;

    @GetMapping("/{id}")
    public UserProfileDTO getProfile(@PathVariable
         String id) {
        UserProfile profile = userService.findById(id);
        return UserProfileDTO.from(profile);
    }

    @PutMapping("/{id}")
    public ResponseEntity<Void> updateProfile(
            @PathVariable String id,
            @RequestBody UpdateProfileCommand command)
                {
        userService.updateProfile(id, command);
        return ResponseEntity.ok().build();
    }
}

// Event fuer andere Services
public class UserProfileUpdatedEvent {
    private final String userId;
    private final String newEmail;
    private final LocalDateTime timestamp;

    // Konstruktor und Getter
}
```

Microservices Design Prinzipien

1. Service Boundaries
- Nach Business Capabilities trennen
- Bounded Context (DDD) beachten
- Datenhoheit festlegen

2. Service Kommunikation
- Synchron vs. Asynchron
- Event-Driven Design
- API Gateway Pattern

3. Datenmanagement
- Database per Service
- Event Sourcing
- CQRS Pattern

4. Resilience
- Circuit Breaker
- Bulkhead Pattern
- Fallback Mechanismen

Design Patterns in der Architektur

Model-View-Controller (MVC)

Trennt Anwendung in drei Hauptkomponenten:
- **Model:** Geschäftslogik und Daten
- **View:** Darstellung der Daten
- **Controller:** Steuerung und Koordination

```java
// Model
public class CustomerModel {
    private String name;
    private List<Order> orders;

    public void addOrder(Order order) {
        orders.add(order);
        notifyViews();
    }
}

// View
public class CustomerView {
    private CustomerModel model;

    public void displayCustomerInfo() {
        System.out.println("Customer: " +
            model.getName());
        System.out.println("Orders: " +
            model.getOrders().size());
    }
}

// Controller
public class CustomerController {
    private CustomerModel model;
    private CustomerView view;

    public void createOrder(OrderData data) {
        Order order = new Order(data);
        model.addOrder(order);
        view.displayCustomerInfo();
    }
}
```

Event-Driven Architecture (EDA)

Basiert auf der Produktion, Erkennung und Reaktion auf Events:

Komponenten:
- Event Producer
- Event Channel
- Event Consumer
- Event Processor

```java
// Event Definition
public class OrderCreatedEvent {
    private final String orderId;
    private final LocalDateTime timestamp;
    private final BigDecimal totalAmount;
}

// Event Producer
public class OrderService {
    private EventBus eventBus;

    public void createOrder(OrderData data) {
        Order order = orderRepository.save(data);
        OrderCreatedEvent event = new
            OrderCreatedEvent(
            order.getId(),
            LocalDateTime.now(),
            order.getTotalAmount()
        );
        eventBus.publish(event);
    }
}

// Event Consumer
@EventListener
public class InvoiceGenerator {
    public void handleOrderCreated(OrderCreatedEvent
        event) {
        generateInvoice(event.getOrderId());
    }
}
```

Architektur-Dokumentation

1. Überblick
- Systemkontext
- Hauptkomponenten
- Technologie-Stack

2. Architektur-Entscheidungen
- Begründungen
- Alternativen
- Trade-offs

3. Technische Konzepte
- Persistenz
- Sicherheit
- Integration
- Deployment

4. Qualitätsszenarien
- Performance
- Skalierbarkeit
- Verfügbarkeit
- Wartbarkeit

Architektur-Dokumentation: REST API

API-Design und Dokumentation:

```java
@RestController
@RequestMapping("/api/v1/orders")
public class OrderController {

    @GetMapping("/{id}")
    @Operation(summary = "Get order by ID",
               description = "Returns detailed order
                   information")
    @ApiResponses({
        @ApiResponse(responseCode = "200",
                     description = "Order found"),
        @ApiResponse(responseCode = "404",
                     description = "Order not found")
    })
    public OrderDTO getOrder(@PathVariable String id) {
        return orderService.findById(id)
                           .map(OrderDTO::from)
                           .orElseThrow(OrderNotFoundExcepti
    }
}
```

Qualitätsszenarien:

- Response Time $<$ 200ms (95. Perzentil)
- Verfügbarkeit 99.9
- Maximal 1000 req/s pro Instance
- Automatische Skalierung ab 70

Integrationsmuster

Integration Patterns

Muster für die Integration von Systemen und Services:

Hauptkategorien:

- **File Transfer:** Datenaustausch über Dateien
- **Shared Database:** Gemeinsame Datenbasis
- **Remote Procedure Call:** Direkter Methodenaufruf
- **Messaging:** Nachrichtenbasierte Kommunikation

Messaging Pattern Implementation

Message Producer und Consumer:

