# 1. Domain Driven Design

## 1.1 Analyse der Ubiquitous Language

Im Projekt *Monetenmanager* wurde eine Ubiquitous Language etabliert, die alle fachlichen Begriffe und ihre Bedeutung innerhalb der Finanzverwaltungsdomäne eindeutig beschreibt. Diese Sprache dient als Brücke zwischen Fachlichkeit und technischer Umsetzung.

| Begriff | Beschreibung |
|---|---|
| **User** | Eine registrierte Person, die mit dem System interagiert. Verfügt über eine UUID, eine E-Mail-Adresse (als Value Object), ein Passwort und verwaltet eigene Budgets, Kategorien und Transaktionen. |
| **Transaction** | Eine finanzielle Bewegung, z. B. Einnahme oder Ausgabe. Enthält eine UUID, einen Betrag (Value Object Amount), ein Datum, eine Beschreibung, einen Typ (EINNAHME oder AUSGABE) und eine zugehörige Kategorie. |
| **Category** | Eine vom Nutzer selbst angelegte Gruppe zur Einordnung von Transaktionen. Besitzt eine UUID, einen Namen (Value Object CategoryName) und eine Farbe (Value Object CategoryColor). |
| **Budget** | Ein vom Benutzer definierter Betrag, der pro Kategorie für einen bestimmten Monat vorgesehen ist. |
| **MonthlyOverview** | Eine automatisch generierte Übersicht der Ausgaben/Einnahmen eines Monats, gruppiert nach Kategorien. |
| **Amount** | Ein Value Object, dass den Betrag einer Transaktion darstellt. Enthält Validierungslogik zur Sicherstellung positiver numerischer Werte. |

## 1.2 Verwendung taktischer DDD-Muster

### 1.2.1 Entity: Transaction

```
public class Transaction {
    private UUID id;
    private UUID userId;
    private String category;
    private Amount amount;
    private LocalDate date;
    private String description;
    private TransactionType type;
    ...
}
```

Eine Transaktion ist eine Entity, da sie eine eindeutige Identität (UUID) besitzt und sich ihre Attribute im Laufe der Zeit ändern können

### 1.2.2 Value Object: Amount

```
public class Amount {
    private final BigDecimal value;

    public Amount(BigDecimal value) {
        if (value == null || value.compareTo(BigDecimal.ZERO) < 0) {
            throw new IllegalArgumentException("Amount must not be null or negative.");
        }
        this.value = value;
    }
...
```

Amount ist ein typisches Value Object: Es ist unveränderlich, besitzt keine eigene Identität und kann bei gleicher Wertebasis ersetzt werden.

Weitere Value Objects:

- Email für Benutzerauthentifizierung
- CategoryName und CategoryColor zur klaren Definition von Kategorien

#### 1.2.3 Aggregate Root: User

```
public class User implements AggregateRoot {
    private final UUID id;
    private final String name;
    private final Email email;
    private final String password;
...}
```

Der User ist die zentrale Aggregate Root, unter der alle weiteren Benutzer-bezogenen Entitäten (wie Budgets, Kategorien, Transaktionen) logisch gruppiert werden. Änderungen an abhängigen Objekten erfolgen über die Aggregate Root zur Wahrung der Konsistenz.

#### 1.2.4 Repository: UserRepository

```
public interface UserRepository {
    void save(User user);
    Optional<User> findByEmail(Email email);
    Optional<User> findById(UUID id);
}
```

Das Repository abstrahiert den Datenzugriff auf die User-Entität und trennt damit die Domänenschicht von technischen Details wie der Persistenz.

#### 1.2.5 Domain Service: MonthlyOverviewService

```
public class MonthlyOverviewService {
    private final TransactionRepository transactionRepository;
    private final CategoryRepository categoryRepository;
    private final BudgetRepository budgetRepository;
    public MonthlyOverviewService(TransactionRepository
transactionRepository,
                                  CategoryRepository categoryRepository,
                                  ...
    }
 public List<OverviewEntry> getMonthlyOverview(UUID userId, YearMonth
month) {...}
```

Der MonthlyOverviewService enthält domänenspezifische Logik zur Erstellung monatlicher Finanzübersichten. Da diese Logik keine spezifische Entity erfordert, ist ein Domain Service hier die passende Wahl.

