1. Domain Driven Design

1.1 Analyse der Ubiquitous Language

Die Ubiquitous Language meines Projektes beläuft sich auf folgende beispielhafte Begriffe:

- **Produkt**: Ein Artikel, der im Shop angeboten wird. Hat eine UUID, Namen, Bild, Beschreibung, Preis und Kategorie.
- **Address**: Ein Wertobjekt, das die Adresse des Benutzers enthält (Straße, Stadt, PLZ und Land).
- **Benutzer**: Eine registrierte Person, die sich im Shop anmeldet und Einkäufe tätigt. Dieser wird beschrieben durch eine UUID, einen Benutzernamen, eine Adresse, eine E-Mail und ein Passwort.
- **Warenkorb**: Temporärer Speicher für vom Benutzer ausgewählte Produkte. Enthält eine UUID, einen zugehörigen Benutzer und eine Liste an Produkten.
- **Bestellungsstatus**: Der Status der Bestellung.
- **Bestellung**: Wird aus dem Warenkorb generiert, sobald der Benutzer zur Kasse geht. Enthält eine UUID, den zugehörigen Benutzer, eine Liste an Produkten, einen Bestellungsstatus und eine Adresse, an die geschickt werden soll.
- **Kategorie**: Gruppiert ähnliche Produkte, um die Navigation für den Benutzer zu erleichtern. Diese enthält eine UUID, einen Namen und eine Liste an Kategorien.

1.2 Verwendung taktischer Muster

- **Entities**
- **Value Objects**
- **Aggregates**
- **Repositories**
- **Domain Services**

1.2.1 Entity: User

```
@Entity
@Table(name = "users")
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private UUID id;
```

```
    @Column(nullable = false, unique = true)
    private String username;

    @Embedded
    private Address address;

    @Column(nullable = false, unique = true)
    private Email email;

    @Column(nullable = false)
    private String password;

    @Enumerated(EnumType.STRING)
    @Column(nullable = false)
    private Role role;

    @ElementCollection(fetch = FetchType.EAGER)
    @CollectionTable(
        name = "user_permissions",
        joinColumns = @JoinColumn(name = "user_id")
    )
    @Column(name = "permission")
    @Enumerated(EnumType.STRING)
    private Set<Permission> permissions = new HashSet<>();

    public User() {}

    public User(String username, Email email, String password, Address ac
        this.username = username;
        this.email = email;
        this.password = password;
        this.address = address;
        this.role = role;
    }
```

- User ist eine Entity, weil sie eine eindeutige Identität (UUID id) besitzt.
- User-Daten wie Adresse oder E-Mail können sich über die Zeit ändern.

1.2.2 Value Object: Address

```
@Data
@Embeddable
```

```
public class Address {
    private String street;
    private String city;
    private String postalCode;
    private String country;

    protected Address() {}

    public Address(String street, String city, String postalCode, String
        this.street = street;
        this.city = city;
        this.postalCode = postalCode;
        this.country = country;
    }

    public String getStreet() {return street;}
    public String getCity() {return city;}
    public String getPostalCode() {return postalCode;}
    public String getCountry() {return country;}
}
```

- Address ist ein Value Object, weil es keine eigene Identität hat.
- Ideal für Wiederverwendung und Vergleich.

1.2.3 Aggregate: User als Root

- User ist die Aggregate Root, da es die Verwaltung von Address als eingebettetes Value Object übernimmt.
- Änderungen an der Adresse erfolgen nur über die User-Entität, um Konsistenz zu gewährleisten.

1.2.4 Repository: UserRepository

UserRepository als Schnittstelle

```
public interface UserRepository{
    Optional<User> findByEmail(Email email);
    Optional<User> findByUsername(String username);
    Optional<User> findById(UUID id);
    List<User> findAll();
    User save(User user);
    void deleteById(UUID id);
}
```

JpaUserRepository als Repository-Implementierung

```
@Repository
public interface JpaUserRepository extends JpaRepository<User, UUID>, Use
    Optional<User> findByEmail(String email);
    Optional<User> findByUsername(String username);
}
```

- UserRepository definiert die Schnittstelle für den Datenzugriff auf die User-Entität.
- JpaUserRepository implementiert die Datenbankinteraktion mit Spring Data JPA und kapselt SQL-Logik.
- Dadurch bleibt die Domäne von technischen Details getrennt.

