DHBW
Duale Hochschule
Baden-Württemberg
Karlsruhe

Advanced Software Engineering

Programmentwurf - Protokoll

von

Lukas Hertkorn (6636218)

Niklas Raphael Krauth (2562445)

Lukas Melcher (3812765)

Abgabedatum 04. Mai 2025

Inhaltsverzeichnis

1 Einführung (4P)

1.1 Übersicht über die Applikation (1P)

[Was macht die Applikation? Wie funktioniert sie? Welches Problem löst sie/welchen Zweck hat sie?]

Bilanzius ist eine Finanz-App, mit der Nutzer ihre persönlichen Finanzen einfach verwalten können. Jeder Nutzer kann mehrere Bankkonten anlegen und Geldbeträge sowie Buchungen verwalten. Außerdem lassen sich Ausgaben und Einnahmen verschiedenen Kategorien zuordnen, um einen besseren Überblick über das eigene Geld zu behalten. Die App bietet Funktionen wie das Erstellen neuer Transaktionen, das Anlegen und Löschen von Konten sowie eine einfache Verwaltung von Kategorien. Zusätzlich gibt es ein kleines Berichtssystem, das Finanzübersichten in Form von Tabellen ausgeben kann.

1.2 Starten der Applikation (1P)

[Wie startet man die Applikation? Was für Voraussetzungen werden benötigt? Schritt-für-Schritt-Anleitung]

Gebaute App:

- Im Terminal *java –jar bilanzius.jar* ausführen

Code:

- Projekt in preferierter IDE öffnen
- Datei org.bilanzius.Main öffnen
- main() Funktion über die IDE starten

1.3 Technischer Überblick (2P)

[Nennung und Erläuterung der Technologien (z.B. Java, MySQL, ...), jeweils Begründung für den Einsatz der Technologien]

- Java – In der Vorlesung verwendete Programmiersprache.
- SQLite – Lokale Datenbank, die keine extra Installation benötigt.
- Maven – Dependency Manager sowie Build Tool
- GSON – Bibliothek, um JSON-Daten zu verwalten
- JUnit - Als Framework für automatisierte Softwaretests

2 Softwarearchitektur (8P)

2.1 Gewählte Architektur (4P)

[In der Vorlesung wurden Softwarearchitekturen vorgestellt. Welche Architektur wurde davon umgesetzt? Analyse und Begründung inkl. UML der wichtigsten Klassen, sowie Einordnung dieser Klassen in die gewählte Architektur]

Analyse

Das Projekt Bilanzius basiert auf einem klassischen 3-Schichtenmodell. Ziel dieser Architektur ist es, die Verantwortlichkeiten klar zu trennen und eine lose Kopplung zwischen den Komponenten zu ermöglichen. Die drei Hauptschichten sind:

1. **Präsentationsschicht**
   Diese Schicht stellt die Schnittstelle zur Benutzerinteraktion dar. In unserem Fall besteht sie hauptsächlich aus der `Main`-Klasse, die Benutzereingaben verarbeitet und an die entsprechenden Komponenten weiterleitet.

2. **Domänenschicht (Domain Layer)**
   Die zentrale Schicht enthält die gesamte Geschäftslogik. Hier befinden sich:

   - Die **Command-Klassen**: Jede Klasse repräsentiert einen ausführbaren Befehl (z.B. `/help`, `/exit`).
   - Der **CommandController**: Dieser dient als Vermittler und Dispatcher zwischen Eingaben der Präsentationsschicht und ausführbaren Commands.
   - Die **Service-Interfaces** wie BankAccountService, UserService usw., die fachliche Operationen definieren.

3. **Datenschicht (Data Layer)**
   Diese Schicht kapselt den Zugriff auf die persistente Datenhaltung (SQLite). Die konkreten Implementierungen der Service-Interfaces (`SqliteBankAccountService`, `SqliteUserDatabaseService`, usw.) befinden sich hier. Sie sind über das Repository `DatabaseServiceRepository` zentral angebunden.

