

Softwarearchitektur und Design

Design Patterns und UML-Modellierung

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:

```
1 // Presentation Layer
2 public class CustomerController {
3     private CustomerService service;
4
5     public CustomerDTO getCustomer(String id) {
6         Customer customer = service.findById(id);
7         return CustomerDTO.from(customer);
8     }
9 }
10
11 // Application Layer
12 public class CustomerService {
13     private CustomerRepository repository;
14
15     public Customer findById(String id) {
16         validateId(id);
17         return repository.findById(id)
18             .orElseThrow(CustomerNotFoundException::new);
19     }
20 }
21
22 // Domain Layer
23 public class Customer {
24     private CustomerId id;
25     private String name;
26     private Address address;
27
28     public void updateAddress(Address newAddress) {
29         validateAddress(newAddress);
30         this.address = newAddress;
31     }
32 }
33
34 // Persistence Layer
35 public class CustomerRepository {
36     private JpaRepository<Customer, CustomerId>
37         jpaRepo;
38
39     public Optional<Customer> findById(String id) {
40         return jpaRepo.findById(new CustomerId(id));
41     }
42 }
```

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

```
1 // Performance-optimierte Produktabfrage
2 @Cacheable(value = "products")
3 public ProductDTO getProduct(String id) {
4     ProductDTO product = cache.get(id);
5     if (product == null) {
6         product = repository.findById(id)
7             .map(this::toDTO)
8             .orElseThrow();
9         cache.put(id, product);
10    }
11    return product;
12 }
```

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:

```
1 // Presentation Layer
2 public class CustomerUI {
3     private CustomerService service;
4     public void showCustomerDetails(int id) {
5         Customer customer = service.getCustomer(id);
6         // display logic
7     }
8 }
9
10 // Business Layer
11 public class CustomerService {
12     private CustomerRepository repository;
13     public Customer getCustomer(int id) {
14         return repository.findById(id);
15     }
16 }
17
18 // Data Layer
19 public class CustomerRepository {
20     public Customer findById(int id) {
21         // database access
22         return customer;
23     }
24 }
```

Vorteile:

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

Clean Architecture Implementation

Strukturbeispiel für einen Online-Shop:

```
1 // Enterprise Business Rules (Entities)
2 public class Order {
3     private List<OrderItem> items;
4     private OrderStatus status;
5
6     public Money calculateTotal() {
7         return items.stream()
8             .map(OrderItem::getSubtotal)
9             .reduce(Money.ZERO, Money::add);
10    }
11 }
12
13 // Application Business Rules (Use Cases)
14 public class CreateOrderUseCase {
15     private OrderRepository repository;
16     private PaymentGateway paymentGateway;
17
18     public OrderId execute(CreateOrderCommand command)
19     {
20         Order order = new Order(command.getItems());
21         PaymentResult result = paymentGateway.process(
22             order.calculateTotal());
23         if (result.isSuccess()) {
24             return repository.save(order);
25         }
26         throw new PaymentFailedException();
27     }
28 }
29
30 // Interface Adapters
31 public class OrderController {
32     private CreateOrderUseCase createOrderUseCase;
33
34     public OrderResponse createOrder(OrderRequest
35     request) {
36         CreateOrderCommand command =
37             mapToCommand(request);
38         OrderId id =
39             createOrderUseCase.execute(command);
40         return new OrderResponse(id);
41     }
42 }
```

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:

```
1 @RestController
2 @RequestMapping("/api/users")
3 public class UserProfileController {
4     private final UserService userService;
5
6     @GetMapping("/{id}")
7     public UserProfileDTO getProfile(@PathVariable
8     String id) {
9         UserProfile profile = userService.findById(id);
10        return UserProfileDTO.from(profile);
11    }
12
13    @PutMapping("/{id}")
14    public ResponseEntity<Void> updateProfile(
15        @PathVariable String id,
16        @RequestBody UpdateProfileCommand command)
17    {
18        userService.updateProfile(id, command);
19        return ResponseEntity.ok().build();
20    }
21
22    // Event fuer andere Services
23    public class UserProfileUpdatedEvent {
24        private final String userId;
25        private final String newEmail;
26        private final LocalDateTime timestamp;
27
28        // Konstruktor und Getter
29    }
```

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

```
1 // Model
2 public class CustomerModel {
3     private String name;
4     private List<Order> orders;
5
6     public void addOrder(Order order) {
7         orders.add(order);
8         notifyViews();
9     }
10 }
11
12 // View
13 public class CustomerView {
14     private CustomerModel model;
15
16     public void displayCustomerInfo() {
17         System.out.println("Customer: " +
18             model.getName());
19         System.out.println("Orders: " +
20             model.getOrders().size());
21     }
22 }
23
24 // Controller
25 public class CustomerController {
26     private CustomerModel model;
27     private CustomerView view;
28
29     public void createOrder(OrderData data) {
30         Order order = new Order(data);
31         model.addOrder(order);
32         view.displayCustomerInfo();
33     }
34 }
```