#### 1.2.6 DTOs und Mapper

Da die Anwendung vollständig über die Konsole läuft und keine externe Schnittstelle (z. B. REST-API) anbietet, wurde auf separate DTOs und Mapper verzichtet. Die Datenflüsse sind durch die klare Schichtenarchitektur dennoch sauber strukturiert.

## 2. Clean Architecture

### 2.1 Schichtarchitektur und Begründung

Das Projekt *Monetenmanager* folgt den Prinzipien der Clean Architecture. Ziel dieser Architektur ist es, die Geschäftslogik (Domain) vollständig von externen Einflüssen wie Frameworks, Datenbanken oder der Benutzerschnittstelle zu trennen. Dadurch bleibt der fachliche Kern der Anwendung stabil, erweiterbar und unabhängig testbar.

**Schichten im Überblick**

| Schicht | Beschreibung |
|---|---|
| **CLI-Schicht (Command Line Interface)** | Beinhaltet alle CLI-Handler wie UserCLIHandler, TransactionCLIHandler etc. Diese Kommandos sind für die Interaktion mit dem Benutzer zuständig – sie fragen Eingaben ab und geben Ausgaben strukturiert zurück. |
| **Service-Schicht** | Implementiert alle Anwendungsfälle bzw. Use Cases wie das Anlegen einer Transaktion, das Setzen eines Budgets oder das Erstellen von Monatsübersichten. Hier befinden sich Klassen wie UserService, TransactionService oder MonthlyOverviewService. |
| **Domain-Schicht** | Beinhaltet die zentrale Geschäftslogik: Entities, Value Objects, Aggregate Roots sowie Domain Services. Diese Schicht kennt weder Frameworks noch Datenbanktechnologien. |
| **Repository-Schicht** | Realisiert die technische Umsetzung von Schnittstellen, z. B. UserRepositoryImpl, TransactionRepositoryImpl. Sie kapselt den Zugriff auf die PostgreSQL-Datenbank mittels Spring Data JPA. |
| **Plugins-Schicht** | Enthält den Einstiegspunkt der Anwendung, typischerweise die Main-Methode, sowie die Initialisierung der Spring Boot-Anwendung. |

### 2.2 Umsetzung im Projekt

- Die CLI-Schicht bildet die konkrete Benutzeroberfläche, logisch vergleichbar mit einem REST-Controller in einer Web-App, aber auf Konsoleninteraktion ausgelegt.
- Die Service-Schicht vermittelt zwischen CLI und Domainlogik. Sie enthält fachlich motivierte Services, die Geschäftsprozesse orchestrieren.
- In der Domain-Schicht werden alle Kernmodelle abgebildet: Nutzer, Kategorien, Budgets, Transaktionen sowie deren Regeln und Validierungen.
- Die Infrastructure-Schicht ist zuständig für konkrete Implementierungen von Repositories und ggf. technischen Hilfsklassen (z. B. Persistenzlogik).
- Die Plugins-Schicht ist schmal gehalten und dient nur der Initialisierung – ganz im Sinne von Clean Architecture.

Die Abhängigkeiten verlaufen nur von außen nach innen: CLI → Application → Domain. Die Infrastruktur wird injiziert, aber nie direkt von der Domain angesprochen.

## 3. Programming Principles

Im Projekt *Monetenmanager* wurden verschiedene bewährte Programmierprinzipien angewendet, um Wartbarkeit, Erweiterbarkeit und Lesbarkeit sicherzustellen. Die folgenden Prinzipien aus SOLID und GRASP wurden dabei bewusst berücksichtigt.