1.2.5 Domain Service: UserService

```
@Service
public class UserService {

    private UserRepository userRepository;

    public UserService(UserRepository userRepository){
        this.userRepository = userRepository;
    }

    public UserDTO registerUser(RegisterUserDTO registerUserDTO) {

        if (userRepository.findByEmail(registerUserDTO.getEmail()).isPres
            throw new UserRegistrationException("Diese E-Mail wird bereit
        }
        if (userRepository.findByUsername(registerUserDTO.getUsername()).
            throw new UserRegistrationException("Dieser Benutzername wirc
        }

        UserFactory userFactory;

        if (registerUserDTO.getRole().equals(Role.ADMIN)) {
            userFactory = new AdminUserFactory();

        } else {
            userFactory = new DefaultUserFactory();
        }
```

```
        User userCreated = userFactory.createUser(
            registerUserDTO.getUsername(),
            registerUserDTO.getEmail(),
            registerUserDTO.getPassword(),
            registerUserDTO.getAddress()
        );

        User savedUser = userRepository.save(userCreated);
        return UserMapper.toDTO(savedUser);
    }

    public UserDTO getUserById(UUID id) {
        User user = userRepository.findById(id)
                .orElseThrow(() -> new UserNotFoundException("Benutzer ni
        return UserMapper.toDTO(user);
    }

    public UserDTO updateUser(UUID id, User updatedUser) {
        User existingUser = userRepository.findById(id)
                .orElseThrow(() -> new UserNotFoundException("Benutzer ni

        existingUser.setUsername(updatedUser.getUsername());
        existingUser.setEmail(updatedUser.getEmail());

        return UserMapper.toDTO(userRepository.save(existingUser));
    }

    public void deleteUser(UUID id) {
        userRepository.deleteById(id);
    }
}
```

- UserService ist ein Domain Service, weil er geschäftslogische Regeln zur Benutzerregistrierung kapselt.
- Nutzt UserRepository, um die Datenbankabfragen zu abstrahieren.

1.2.6 Data Transfer Object (DTO): UserDTO

```
public class UserDTO {

    private UUID id;
    private String username;
```

```
    private Email email;
    private Address address;
    private Role role;


    public UserDTO(UUID id, String username, Email email, Address address
        this.id = id;
        this.username = username;
        this.email = email;
        this.address = address;
        this.role = role;
    }
}
```

Für die Kommunikation zwischen den Schichten wird ein DTO benutzt. Dieses enthält ausschließlich Felder zur Repräsentation der Daten, aber keine Geschäftslogik. Dieses Prinzip wurde benutzt, weil man dadurch

- interne Entitäten von der Außenwelt entkoppeln kann.
- verhindern kann, dass versehentlich Zugriff auf sensible Daten wie das Passwort, durchgegührt wird.

1.2.6 Mapper: UserMapper

```
public class UserMapper {

    private UserMapper() {}

    public static UserDTO toDTO(User user) {
        return new UserDTO(user.getId(), user.getUsername(), user.getEmai
    }

    public static User toEntity(RegisterUserDTO registerUserDTO) {
        return new User(registerUserDTO.getUsername(), registerUserDTO.ge
    }
}
```

Ein Mapper konvertiert zwischen DTOs und Entitäten. Hier wird ein statischer Mapper verwendet, um die manuelle Zuordnung durchzuführen.

2. Clean Architecture

2.1 Schichtarchitektur und Begründung

Das Projekt ist in folgende Schichten unterteilt:

Plugins-Schicht (0-plugins):

- Enthält die Main Methode der Anwendung.
- Sie dient als Einstiegspunkt der Anwendung und initialisiert alle notwendigen Framework-Komponenten (Spring Boot).