```java
// Message Definition
public class OrderMessage {
    private String orderId;
    private String customerId;
    private BigDecimal amount;
    private OrderStatus status;
}

// Message Producer
public class OrderProducer {
    private MessageQueue messageQueue;

    public void sendOrderCreated(Order order) {
        OrderMessage message = new OrderMessage(
            order.getId(),
            order.getCustomerId(),
            order.getAmount(),
            OrderStatus.CREATED
        );
        messageQueue.send("orders", message);
    }
}

// Message Consumer
public class OrderProcessor {
    @MessageListener(queue = "orders")
    public void processOrder(OrderMessage message) {
        if (message.getStatus() ==
            OrderStatus.CREATED) {
            processNewOrder(message);
        }
    }

    private void processNewOrder(OrderMessage message)
        {
        // Verarbeitung der Bestellung
        validateOrder(message);
        updateInventory(message);
        notifyCustomer(message);
    }
}
```

API Gateway Pattern

Zentraler Einstiegspunkt für Client-Anfragen:

Verantwortlichkeiten:

- Routing von Anfragen
- Authentifizierung/Autorisierung
- Last-Verteilung
- Caching
- Monitoring
- API-Versionierung

```java
@Component
public class ApiGateway {
    private final AuthService authService;
    private final ServiceRegistry registry;

    @GetMapping("/api/v1/**")
    public ResponseEntity<Object> routeRequest(
            HttpServletRequest request,
            @RequestHeader("Authorization") String
                token) {

        // Authentifizierung
        if (!authService.validateToken(token)) {
            return ResponseEntity.status(401).build();
        }

        // Service Discovery
        String serviceName =
            extractServiceName(request);
        ServiceInstance instance =
            registry.getInstance(serviceName);

        // Request Weiterleitung
        return forwardRequest(instance, request);
    }
}
```

API Design Best Practices

1. **Ressourcen-Orientierung**
   - Klare Ressourcen-Namen
   - Hierarchische Struktur
   - Korrekte HTTP-Methoden
2. **Versionierung**
   - Explizite Versions-Nummer
   - Abwärtskompatibilität
   - Migrations-Strategie
3. **Fehlerbehandlung**
   - Standardisierte Fehler-Formate
   - Aussagekräftige Fehlermeldungen
   - Korrekte HTTP-Status-Codes
4. **Dokumentation**
   - OpenAPI/Swagger
   - Beispiele und Use Cases
   - Fehlerszenarien

REST API Design

Ressourcen-Design für E-Commerce System:

```java
@RestController
@RequestMapping("/api/v1")
public class ProductController {

    // Collection Resource
    @GetMapping("/products")
    public PagedResponse<ProductDTO> getProducts(
            @RequestParam(defaultValue = "0") int page,
            @RequestParam(defaultValue = "20") int
                size) {
        return productService.findAll(page, size);
    }

    // Single Resource
    @GetMapping("/products/{id}")
    public ProductDTO getProduct(@PathVariable String
        id) {
        return productService.findById(id);
    }

    // Sub-Resource Collection
    @GetMapping("/products/{id}/reviews")
    public List<ReviewDTO> getProductReviews(
            @PathVariable String id) {
        return reviewService.findByProductId(id);
    }

    // Error Handling
    @ExceptionHandler(ProductNotFoundException.class)
    public ResponseEntity<ErrorResponse>
        handleNotFound(
            ProductNotFoundException ex) {
        ErrorResponse error = new ErrorResponse(
            "PRODUCT_NOT_FOUND",
            ex.getMessage()
        );
        return ResponseEntity.status(404).body(error);
    }
}
```

Design Pattern Anwendung

Systematisches Vorgehen bei der Pattern-Auswahl:

1. Problem analysieren

- Kernproblem identifizieren
- Qualitätsanforderungen beachten
- Kontext verstehen

2. Pattern auswählen

- Passende Pattern-Kategorie wählen
- Alternativen evaluieren
- Trade-offs abwägen

3. Pattern implementieren

- Struktur übertragen
- An Kontext anpassen
- Auf GRASP-Prinzipien achten

Factory Method Pattern

Problem: Objekterzeugung soll flexibel und erweiterbar sein.