Event-Driven Architecture (EDA)

Basiert auf der Produktion, Erkennung und Reaktion auf Events:

Komponenten:

- Event Producer
- Event Channel
- Event Consumer
- Event Processor

```
1 // Event Definition
2 public class OrderCreatedEvent {
3     private final String orderId;
4     private final LocalDateTime timestamp;
5     private final BigDecimal totalAmount;
6 }
7
8 // Event Producer
9 public class OrderService {
10     private EventBus eventBus;
11
12     public void createOrder(OrderData data) {
13         Order order = orderRepository.save(data);
14         OrderCreatedEvent event = new
15             OrderCreatedEvent(
16                 order.getId(),
17                 LocalDateTime.now(),
18                 order.getTotalAmount()
19             );
20         eventBus.publish(event);
21     }
22 }
23
24 // Event Consumer
25 @EventListener
26 public class InvoiceGenerator {
27     public void handleOrderCreated(OrderCreatedEvent
28         event) {
29         generateInvoice(event.getOrderId());
30     }
31 }
```

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:

```
1 @RestController
2 @RequestMapping("/api/v1/orders")
3 public class OrderController {
4
5     @GetMapping("/{id}")
6     @Operation(summary = "Get order by ID",
7         description = "Returns detailed order
8             information")
9     @ApiResponse(responseCode = "200",
10         description = "Order found"),
11     @ApiResponse(responseCode = "404",
12         description = "Order not found")
13 })
14 public OrderDTO getOrder(@PathVariable String id) {
15     return orderService.findById(id)
16         .map(OrderDTO::from)
17         .orElseThrow(OrderNotFoundException::new);
18 }
19 }
```

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 and Consumer:

```
1 // Message Definition
2 public class OrderMessage {
3     private String orderId;
4     private String customerId;
5     private BigDecimal amount;
6     private OrderStatus status;
7 }
8
9 // Message Producer
10 public class OrderProducer {
11     private MessageQueue messageQueue;
12
13     public void sendOrderCreated(Order order) {
14         OrderMessage message = new OrderMessage(
15             order.getId(),
16             order.getCustomerId(),
17             order.getAmount(),
18             OrderStatus.CREATED
19         );
20         messageQueue.send("orders", message);
21     }
22 }
23
24 // Message Consumer
25 public class OrderProcessor {
26     @MessageListener(queue = "orders")
27     public void processOrder(OrderMessage message) {
28         if (message.getStatus() ==
29             OrderStatus.CREATED) {
30             processNewOrder(message);
31         }
32     }
33
34     private void processNewOrder(OrderMessage message)
35     {
36         // Verarbeitung der Bestellung
37         validateOrder(message);
38         updateInventory(message);
39         notifyCustomer(message);
40     }
41 }
```

API Gateway Pattern

Zentraler Einstiegspunkt für Client-Anfragen:

Verantwortlichkeiten:

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

```
1 @Component
2 public class ApiGateway {
3     private final AuthService authService;
4     private final ServiceRegistry registry;
5
6     @GetMapping("/api/v1/**")
7     public ResponseEntity<Object> routeRequest(
8         HttpServletRequest request,
9         @RequestHeader("Authorization") String
10            token) {
11
12         // Authentifizierung
13         if (!authService.validateToken(token)) {
14             return ResponseEntity.status(401).build();
15         }
16
17         // Service Discovery
18         String serviceName =
19             extractServiceName(request);
20         ServiceInstance instance =
21             registry.getInstance(serviceName);
22
23         // Request Weiterleitung
24         return forwardRequest(instance, request);
25     }
26 }
```

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:

```
1 @RestController
2 @RequestMapping("/api/v1")
3 public class ProductController {
4
5     // Collection Resource
6     @GetMapping("/products")
7     public PagedResponse<ProductDTO> getProducts(
8         @RequestParam(defaultValue = "0") int page,
9         @RequestParam(defaultValue = "20") int
10            size) {
11         return productService.findAll(page, size);
12     }
13
14     // Single Resource
15     @GetMapping("/products/{id}")
16     public ProductDTO getProduct(@PathVariable String
17        id) {
18         return productService.findById(id);
19     }
20
21     // Sub-Resource Collection
22     @GetMapping("/products/{id}/reviews")
23     public List<ReviewDTO> getProductReviews(
24         @PathVariable String id) {
25         return reviewService.findByProductId(id);
26     }
27
28     // Error Handling
29     @ExceptionHandler(ProductNotFoundException.class)
30     public ResponseEntity<ErrorResponse>
31        handleNotFound(
32            ProductNotFoundException ex) {
33         ErrorResponse error = new ErrorResponse(
34             "PRODUCT_NOT_FOUND",
35             ex.getMessage());
36         return ResponseEntity.status(404).body(error);
37     }
38 }
```