Adapter-Schicht (1-adapters):

- Verantwortlich für die Kapselung von Infrastruktur- und Schnittstellenlogik.
- Beinhaltet wichtige Komponenten wie:
  - REST-Controller.
  - JPA Repositories, die als Brücke zur Datenbank fungieren.
  - Security-Konfigurationen wie die JWT-basierte Authentifizierung oder CORS-Konfiguration für das Frontend.

Application-Schicht (2-application):

- Definiert Use Cases, also spezifische Geschäftsprozesse der Anwendung.
- Implementiert als eigenständige Klassen um Erweiterbarkeit zu gewährleisten und vereinfachen

Beispiel: **PlaceOrderUseCase**

```
@Component
public class PlaceOrderUseCase {

    private final OrderService orderService;

    public PlaceOrderUseCase(OrderService orderService) {
        this.orderService = orderService;
    }

    public OrderDTO execute(UUID userId) {
        return orderService.placeOrder(userId);
    }
}
```

Domain-Schicht (3-domain):

- Enthält zentrale Geschäftslogik, Entitäten, Value Objects, Services und Repositories.
- Die Domain-Schicht hat keine Abhängigkeit zu externen Frameworks, was sie besonders stabil und langlebig macht.

Abstraktionscode-Schicht (4-abstractioncode)

- Diese Schicht ist für wiederverwendbare, domänenunabhängige Logik gedacht.
- Diese kapselt wiederverwendbare, domänenunabhängige Logik.
- In der aktuellen Implementierung ist sie leer.
  Kapselt wiederverwendbare, domänenunabhängige Logik. Ist in meinem Fall leer.

2.2 Implementierte Schichten

- Alle fünf Schichten wurden implementiert.
- Plugins (0-plugins) enthält nur die Main-Methode und übernimmt keine Geschäftslogik.
- Abstraktionscode (4-abstractioncode) ist aktuell leer, kann aber später für übergreifende Funktionen verwendet werden.

3. Programming Principles

1. Single Responsibility Principle (SRP – SOLID)

Prinzip:
Eine Klasse sollte nur eine einzige Verantwortlichkeit haben. Änderungen an der Klasse sollten nur aus einem Grund nötig sein.

Anwendung im Projekt:
Der Cart CartQuantityService ist ausschließlich dafür da, um die Anzahl eines Produktes im Warenkorb zu ändern.

Codeausschnitt:

```
@Service
public class CartQuantityService {

    public void updateQuantity(Cart cart, Product product, int quantity)
        List<Product> updatedProducts = new ArrayList<>();
```

```
            for (int i = 0; i < quantity; i++) {
                updatedProducts.add(product);
            }

            cart.getProducts().removeIf(p -> p.getId().equals(product.getId()
            cart.getProducts().addAll(updatedProducts);
        }

    }
```

Begründung:
Jede Klasse hat jetzt eine klar abgegrenzte Verantwortung. Das erhöht Testbarkeit, Verständlichkeit und Wartbarkeit.

2. Don’t Repeat Yourself (DRY)

Prinzip:
Wiederhole dich nicht – zentrale Logik, die mehrfach vorkommt, soll an einer Stelle definiert sein.

Anwendung im Projekt:
Der GlobalExceptionHandler verwaltet Global die Exceptions. Darüber hinaus wurde der String-Literal "error" mehrfach verwendet. Das wurde durch eine Konstante ersetzt.

Beispiel:

```
@RestControllerAdvice
public class GlobalExceptionHandler {

    private static final String ERROR_KEY = "error";

        @ExceptionHandler(UserRegistrationException.class)
        public ResponseEntity<Map<String, String>> handleUserRegistratior
            Map<String, String> errorResponse = new HashMap<>();
            errorResponse.put(ERROR_KEY, ex.getMessage());
            return new ResponseEntity<>(errorResponse, HttpStatus.BAD_REQ
        }
}
```

Begründung:
Einfache Wartbarkeit und Erweiterbarkeit.