```java
// Creator
public abstract class DocumentCreator {
    public abstract Document createDocument();

    public void openDocument() {
        Document doc = createDocument();
        doc.open();
    }
}

// Concrete Creator
public class PDFDocumentCreator extends
    DocumentCreator {
    @Override
    public Document createDocument() {
        return new PDFDocument();
    }
}

// Product Interface
public interface Document {
    void open();
    void save();
}

// Concrete Product
public class PDFDocument implements Document {
    @Override
    public void open() {
        // PDF-spezifische Implementation
    }

    @Override
    public void save() {
        // PDF-spezifische Implementation
    }
}
```

Strategy Pattern

Problem: Algorithmus soll zur Laufzeit austauschbar sein.

```java
// Strategy Interface
public interface PaymentStrategy {
    void pay(Money amount);
}

// Concrete Strategies
public class CreditCardStrategy implements
    PaymentStrategy {
    private String cardNumber;

    @Override
    public void pay(Money amount) {
        // Kreditkarten-Zahlung
    }
}

public class PayPalStrategy implements
    PaymentStrategy {
    private String email;

    @Override
    public void pay(Money amount) {
        // PayPal-Zahlung
    }
}

// Context
public class ShoppingCart {
    private PaymentStrategy paymentStrategy;

    public void
        setPaymentStrategy(PaymentStrategy
        strategy) {
        this.paymentStrategy = strategy;
    }

    public void checkout(Money amount) {
        paymentStrategy.pay(amount);
    }
}
```

Observer Pattern

Problem: Objekte sollen über Änderungen informiert werden.

```java
// Observer Interface
public interface OrderObserver {
    void onOrderStateChange(Order order);
}

// Concrete Observer
public class EmailNotifier implements
    OrderObserver {
    @Override
    public void onOrderStateChange(Order order)
        {
        sendEmail(order.getCustomer(),
                  "Order status: " +
                      order.getStatus());
    }
}

// Observable
public class Order {
    private List<OrderObserver> observers = new
        ArrayList<>();
    private OrderStatus status;