### 3.1 Single Responsibility Principle (SRP – SOLID)

**Prinzip:** Eine Klasse sollte nur eine einzige Verantwortlichkeit haben. Änderungen sollten nur aus einem einzigen Grund erfolgen.

**Anwendung im Projekt:** Jede Serviceklasse in der Application-Schicht verfolgt eine klar definierte Aufgabe. Zum Beispiel:

```
@Component
public class TransactionService {
    public void createTransaction(UUID userId, String category, Amount
amount, TransactionType type) {…}
    public List<Transaction> getTransactionsForUser(UUID userId) {...}
```

Die Serviceklasse TransactionService ist ausschließlich für die Verwaltung von Transaktionen zuständig. Sie kümmert sich weder um User-Logik noch um Persistenzdetails, das machen Repository-Interfaces und die Infrastruktur.

Auch CLI-Handler wie TransactionCLIHandler kapseln ausschließlich Ein-/Ausgabeoperationen und delegieren Logik an den Service.

### 3.2 Don’t Repeat Yourself (DRY)

**Prinzip:** Wiederhole dich nicht, zentrale Logik soll nur einmal existieren.

**Anwendung im Projekt:**

- Methoden wie das Parsen von Eingaben oder Validierungen wurden in Hilfsklassen ausgelagert oder zentral im Service abgehandelt.
- Value Objects wie Email oder Amount kapseln Validierungslogik, um dieselben Abfragen nicht mehrfach in verschiedenen Services durchführen zu müssen.
- Ein Beispiel ist das Email-Value Object:

```
public class Email {


    private static final Pattern EMAIL_PATTERN =
            Pattern.compile("^[A-Za-z0-9+_.-]+@[A-Za-z0-9.-]+$");
    private final String value;
    public Email(String value) {
        if (value == null || !EMAIL_PATTERN.matcher(value).matches()) {
            throw new IllegalArgumentException("Ungültige E-Mail-Adresse:
" + value);
        }
        this.value = value;
    }
```

Diese Logik muss dadurch **nicht** in jedem Service oder CLI-Handler dupliziert werden.

### 3.3 Low Coupling / High Cohesion (GRASP)

**Prinzip:** Klassen sollten möglichst unabhängig voneinander sein (Low Coupling) und in sich eine starke thematische Kohärenz aufweisen (High Cohesion).

**Anwendung im Projekt:**

- Jeder CLI-Handler ist unabhängig und zuständig für genau eine Domäne (User, Category, Transaction etc.).
- Services arbeiten über Interfaces, sodass Implementierungen austauschbar sind.
- Die Architektur trennt UI, Use Cases und Domänenmodell strikt voneinander.

### 3.4 Controller (GRASP)

**Prinzip:** Controller (hier: CLI-Handler) sind zuständig für Eingaben, Validierungen und Weiterleitung an die Geschäftslogik, aber nicht für fachliche Logik selbst.

**Anwendung im Projekt:** Die TransactionCLIHandler-Klasse kümmert sich z. B. nur um Benutzereingaben und die Steuerung des CLI-Flows:

```
public class TransactionCLIHandler {
    public void handleAddTransaction(UUID userId) {
        ...
        transactionService.createTransaction(...);
    }
}
```

Die eigentliche Geschäftslogik zur Validierung, Berechnung und Persistenz befindet sich im TransactionService.

### 3.5 Open-Closed Principle (OCP – SOLID)

**Prinzip:** Klassen sollen für Erweiterungen offen, aber für Modifikationen geschlossen sein.

**Anwendung im Projekt:**

- Durch den Einsatz von Value Objects (z. B. Email, CategoryName) kann Validierungslogik erweitert werden, ohne an mehreren Stellen im Code nachträglich Änderungen vornehmen zu müssen.
- Die Services greifen auf Repositories über Interfaces zu. Neue Persistenzstrategien (z. B. Caching, alternative DBs) können eingebaut werden, ohne bestehende Klassen zu verändern.