3. Low Coupling / High Cohesion (GRASP)

Prinzip:

- Low Coupling: Klassen sollen möglichst unabhängig voneinander sein.
- High Cohesion: Klassen sollten thematisch fokussiert und logisch zusammenhängend sein.

Anwendung im Projekt:
Die Aufteilung in einzelne, fachlich fokussierte UseCases wie AddProductToCartUseCase oder UpdateOrderStatusUseCase sorgt für saubere Trennung der Aufgaben.

Beispiel:

```
@Component
public class UpdateOrderStatusUseCase {

    private final OrderService orderService;

    public UpdateOrderStatusUseCase(OrderService orderService) {
        this.orderService = orderService;
    }

    public OrderDTO execute(UUID orderId, OrderStatus status) {
        return orderService.updateOrderStatus(orderId, status);
    }
}
```

Begründung:
Falls sich eine neue Aufgabe ergibt, die implementiert werden muss kann einfach eine neue UseCase erstellt werden, die auf den jeweiligen Service zugreift. Dadurch kann man das Projekt leicht erweitern oder anpassen.

4. Controller (GRASP)

Prinzip:
Ein Controller ist zuständig für das Entgegennehmen von Eingaben und Delegieren an die Fachlogik – nicht für Geschäftslogik.

Anwendung im Projekt:
Die ProductController-Klasse leitet alle Aktionen an spezialisierte UseCases weiter. Keine Geschäftslogik befindet sich im Controller.

Beispiel:

```
@RestController
@RequestMapping("/products")
public class ProductController {

    private final GetAllProductsUseCase getAllProductsUseCase;
    private final GetProductByIdUseCase getProductByIdUseCase;
    private final UpdateProductUseCase updateProductUseCase;

    public ProductController(
        GetAllProductsUseCase getAllProductsUseCase,
        GetProductByIdUseCase getProductByIdUseCase,
        UpdateProductUseCase updateProductUseCase){

        this.getAllProductsUseCase = getAllProductsUseCase;
        this.getProductByIdUseCase = getProductByIdUseCase;
        this.updateProductUseCase = updateProductUseCase;
    }

    @GetMapping
    public List<ProductDTO> getAllProducts() {
        return getAllProductsUseCase.execute();
    }

    @GetMapping("/{id}")
    public ProductDTO getProductById(@PathVariable UUID id) {
        return getProductByIdUseCase.execute(id);
    }

    @PutMapping("/update/{productId}")
    public ProductDTO updateProduct(@PathVariable UUID productId, @Reques
        return updateProductUseCase.execute(productId, updatedProduct);
    }
}
```

Begründung:
Der Controller dient als „Koordinator“, nicht als Logikträger. Die Trennung ist sauber und entspricht dem GRASP-Prinzip „Controller“.

5. Open-Closed Principle (OCP – SOLID)

Prinzip:
Klassen sollten für Erweiterung offen, aber für Modifikation geschlossen sein. Neue Funktionalität sollte ergänzt, nicht überschrieben werden.

Anwendung im Projekt:
Durch das Factory Pattern (UserFactory, AdminUserFactory, DefaultUserFactory) können neue Benutzerrollen hinzugefügt werden, ohne bestehende Codebasis zu ändern.

Beispiel:

```
public abstract class UserFactory {

    private final Role role;

    protected UserFactory(Role role) {
        this.role = role;
    }

    public User createUser(String username, Email email, String password,
        User user = new User(username, email, password, address, this.rol
        afterCreation(user);
        return user;
    };

    protected void afterCreation(User user) {}
}

public class AdminUserFactory extends UserFactory{

    public AdminUserFactory() {
        super(Role.ADMIN);
    }

    @Override
    protected void afterCreation(User user) {
        user.addPermission(Permission.CREATE_PRODUCTS);
        user.addPermission(Permission.DELETE_PRODUCTS);
        user.addPermission(Permission.MANAGE_DISCOUNTS);
    }
}
```

Begründung:
Neue Rollen (z. B. Support oder Developer) können hinzugefügt werden, ohne den Basiscode zu ändern – Erweiterung statt Änderung.