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.

```
1 // Creator
2 public abstract class DocumentCreator {
3     public abstract Document createDocument();
4
5     public void openDocument() {
6         Document doc = createDocument();
7         doc.open();
8     }
9 }
10
11 // Concrete Creator
12 public class PDFDocumentCreator extends
13    DocumentCreator {
14     @Override
15     public Document createDocument() {
16         return new PDFDocument();
17     }
18 }
19
20 // Product Interface
21 public interface Document {
22     void open();
23     void save();
24 }
25
26 // Concrete Product
27 public class PDFDocument implements Document {
28     @Override
29     public void open() {
30         // PDF-spezifische Implementation
31     }
32
33     @Override
34     public void save() {
35         // PDF-spezifische Implementation
36     }
37 }
```

Strategy Pattern

Problem: Algorithmus soll zur Laufzeit austauschbar sein.

```
1 // Strategy Interface
2 public interface PaymentStrategy {
3     void pay(Money amount);
4 }
5
6 // Concrete Strategies
7 public class CreditCardStrategy implements
8     PaymentStrategy {
9     private String cardNumber;
10
11     @Override
12     public void pay(Money amount) {
13         // Kreditkarten-Zahlung
14     }
15
16 public class PayPalStrategy implements
17     PaymentStrategy {
18     private String email;
19
20     @Override
21     public void pay(Money amount) {
22         // PayPal-Zahlung
23     }
24 }
25
26 // Context
27 public class ShoppingCart {
28     private PaymentStrategy paymentStrategy;
29
30     public void
31         setPaymentStrategy(PaymentStrategy
32         strategy) {
33         this.paymentStrategy = strategy;
34     }
35
36     public void checkout(Money amount) {
37         paymentStrategy.pay(amount);
38     }
39 }
```

Observer Pattern

Problem: Objekte sollen über Änderungen informiert werden.

```
1 // Observer Interface
2 public interface OrderObserver {
3     void onOrderStateChange(Order order);
4 }
5
6 // Concrete Observer
7 public class EmailNotifier implements
8     OrderObserver {
9     @Override
10     public void onOrderStateChange(Order order)
11     {
12         sendEmail(order.getCustomer(),
13             "Order status: " +
14             order.getStatus());
15     }
16 }
17
18 // Observable
19 public class Order {
20     private List<OrderObserver> observers = new
21         ArrayList<>();
22     private OrderStatus status;
23
24     public void addObserver(OrderObserver
25         observer) {
26         observers.add(observer);
27     }
28
29     public void setStatus(OrderStatus
30         newStatus) {
31         this.status = newStatus;
32         notifyObservers();
33     }
34
35     private void notifyObservers() {
36         observers.forEach(o ->
37             o.onOrderStateChange(this));
38     }
39 }
```

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:

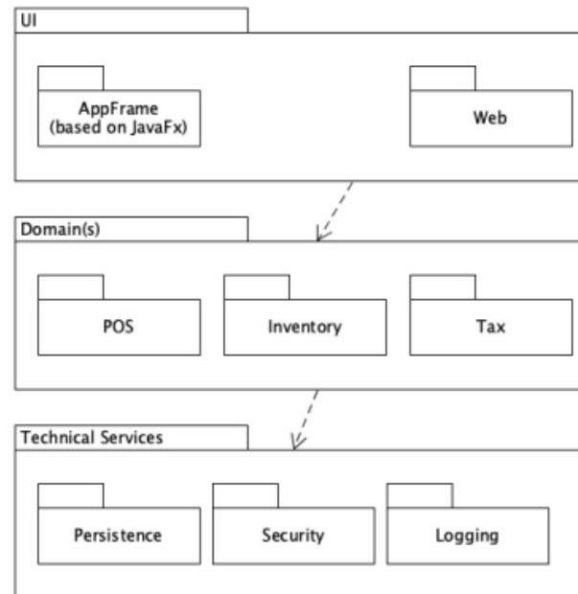
- Zustände und Übergänge
- Start- und Endzustände
- Composite States
- Historie und Parallelität

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:

```
1 public interface OrderRepository {
2     Optional<Order> findById(OrderId id);
3     void save(Order order);
4 }
5
6 public class Order {
7     private OrderId id;
8     private Customer customer;
9     private List<OrderLine> orderLines;
10    private OrderStatus status;
11
12    public Money calculateTotal() {
13        return orderLines.stream()
14            .map(OrderLine::getSubTotal)
15            .reduce(Money.ZERO,
16                Money::add);
17    }
18 }
19
20 public class OrderLine {
21     private Product product;
22     private int quantity;
23     private Money price;
24
25     public Money getSubTotal() {
26         return price.multiply(quantity);
27     }
28 }
```