## 4. Unit Tests

Unit Tests sind ein wichtiger Bestandteil der Qualitätssicherung im *Monetenmanager*-Projekt. Sie dienen dazu, die fachliche Logik der Anwendung isoliert zu überprüfen, Fehler frühzeitig zu erkennen und Änderungen sicher durchzuführen.

### 4.1 Teststrategie

- **Ziel:** Sicherstellung der Korrektheit von Use Cases und Geschäftslogik
- **Frameworks:** JUnit 5 & Mockito
- **Testebene:** Fokus auf *Services* (z. B. TransactionService, MonthlyOverviewService, UserService)
- **Mocking:** Repositories und externe Abhängigkeiten werden gemockt, um Tests unabhängig von Datenbank oder CLI zu halten

### 4.2 Beispiel: Test eines Service-Use-Case

Was wird getestet?

**TransactionServiceTest**

- Ob eine gültige Transaktion korrekt hinzugefügt wird
- Ob die Validierung (z. B. positiver Betrag) greift
- Ob die Transaktion korrekt an das Repository übergeben wird

```
@Test
    public void testCreateTransaction_savesTransaction() {
        UUID userId = UUID.randomUUID();
        String category = "Essen";
        Amount amount = new Amount(new BigDecimal("100.00"));
        TransactionType type = TransactionType.AUSGABE;

        service.createTransaction(userId, category, amount, type);

        verify(repository).save(argThat(t ->
                t.getUserId().equals(userId) &&
                t.getCategory().equals(category) &&
                t.getAmount().getValue().compareTo(new
BigDecimal("100.00")) == 0 &&
                t.getType() == type
        ));
    }
```

### 4.3 Beispiel: MonthlyOverviewService

Was wird getestet?

**MonthlyOverviewServiceTest**

- Ob die Übersicht korrekt erstellt wird
- Ob die Transaktionen eines Monats korrekt gruppiert werden
- Ob Summen je Kategorie korrekt berechnet werden

```
@Test
    public void testGenerateMonthlyOverview_returnsCorrectStructure() {
        UUID userId = UUID.randomUUID();
        YearMonth month = YearMonth.of(2025, 5);
        Transaction t1 = new Transaction(UUID.randomUUID(), userId,
"Essen", new Amount(new BigDecimal("50.00")),
            TransactionType.AUSGABE, LocalDateTime.of(2025, 5, 10, 10,
0));
        Transaction t2 = new Transaction(UUID.randomUUID(), userId,
"Essen", new Amount(new BigDecimal("30.00")),
            TransactionType.AUSGABE, LocalDateTime.of(2025, 5, 12, 12,
0));
        when(transactionRepository.findByUserId(userId)).thenReturn(List.o
f(t1, t2));
        when(categoryRepository.findByUserId(userId)).thenReturn(List.of(
            new Category(UUID.randomUUID(), userId, new
CategoryName("Essen"), CategoryType.AUSGABE, new CategoryColor("rot"),
false)
        ));
        when(budgetRepository.findByUserId(userId)).thenReturn(List.of());


        List<MonthlyOverviewService.OverviewEntry> overview =
overviewService.getMonthlyOverview(userId, month);
        assertEquals(1, overview.size());
        assertEquals("Essen", overview.get(0).categoryName());
        assertEquals(new BigDecimal("80.00"), overview.get(0).spent());
        verify(transactionRepository, times(1)).findByUserId(userId);
}
```

### 4.4 Weitere getestete Komponenten

| Komponente | Getestete Aspekte |
|---|---|
| **UserService** | Registrierung, E-Mail-Prüfung, Benutzerabruf |
| **CategoryService** | Anlegen, Validierung von Namen/Farben |
| **BudgetService** | Setzen und Aktualisieren von Budgets |

### 4.5 Fazit

Die Unit Tests tragen wesentlich zur Robustheit und Änderbarkeit der Anwendung bei. Dank Mocks und klarer Schichtentrennung können Tests gezielt und isoliert erfolgen, ohne Abhängigkeiten zur echten Datenbank oder zur CLI.