4. Unit Tests

Im Projekt wurden ingesamt 17 Unit Tests geschrieben, um die Geschäftslogik zu testen. Dabei kamen JUnit 5.9 und Mockito zum Einsatz, um Abhängigkeiten wie Datenbanken oder externe Services zu mocken.

Beispiel:

```
@Test
void testLoginWithValidCredentials_ShouldReturnLoginResult() {

    Email email = new Email("test@domain.de");
    String password = "testPassword";

    User mockUser = new User();
    mockUser.setId(UUID.randomUUID());
    mockUser.setEmail(email);
    mockUser.setPassword(password);

    when(userRepository.findByEmail(email)).thenReturn(Optional.of(mockUs
    when(jwtProvider.generateToken(anyString())).thenReturn("mocked-jwt-t

    LoginResult loginResult = authService.login(email, password);

    assertNotNull(loginResult, "LoginResult should not be null.");
    assertEquals(mockUser, loginResult.getUser(), "Returned user should m
    assertEquals("mocked-jwt-token", loginResult.getToken(), "Token shoul
}
```

Was wird getestet?

- Ob ein erfolgreicher Login bei gültigen Zugangsdaten möglich ist.
- Ob der zurückgegebene LoginResult die korrekten Daten enthält.
- Ob der generierte JWT-Token korrekt gemockt wird.

Beispiel:

```
@Test
void testGetProductById_Found_ShouldReturnProductDTO() {

    UUID productId = UUID.randomUUID();
```

```
        Product mockProduct = new Product();
        mockProduct.setId(productId);
        mockProduct.setName("Test Product");

        Category mockCategory = new Category("TestCategory");

        mockProduct.setCategory(mockCategory);

        when(productRepository.findById(productId)).thenReturn(Optional.of(mc

        ProductDTO result = productService.getProductById(productId);

        assertNotNull(result, "ProductDTO sollte nicht null sein.");
        assertEquals(productId, result.getId(), "Die ID des DTO sollte mit de
        assertEquals("Test Product", result.getName(), "Der Name des DTO soll
    }
```

Was wird getestet?

- Ob ein Produkt mit gültiger ID korrekt aus dem Repository geladen wird.
- Ob die Rückgabe als ProductDTO korrekt gemappt ist.
- Ob Name und ID mit dem erwarteten Wert übereinstimmen

Weitere Unit Tests befinden sich im Verzeichnis
`backend/ecommerce-shop/3-domain/src/test/java/com/ecommerce/domain/service`

5. Refactoring

Durchgeführte Refactorings

1. Refactoring: Replace Data Value with Object

In der ursprünglichen Version wurde die E-Mail-Adresse eines Nutzers lediglich als String gespeichert. Diese Implementierung birgt Schwächen, da keinerlei Validierung, Formatierung oder semantische Trennung möglich ist.
Zur Verbesserung wurde ein Value Object Email eingeführt, das die E-Mail-Adresse kapselt, validiert und einheitlich behandelt.
Dies erhöht die Typsicherheit und sorgt für eine bessere Trennung von Domänenlogik.
Commit-Link

2. Refactoring: Extract Class

Die Methode `updateProductQuantity(...)` im CartService enthielt umfangreiche Logik zur Berechnung und Manipulation der Produktmenge im Warenkorb.
Um die Single Responsibility des Services zu wahren und die Lesbarkeit zu verbessern, wurde diese Logik in eine eigene Klasse CartQuantityService ausgelagert.
Damit ist die Verantwortlichkeit klar getrennt, testbarer und besser wartbar.
Commit-Link

Identifizierte Code Smells

1. Duplicate Code in Use Cases

In verschiedenen Use Cases (z. B. UpdateProductQuantity, RemoveProductFromCart) wurde dieselbe Cart-Logik wie in den zugehörigen Services verwendet. Diese Duplikate wurden in den Use Cases entfernt. Commit-Link

---

2. Long Parameter List im ProductController

Der Konstruktor des ProductController umfasst acht Use Cases als Parameter. Dies erschwert die Lesbarkeit, Testbarkeit und Erweiterbarkeit des Codes.
Ein Refactoring wie Introduce Parameter Object oder eine modulare Aufteilung des Controllers könnte hier helfen, die Verantwortung besser zu trennen und den Code zu vereinfachen.