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

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

    private void notifyObservers() {
        observers.forEach(o ->
            o.onOrderStateChange(this));
    }
}
```

UML Modellierung im Design

Einsatz verschiedener UML-Diagramme im Design-Prozess:

1. Klassendiagramm

- Design der Klassenstruktur
- Beziehungen zwischen Klassen
- Attribute und Methoden
- Pattern-Strukturen

2. Sequenzdiagramm

- Interaktion zwischen Objekten
- Methodenaufrufe
- Zeitliche Abfolge
- Use-Case Realisierung

3. Zustandsdiagramm

- Objektzustände
- Zustandsübergänge
- Ereignisse und Aktionen
- Lifecycle Modellierung

UML-Modellierung

Statische vs. Dynamische Modelle

Statische Modelle (Struktur):
- UML-Klassendiagramm
- Fokus auf Pakete, Klassen, Attribute
- Keine Methodenimplementierung

Dynamische Modelle (Verhalten):
- UML-Interaktionsdiagramme
- Fokus auf Logik und Verhalten
- Implementierung der Methoden

UML-Diagrammtypen

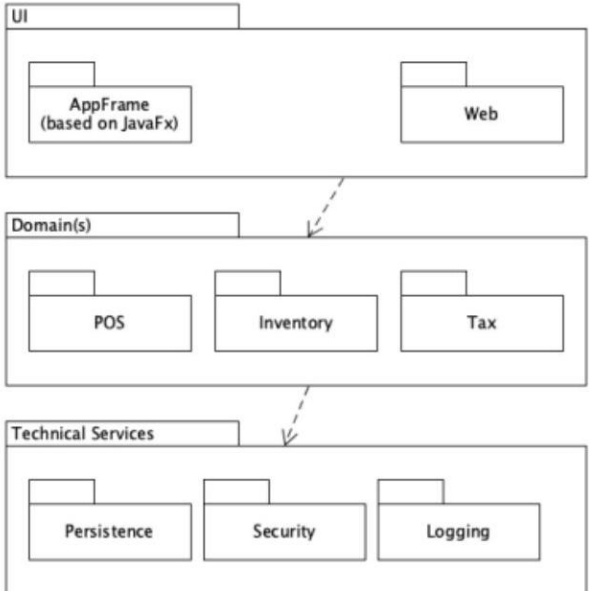
1. Klassendiagramm:
- Klassen und aktive Klassen
- Attribute und Operationen
- Sichtbarkeiten und Beziehungen
- Interfaces und Realisierungen

2. Sequenzdiagramm:
- Lebenslinien und Nachrichten
- Synchrone/Asynchrone Kommunikation
- Aktivierung und Deaktivierung
- Alternative Abläufe

3. Zustandsdiagramm:
- Zustände und Übergänge
- Start- und Endzustände
- Composite States
- Historie und Parallelität

4. Aktivitätsdiagramm:
- Aktionen und Aktivitäten
- Kontroll- und Datenflüsse
- Verzweigungen und Zusammenführungen
- Partitionen (Swimlanes)

UML-Paketdiagramm:
- Definition von Teilsystemen
- Gruppierung von Elementen
- Abhängigkeiten zwischen Paketen



UML Diagrammauswahl

Entscheidungshilfe für die Wahl des UML-Diagrammtyps:

1. Strukturbeschreibung benötigt:
- Klassendiagramm für Typen und Beziehungen
- Paketdiagramm für Modularisierung
- Komponentendiagramm für Bausteinsicht
- Verteilungsdiagramm für Deployment

2. Verhaltensbeschreibung benötigt:
- Sequenzdiagramm für Interaktionsabläufe
- Aktivitätsdiagramm für Workflows
- Zustandsdiagramm für Objektlebenszyklen
- Kommunikationsdiagramm für Objektkollaborationen

3. Abstraktionsebene wählen:
- Analyse: Konzeptuelle Diagramme
- Design: Detaillierte Spezifikation
- Implementation: Codenahes Design

Responsibility Driven Design (RDD)

Design basierend auf Verantwortlichkeiten:
- Klassenentwurf nach Rollen
- Kollaborationsbeziehungen
- Implementierung durch Attribute/Methoden
- Anwendbar auf allen Ebenen

Prüfungsaufgabe: UML-Modellierung

Aufgabe: Modellieren Sie für ein Bibliothekssystem die Ausleihe eines Buches mit:
- Klassendiagramm der beteiligten Klassen
- Sequenzdiagramm des Ausleihvorgangs
- Zustandsdiagramm für ein Buchexemplar

Bewertungskriterien:
- Korrekte UML-Notation
- Vollständigkeit der Modellierung
- Konsistenz zwischen Diagrammen
- Angemessener Detaillierungsgrad

GRASP Prinzipien

General Responsibility Assignment Software Patterns:
- **Information Expert:** Verantwortung basierend auf Information
- **Creator:** Objekterstellung bei starker Beziehung
- **Controller:** Zentrale Steuerungslogik
- **Low Coupling:** Minimale Abhängigkeiten
- **High Cohesion:** Starker innerer Zusammenhang
- **Polymorphism:** Flexibilität durch Schnittstellen
- **Pure Fabrication:** Künstliche Klassen für besseres Design
- **Indirection:** Vermittler für Flexibilität
- **Protected Variations:** Kapselung von Änderungen

GRASP Anwendung

Szenario: Online-Shop Warenkorb-Funktionalität

GRASP-Prinzipien angewandt:
- **Information Expert:**
  - Warenkorb kennt seine Positionen
  - Berechnet selbst Gesamtsumme
- **Creator:**
  - Warenkorb erstellt Warenkorbpositionen
  - Bestellung erstellt aus Warenkorb
- **Controller:**
  - ShoppingController koordiniert UI und Domain
  - Keine Geschäftslogik im Controller
- **Low Coupling:**
  - UI kennt nur Controller
  - Domain unabhängig von UI

UML-Modellierung

UML Diagrammtypen Übersicht

UML bietet verschiedene Diagrammtypen für statische und dynamische Modellierung:

Statische Modelle:
- Klassendiagramm
- Paketdiagramm
- Komponentendiagramm
- Verteilungsdiagramm

Dynamische Modelle:
- Sequenzdiagramm
- Kommunikationsdiagramm
- Zustandsdiagramm
- Aktivitätsdiagramm

Klassendiagramm

Hauptelemente:

- **Klassen:**
  - Name der Klasse
  - Attribute mit Sichtbarkeit
  - Operationen mit Parametern
- **Beziehungen:**
  - Assoziation (normaler Pfeil)
  - Vererbung (geschlossener Pfeil)
  - Implementierung (gestrichelter Pfeil)
  - Aggregation (leere Raute)
  - Komposition (gefüllte Raute)
- **Interfaces:**
  - Stereotyp «interface»
  - Nur Methodensignaturen
  - Implementierungsbeziehung

Klassendiagramm: E-Commerce System

Domänenmodell mit wichtigen Beziehungen:

```
public interface OrderRepository {
    Optional<Order> findById(OrderId id);
    void save(Order order);
}

public class Order {
    private OrderId id;
    private Customer customer;
    private List<OrderLine> orderLines;
    private OrderStatus status;

    public Money calculateTotal() {
        return orderLines.stream()
                         .map(OrderLine::getSubTotal)
                         .reduce(Money.ZERO,
                             Money::add);
    }
}

public class OrderLine {
    private Product product;
    private int quantity;
    private Money price;

    public Money getSubTotal() {
        return price.multiply(quantity);
    }
}
```

Sequenzdiagramm

Notationselemente:

- **Lebenslinien:**
  - Objekte als Rechtecke
  - Vertikale gestrichelte Linie
  - Aktivierungsbalken für Ausführung
- **Nachrichten:**
  - Synchron (durchgezogener Pfeil)
  - Asynchron (offener Pfeil)
  - Antwort (gestrichelter Pfeil)
  - Parameter und Rückgabewerte
- **Kontrollelemente:**
  - alt (Alternative)
  - loop (Schleife)
  - opt (Optional)
  - par (Parallel)

Sequenzdiagramm: Bestellprozess

Interaktion zwischen Komponenten:

```
public class OrderService {
    private final OrderRepository orderRepo;
    private final PaymentService paymentService;

    public OrderConfirmation processOrder(OrderRequest
        request) {
        // Validiere Bestellung
        validateOrder(request);

        // Erstelle Order
        Order order = createOrder(request);
        orderRepo.save(order);

        // Prozessiere Zahlung
        PaymentResult result = paymentService
            .processPayment(order.getId(),
                order.getTotal());

        // Bestaetige Bestellung
        if (result.isSuccessful()) {
            order.confirm();
            orderRepo.save(order);
            return new OrderConfirmation(order);
        }

        throw new PaymentFailedException();
    }
}
```

Zustandsdiagramm

Notationselemente:

- **Zustände:**
  - Startzustand (gefüllter Kreis)
  - Endzustand (Kreis mit Punkt)
  - Einfache Zustände (Rechteck)
  - Zusammengesetzte Zustände
- **Transitionen:**
  - Event [Guard] / Action
  - Interne Transitionen
  - Selbsttransitionen
- **Spezielle Elemente:**
  - History State (H)
  - Deep History (H*)
  - Entry/Exit Points
  - Choice Points

Zustandsdiagramm: Bestellstatus

Implementation eines State Patterns:

```
public interface OrderState {
    void process(Order order);
    void cancel(Order order);
    void ship(Order order);
}

public class NewOrderState implements OrderState {
    @Override
    public void process(Order order) {
        validateOrder(order);
        order.setState(new ProcessingState());
    }

    @Override
    public void cancel(Order order) {
        order.setState(new CancelledState());
    }

    @Override
    public void ship(Order order) {
        throw new IllegalStateException(
            "Cannot ship new order");
    }
}

public class Order {
    private OrderState state;

    public void process() {
        state.process(this);
    }

    void setState(OrderState newState) {
        this.state = newState;
    }
}
```

Aktivitätsdiagramm

Hauptelemente:

- **Aktionen:**
  - Atomare Aktionen
  - Call Behavior Action
  - Send/Receive Signal
- **Kontrollfluss:**
  - Verzweigungen (Diamond)
  - Parallelisierung (Balken)
  - Join/Merge Nodes
- **Strukturierung:**
  - Activity Partitions (Swimlanes)
  - Structured Activity Nodes
  - Interruptible Regions

Aktivitätsdiagramm: Bestellabwicklung

Implementation eines Geschäftsprozesses:

```
public class OrderProcessor {
    public void processOrder(Order order) {
        // Parallele Verarbeitung
        CompletableFuture.allOf(
            validateInventory(order),
            validatePayment(order)
        ).thenRun(() -> {
            if (order.isValid()) {
                fulfillOrder(order);
            } else {
                handleValidationFailure(order);
            }
        });
    }

    private CompletableFuture<Void> validateInventory(
            Order order) {
        return CompletableFuture.runAsync(() -> {
            order.getItems().forEach(item -> {
                if
                    (!inventoryService.isAvailable(item))
                    {
                    throw new
                        OutOfStockException(item);
                }
            });
        });
    }
}
```

Verteilungsdiagramm

Elemente:

- **Nodes:**
  - Device Nodes
  - Execution Environment
  - Artifacts
- **Verbindungen:**
  - Kommunikationspfade
  - Protokolle
  - Multiplizitäten
- **Deployment:**
  - Deployment Specifications
  - Manifestationen

Verteilungsdiagramm: Microservice-Architektur

Deployment-Konfiguration:

```
@Configuration
public class ServiceConfig {
    @Value("${service.host}")
    private String serviceHost;

    @Value("${service.port}")
    private int servicePort;

    @Bean
    public ServiceRegistry registry() {
        return ServiceRegistry.builder()
            .host(serviceHost)
            .port(servicePort)
            .healthCheck("/health")
            .build();
    }

    @Bean
    public LoadBalancer loadBalancer(
            ServiceRegistry registry) {
        return new RoundRobinLoadBalancer(registry);
    }
}
```

Kommunikationsdiagramm

Hauptelemente:

- **Objekte:**
  - Als Rechtecke dargestellt
  - Mit Objektname und Klasse
  - Verbunden durch Links
- **Nachrichten:**
  - Nummerierte Sequenz
  - Synchrone/Asynchrone Aufrufe
  - Parameter und Rückgabewerte
- **Steuerungselemente:**
  - Bedingte Nachrichten [condition]
  - Iterationen *
  - Parallele Ausführung ||

Kommunikationsdiagramm: Shopping Cart

Objektinteraktionen beim Checkout:

```
public class ShoppingCart {
    private List<CartItem> items;
    private CheckoutService checkoutService;

    public Order checkout() {
        // 1: validateItems()
        validateItems();

        // 2: calculateTotal()
        Money total = calculateTotal();

        // 3: createOrder(items, total)
        Order order = checkoutService.createOrder(
            items, total);

        // 4: clearCart()
        items.clear();

        return order;
    }
}
```

Paketdiagramm

Elemente:

- **Pakete:**
  - Gruppierung von Modellelementen
  - Hierarchische Strukturierung
  - Namensräume
- **Abhängigkeiten:**
  - Import/Export von Elementen
  - «use» Beziehungen
  - Zugriffsrechte

UML Diagrammauswahl

Entscheidungshilfen für die Wahl des passenden Diagrammtyps:

1. Statische Struktur

- Klassendiagramm für Typen und Beziehungen
- Paketdiagramm für Modularisierung
- Komponentendiagramm für Bausteinsicht
- Verteilungsdiagramm für physische Verteilung

2. Dynamisches Verhalten

- Sequenzdiagramm für zeitliche Abläufe
- Kommunikationsdiagramm für Objektkollaborationen
- Zustandsdiagramm für Objektlebenszyklen
- Aktivitätsdiagramm für Geschäftsprozesse

3. Verwendungszweck

- Analyse: Konzeptuelle Modellierung
- Design: Detaillierte Spezifikation
- Implementation: Code-nahe Darstellung
- Dokumentation: Architekturübersicht

Beispiel eines kompletten Designs:

```java
// Paketstruktur
package com.example.shop;

// Domain Model
public class Product {
    private ProductId id;
    private String name;
    private Money price;
    private Category category;
}

// Service Layer
@Service
public class ProductService {
    private final ProductRepository repository;
    private final PriceCalculator calculator;

    public Product updatePrice(
            ProductId id, Money newPrice) {
        Product product = repository.findById(id)
            .orElseThrow(ProductNotFoundException::new);

        Money calculatedPrice = calculator
            .calculateFinalPrice(newPrice);

        product.updatePrice(calculatedPrice);
        return repository.save(product);
    }
}

// Controller Layer
@RestController
@RequestMapping("/api/products")
public class ProductController {
    private final ProductService service;

    @PutMapping("/{id}/price")
    public ProductDTO updatePrice(
            @PathVariable ProductId id,
            @RequestBody PriceUpdateRequest request) {
        Product product = service.updatePrice(
            id, request.getNewPrice());
        return ProductDTO.from(product);
    }
}
```