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:

```
1 public class OrderService {
2     private final OrderRepository orderRepo;
3     private final PaymentService paymentService;
4
5     public OrderConfirmation processOrder(OrderRequest
6         request) {
7         // Validiere Bestellung
8         validateOrder(request);
9
10        // Erstelle Order
11        Order order = createOrder(request);
12        orderRepo.save(order);
13
14        // Prozeduriere Zahlung
15        PaymentResult result = paymentService
16            .processPayment(order.getId(),
17                order.getTotal());
18
19        // Bestaetige Bestellung
20        if (result.isSuccessful()) {
21            order.confirm();
22            orderRepo.save(order);
23            return new OrderConfirmation(order);
24        }
25        throw new PaymentFailedException();
26    }
27 }
```

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:

```
1 public interface OrderState {
2     void process(Order order);
3     void cancel(Order order);
4     void ship(Order order);
5 }
6
7 public class NewOrderState implements OrderState {
8     @Override
9     public void process(Order order) {
10        validateOrder(order);
11        order.setState(new ProcessingState());
12    }
13
14    @Override
15    public void cancel(Order order) {
16        order.setState(new CancelledState());
17    }
18
19    @Override
20    public void ship(Order order) {
21        throw new IllegalStateException(
22            "Cannot ship new order");
23    }
24 }
25
26 public class Order {
27     private OrderState state;
28
29     public void process() {
30        state.process(this);
31    }
32
33    void setState(OrderState newState) {
34        this.state = newState;
35    }
36 }
```


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:

```
1 public class OrderProcessor {
2     public void processOrder(Order order) {
3         // Parallele Verarbeitung
4         CompletableFuture.allOf(
5             validateInventory(order),
6             validatePayment(order)
7         ).thenRun(() -> {
8             if (order.isValid()) {
9                 fulfillOrder(order);
10            } else {
11                handleValidationFailure(order);
12            }
13        });
14    }
15
16    private CompletableFuture<Void> validateInventory(
17        Order order) {
18        return CompletableFuture.runAsync(() -> {
19            order.getItems().forEach(item -> {
20                if
21                    (!inventoryService.isAvailable(item))
22                    {
23                        throw new
24                            OutOfStockException(item);
25                    }
26            });
27        });
28    }
29 }
```

Verteilungsdiagramm

Elemente:

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

Verteilungsdiagramm: Microservice-Architektur

Deployment-Konfiguration:

```
1 @Configuration
2 public class ServiceConfig {
3     @Value("${service.host}")
4     private String serviceHost;
5
6     @Value("${service.port}")
7     private int servicePort;
8
9     @Bean
10    public ServiceRegistry registry() {
11        return ServiceRegistry.builder()
12            .host(serviceHost)
13            .port(servicePort)
14            .healthCheck("/health")
15            .build();
16    }
17
18    @Bean
19    public LoadBalancer loadBalancer(
20        ServiceRegistry registry) {
21        return new RoundRobinLoadBalancer(registry);
22    }
23 }
```

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:

```
1 public class ShoppingCart {
2     private List<CartItem> items;
3     private CheckoutService checkoutService;
4
5     public Order checkout() {
6         // 1: validateItems()
7         validateItems();
8
9         // 2: calculateTotal()
10        Money total = calculateTotal();
11
12        // 3: createOrder(items, total)
13        Order order = checkoutService.createOrder(
14            items, total);
15
16        // 4: clearCart()
17        items.clear();
18
19        return order;
20    }
21 }
```

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

UML in der Praxis

Beispiel eines kompletten Designs:

```
1 // Paketstruktur
2 package com.example.shop;
3
4 // Domain Model
5 public class Product {
6     private ProductId id;
7     private String name;
8     private Money price;
9     private Category category;
10 }
11
12 // Service Layer
13 @Service
14 public class ProductService {
15     private final ProductRepository repository;
16     private final PriceCalculator calculator;
17
18     public Product updatePrice(
19         ProductId id, Money newPrice) {
20         Product product = repository.findById(id)
21             .orElseThrow(ProductNotFoundException::new);
22
23         Money calculatedPrice = calculator
24             .calculateFinalPrice(newPrice);
25
26         product.updatePrice(calculatedPrice);
27         return repository.save(product);
28     }
29 }
30
31 // Controller Layer
32 @RestController
33 @RequestMapping("/api/products")
34 public class ProductController {
35     private final ProductService service;
36
37     @PutMapping("/{id}/price")
38     public ProductDTO updatePrice(
39         @PathVariable ProductId id,
40         @RequestBody PriceUpdateRequest request) {
41         Product product = service.updatePrice(
42             id, request.getNewPrice());
43         return ProductDTO.from(product);
44     }
45 }
```