## 5. Refactoring

Im Projekt *Monetenmanager* wurden mehrere gezielte Refactorings durchgeführt, um die Struktur, Verständlichkeit und Wartbarkeit der Anwendung zu verbessern. Dabei standen vor allem Clean Architecture, DDD und Entwicklerfreundlichkeit im Vordergrund.

### 5.1 Refactoring 1: Feature-basierte Ordnerstruktur (DDD + Clean Architecture)

**Problem:** Die ursprüngliche Struktur folgte keinem einheitlichen Konzept. Verantwortlichkeiten waren unscharf verteilt, Klassen lagen häufig in technischen statt fachlichen Kontexten.

**Maßnahme:** Die gesamte Projektstruktur wurde auf **Feature Packages** umgestellt. Jedes Feature wie transaction, user, category oder budget erhielt sein eigenes Package mit Unterteilungen für domain, service, repository und cli.

**Ergebnis:**

- Bessere Orientierung im Projekt
- Klar definierte Domänengrenzen
- Reduzierte Abhängigkeiten
- Grundlage für Clean Architecture gelegt

Git-Commits:
https://github.com/HabubMarcel/MonetenManager/commit/d0c36eeffa4fb9c954e152ef9cbe693a9b63b301

https://github.com/HabubMarcel/MonetenManager/commit/b38ae6d9d6cf6ac0490c2dc941c25973b3fdb9d1

### 5.2 Refactoring 2: Logging und CLI-UX-Verbesserung

**Problem:** CLI-Ausgaben waren inkonsistent, schwer lesbar und basierten auf System.out.println().

**Maßnahme:**

- Einführung von **SLF4J mit Logback** für strukturiertes Logging.
- Technische Ausgaben (Fehlermeldungen, Hinweise zur Ausführung etc.) erfolgen über Logger.info() / Logger.error() in den entsprechenden CLI-Handlern.
- Nutzersichtbare Ausgaben (Menüs, Texteingaben etc.) erfolgen weiterhin über System.out.println() bzw. System.out.printf() innerhalb der CLI-Handler.
- CLI-Dialoge wurden klarer formuliert (z. B. bei Passwortabfragen, Transaktionsübersicht, Eingabeaufforderungen).
- Hilfsmethoden für Passwortmaskierung und Wiederverwendbarkeit von Eingabelogik wurden eingeführt.

**Ergebnis:**

- Einheitliche Benutzererfahrung
- Einfachere Fehlersuche und Debugging
- Separation of Concerns zwischen Logik und Präsentation

Git-Commits:

https://github.com/HabubMarcel/MonetenManager/commit/b9cb1416166bb84e8273c3c882ba20c719a2d475

https://github.com/HabubMarcel/MonetenManager/commit/71690408d49905f77ceb7cef8c33ec550fd490ea

https://github.com/HabubMarcel/MonetenManager/commit/761c2a25bec5efce114249b0b3c369ed540d7e21

### 5.3 Refactoring: Einführung von Value Objects

**Problem:** Primitive Datentypen wie String für E-Mail, Beträge oder Kategorienamen boten keine Typsicherheit und erschwerten Validierung.

**Maßnahme:**

- Einführung von Email, Amount, CategoryName, CategoryColor als eigene Klassen.
- Eingebaute Validierung in den Konstruktoren.
- Verwendung in der gesamten Domain- und Service-Schicht.

**Ergebnis:**

- Mehr Typsicherheit
- Zentrale Validierung
- Weniger Wiederholungen, besserer Testfokus

Git-Commits:

https://github.com/HabubMarcel/MonetenManager/commit/f1dcb0bda08baaeaa0453a3c50584044073e7d7e

### 5.4 Identifizierte Code Smells und Auflösung

| Code Smell | Lösung durch Refactoring |
|---|---|
| **God Classes / Mixed Responsibility** | Aufgespalten in mehrere Services und CLI-Handler |
| **Primitive Obsession** | Value Objects eingeführt |
| **Verstreute Persistenzlogik** | In repository zentralisiert |
| **System.out in Domain-/App-Schichten** | Durch Logging ersetzt |

## 6. Entwurfsmuster

Begründung für den Einsatz des Factory Method Patterns

#### 1. Übersicht

In unserem Projekt muss der Benutzer mit verschiedenen Funktionalitäten der Anwendung über die Konsole interagieren, zum Beispiel zur Verwaltung von Kategorien, Budgets, Transaktionen oder zur Anzeige von Übersichten.