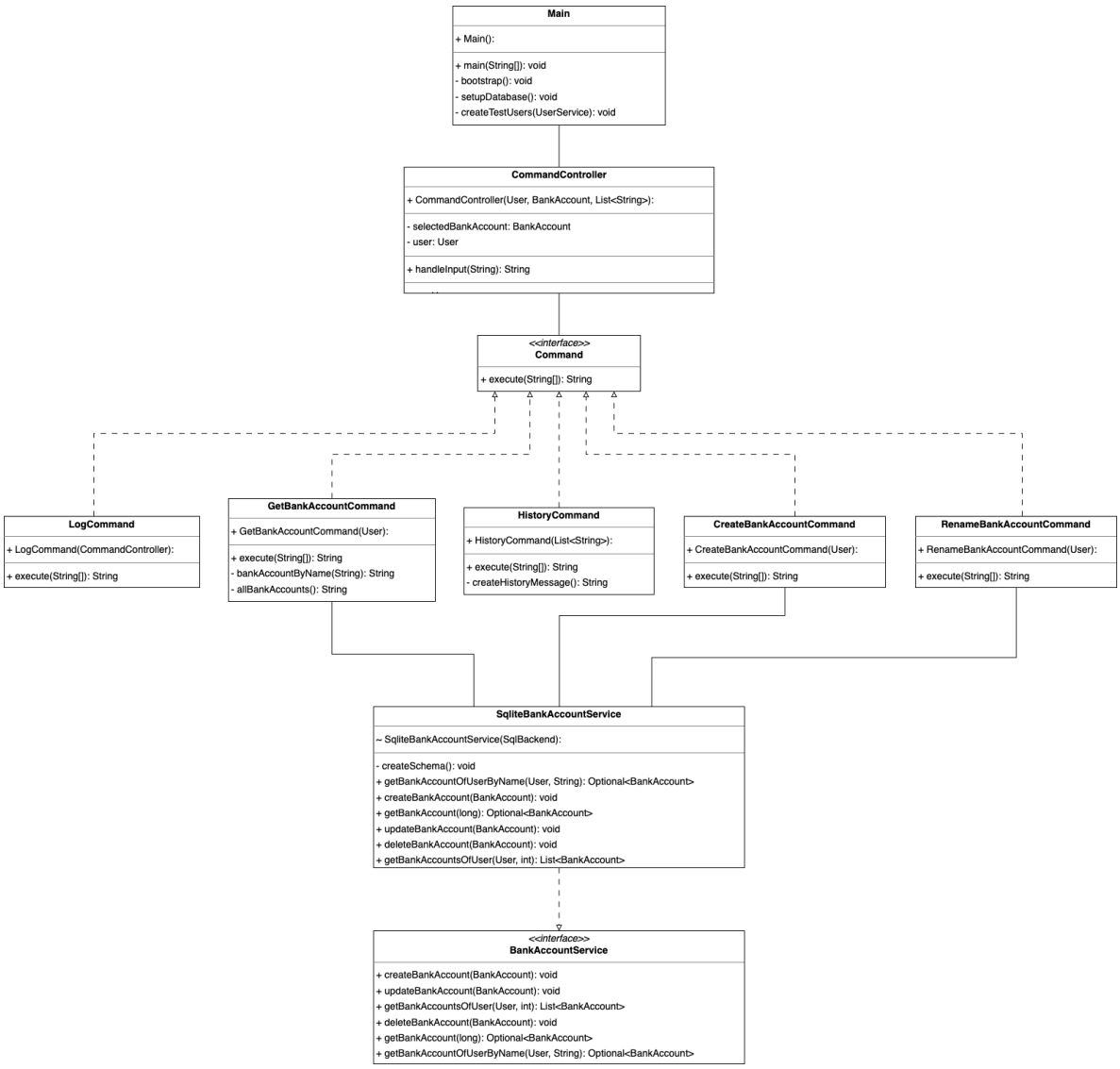
Begründung

Wir haben uns für dieses Schichtenmodell entschieden, weil es:

- eine klare Strukturierung bei wachsender Komplexität bietet
- dadurch einfach erweiterbar ist
- eine lose Kopplung zwischen den Komponenten bietet
- gut mit den Prinzipien von Domain Driven Design kombinierbar ist

UML-Diagramm

Das UML zeigt einen Ausschnitt der Klassen und ihre Einteilung zu den drei Schichten. Zur verbesserten Übersicht sind nicht alle Klassen und Interfaces aufgeführt, da dies zu groß wäre, um noch was zu erkennen zu können.