---

3. Anemic Domain Model

Mehrere Domain-Entitäten (z. B. Cart) enthalten kaum eigene Logik, sondern dienen primär als Datencontainer.
Ein typisches Beispiel ist das direkte Hinzufügen eines Produkts per `cart.getProducts().add(product)`, anstatt die Domänenlogik über `cart.addProduct(product)` zu kapseln.
Dies widerspricht dem Prinzip des objektorientierten Designs, bei dem das Verhalten eng an die Daten gebunden sein sollte.
Commit-Link

---

4. Primitive Obsession

In mehreren Klassen (z. B. RegisterUserDTO oder User) werden Domänenobjekte wie password oder username als einfache String-Typen verwendet.
Dadurch gehen Validierung, Lesbarkeit und Typsicherheit verloren.
Ein Refactoring zu Value Objects wie Password oder Username wäre hier sinnvoll, um Wiederverwendbarkeit zu fördern.

6. Entwurfsmuster

Begründung für den Einsatz des Factory Method Patterns

1. Übersicht

In unserem Projekt müssen unterschiedliche Arten von User-Objekten erzeugt werden, z. B. Ein normaler Standart-User und Admin-User, die besondere Berechtigungen wie das Löschen von Produkten haben. Hierfür wurde das Entwurfssmuster Factory Patern implementiert.

- Eine abstrakte Basisklasse (UserFactory) kapselt die grundlegende Erzeugungslogik.
- Konkrete Unterklassen (AdminUserFactory, DefaultUserFactory) entscheiden, welche Rolle (Role) gesetzt wird. Basierend darauf können Berechtigungen erteilt werden.

So wird vermieden, dass der Service oder Controller explizit wissen müssen, wie genau ein Admin-User im Vergleich zu einem Standard-User erzeugt wird.

2. Warum wird das Muster an der Stelle eingesetzt?

1. **Kapselung der Erzeugungslogik**
   Anstatt bei jeder Instanziierung von User zu entscheiden, ob es sich um einen Admin oder Standard-User handelt, wird diese Entscheidung über konkrete Factory-Klassen getroffen.

2. **Erweiterbarkeit**
   Möchten wir neue Rollen einführen (z. B. Moderator, Support oder Developer), lässt sich problemlos eine weitere Factory-Klasse ergänzen, ohne vorhandenen Code zu verändern.

3. Wie verbessert das Muster den Code?

- **Reduzierte Redundanz**
  Anstatt in mehreren Klassen immer wieder den gleichen Konstruktor mit anderen Parametern

(z. B. `Role.ADMIN` vs. `Role.USER`) aufzurufen, wird der Code in der Basisklasse zentralisiert. Die konkreten Factory-Klassen geben nur noch ihre Spezifika an.

- **Klar strukturierte Verantwortlichkeiten**
  Der Service ist von den Details der Objekt-Initialisierung entkoppelt und ruft lediglich eine gemeinsame Methode (`createUser(...)`) auf.

- **Gute Erweiterbarkeit**
  Neue Rollen oder spezielle Logiken lassen sich hinzufügen, indem man eine weitere Unterklasse der abstrakten Factory erstellt (z. B. ModeratorUserFactory).

4. Vorteile und Nachteile durch den Einsatz des Musters

Vorteile

- **Trennung von Erstellung und Verwendung**
  Die Logik, wie ein User-Objekt gebaut wird, ist von der Logik entkoppelt, die dieses Objekt später nutzt oder speichert.
- **Erweiterbarkeit**
  Man kann leicht zusätzliche Rollen über neue Factories hinzufügen.
- **Einheitliche Validierung**
  Man kann in der abstrakten Factory Valiedierung einbauen und somit Code Duplikate verhindern.