Dafür existieren verschiedene CLI-Handler-Klassen wie:

- UserCLIHandler
- CategoryCLIHandler
- TransactionCLIHandler
- OverviewCLIHandler usw.

Die CLIHandlerFactory wurde eingeführt, um eine zentrale und einheitliche Bereitstellung dieser Handler zu ermöglichen. Sie übernimmt die Rolle einer Factory, die fertig konfigurierte Handler kapselt und bereitstellt.

Ziel ist es, den Einstiegspunkt der Anwendung (MonetenmanagerApplication) übersichtlich zu halten und den Code von konkreten Handler-Abhängigkeiten zu entkoppeln.

### 6.2 Warum wird das Muster an dieser Stelle eingesetzt?

#### 1. Kapselung der Instanzbereitstellung

Anstatt alle Handler direkt im Einstiegspunkt zu erstellen zu müssen, stellt die CLIHandlerFactory sie gesammelt bereit. Die Anwendung benötigt nur ein einziges Objekt (CLIHandlerFactory), um auf alle CLI-Funktionen zuzugreifen.

#### 2. Zentrale Verwaltung und klare Struktur

Wenn sich die Handler oder deren Konstruktoren ändern, muss dies nur an einer Stelle – in der Factory – angepasst werden. Dadurch bleibt der übrige Code stabil und übersichtlich.

#### 3. Testbarkeit und Erweiterbarkeit

Neue CLI-Komponenten (z. B. ReportCLIHandler) können ohne große Umbauten über die Factory eingeführt werden. In Tests können alternative Handler-Factories genutzt werden (z. B. mit Mocks oder Stubs).

### 6.3 Wie verbessert das Muster den Code?

- **Reduzierte Redundanz**

Alle CLI-Handler werden an zentraler Stelle initialisiert. Dadurch entfällt die wiederholte Instanziierung oder Konfiguration in verschiedenen Teilen der Anwendung.

- **Klar strukturierte Verantwortlichkeiten**

Die Anwendung (main) ist von den Details der Handler-Verwaltung entkoppelt. Sie kennt nur die Factory, nicht aber die internen Abhängigkeiten der einzelnen CLI-Komponenten.

- **Gute Erweiterbarkeit**

Neue CLI-Handler lassen sich problemlos hinzufügen, ohne bestehende Logik verändern zu müssen. Die Factory dient dabei als „Plug-and-Play"-Schnittstelle für neue Funktionen.

### 6.4 Vorteile und Nachteile durch den Einsatz des Musters

**Vorteile**

- Trennung von Erstellung und Verwendung: Die Factory kümmert sich um die Handler-Vorbereitung, die Anwendung nutzt sie nur.
- Zentrale Konfiguration: Änderungen an Konstruktoren oder Abhängigkeiten betreffen nur die Factory.
- Erweiterbar: Neue CLI-Bereiche lassen sich leicht integrieren.
- Einheitliche Struktur: Der Einstiegspunkt bleibt schlank und übersichtlich.

**Nachteile**

- Zusätzliche Abstraktionsebene, die in sehr kleinen Anwendungen eventuell überdimensioniert erscheint.
- Factory erzeugt die Objekte nicht dynamisch – sondern verwaltet vorbereitete Instanzen (was vom klassischen Pattern leicht abweicht).

**Git-Commit:**

https://github.com/HabubMarcel/MonetenManager/commit/a8221b05be6c982570c9503957816bf29db7b220