2.2 Domain Code (1P)

[kurze Erläuterung in eigenen Worten, was Domain Code ist – 1 Beispiel im Code zeigen, das bisher noch nicht gezeigt wurde]

Domain Code ist der Teil einer Anwendung, der die eigentliche **Geschäftslogik** beschreibt – also die Regeln, Abläufe und Begriffe der realen Welt, die mit der Software abgebildet werden sollen. Im Gegensatz zu technischem Code (z.B. Datenbank, UI) steht hier die fachliche Bedeutung im Vordergrund. Der Domain Code ist direkt von der Problemdomäne inspiriert und sollte möglichst klar und unabhängig von der Infrastruktur sein.

org.bilanzius.persistence.models.Transaction

```
1  public static Transaction create(User user, BankAccount account, BigDecimal money, String description)
2  {
3      return new Transaction(0,
4              user.getId(), account.getAccountId(), -1, money, Instant.now(), description
5      );  // -1 as default categoryId
6  }
```

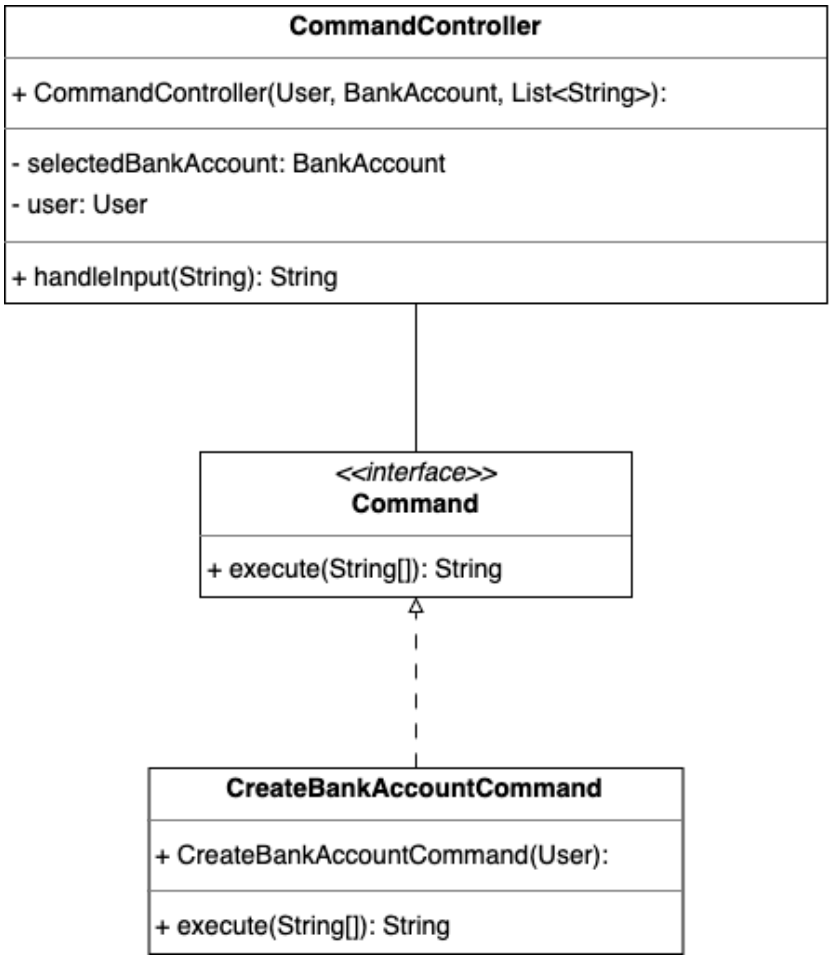
2.3 Analyse der Dependency Rule (3P)

[In der Vorlesung wurde im Rahmen der 'Clean Architecture' die s.g. Dependency Rule vorgestellt. Je 1 Klasse zeigen, die die Dependency Rule einhält und 1 Klasse, die die Dependency Rule verletzt; jeweils UML (mind. die betreffende Klasse inkl. der Klassen, die von ihr abhängen bzw. von der sie abhängt) und Analyse der Abhängigkeiten in beide Richtungen (d.h., von wem hängt die Klasse ab und wer hängt von der Klasse ab) in Bezug auf die Dependency Rule]

Positiv-Beispiel: Dependency Rule

Der `CommandController` verarbeitet Nutzereingaben und delegiert die Ausführung an sogenannte Commands. Dabei kennt er nur das **Interface** `Command` – und nicht die konkreten Implementierungen. Das bedeutet:

- Die Abhängigkeit zeigt nach innen, also zur Domänenschicht.
- Der Controller bleibt unabhängig von technischen Details.
- Konkrete Commands können sich ändern oder erweitert werden, ohne dass der CommandController geändert werden muss.



Hinweis: Weitere Commands wie `HelpCommand`, `ExitCommand` usw. implementieren ebenfalls das Interface `Command`, sind aber aus Platzgründen nicht dargestellt.

Negativ-Beispiel: Dependency Rule

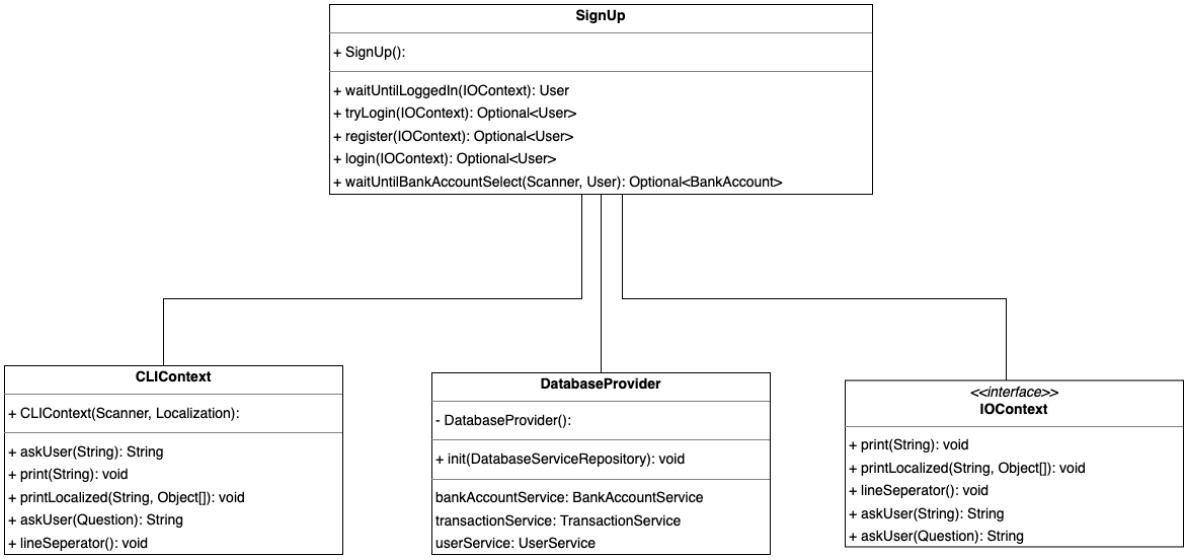
Die `SignUp`-Klasse ist verantwortlich für die Registrierung bzw. Anmeldung von Nutzern. Allerdings vermischt sie mehrere Ebenen der Anwendung in einer einzigen Klasse:

- Sie greift auf **Datenbankservices** zu (`DatabaseProvider`, `UserService`).
- Sie interagiert mit dem **CLI-Interface** (`IOContext`).
- Sie verwendet **Utility-Logik** (`Localization`).

Dadurch verletzt `SignUp` die Dependency Rule gleich mehrfach:

- Die Klasse kennt sowohl **Domänen- als auch Infrastrukturelemente**.
- **Abhängigkeiten verlaufen in beide Richtungen**, was die Wartbarkeit und Testbarkeit erschwert.

Hier hängt die fachlich zentrale Klasse direkt von mehreren äußeren Schichten ab – Verletzung der Clean Architecture Grundsätze.

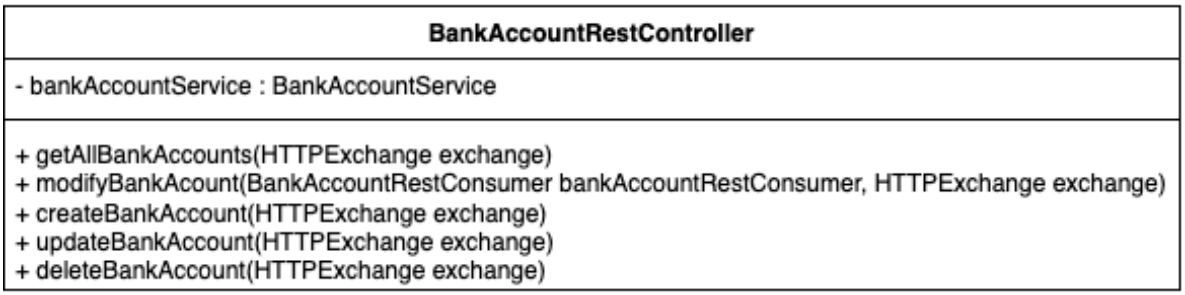


3 SOLID (8P)

3.1 Analyse SRP (3P)

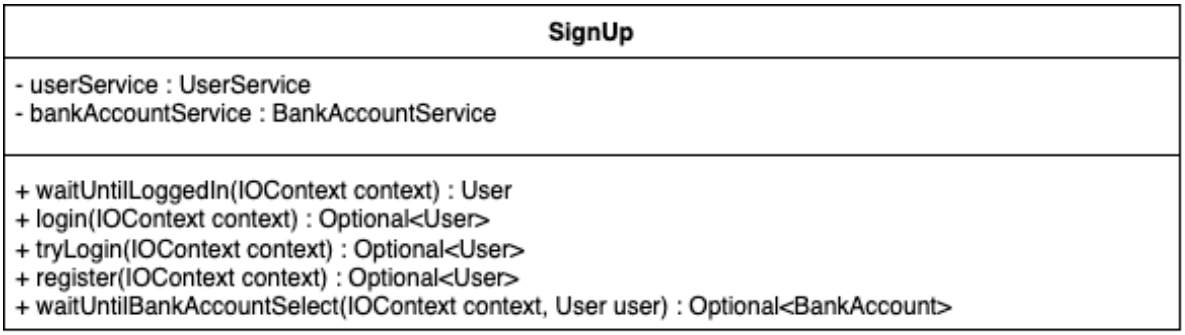
[jeweils eine Klasse als positives und negatives Beispiel für SRP; jeweils UML und Beschreibung der Aufgabe bzw. der Aufgaben und möglicher Lösungsweg des Negativ-Beispiels (inkl. UML)]

Positiv-Beispiel



Der BankAccountRestController ist eine Klasse, welche die CRUD-Endpunkte für die Bankkonten anbietet.

Negativ-Beispiel



Die SingUp-Klasse ist für das Anmelden und Registrieren eines Benutzers vor dem Programm dar. Dadurch das die Methoden waitUntilLoggedIn() sowie waitUntilBankAccountIsSelect() eher weniger mit dem Anmelden zu tun haben, müsste man diese in eine neue Klasse wie zum Beispiel WaitUntil extrahieren.

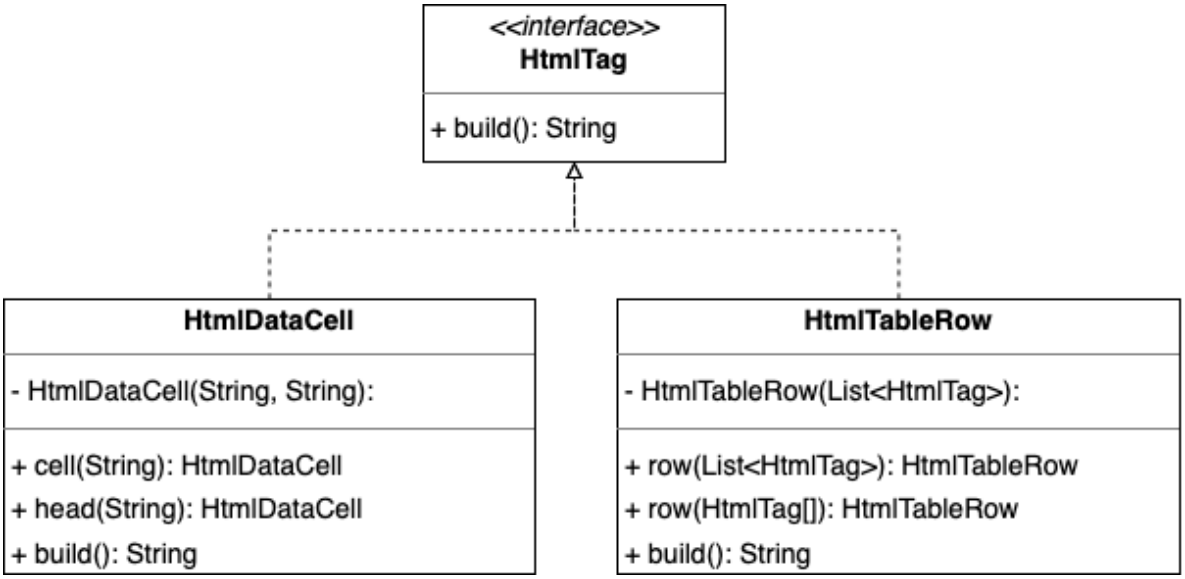
3.2 Analyse OCP (3P)

[jeweils eine Klasse als positives und negatives Beispiel für OCP; jeweils UML und Analyse mit Begründung, warum das OCP erfüllt/nicht erfüllt wurde – falls erfüllt: warum hier sinnvoll/welches Problem gab es? Falls nicht erfüllt: wie könnte man es lösen (inkl. UML)?]

Positiv-Beispiel

Die Klasse `HTMLTag` wurde im Projekt als Interface definiert, das eine einheitliche Methode `build()` bereitstellt. Alle HTML-Komponenten (wie `HTMLTableRow`, `HTMLDataCell` oder `HTMLHeaderCell`) implementieren dieses Interface. Dadurch können neue HTML-Elemente flexibel ergänzt werden, ohne die bestehende Logik anzupassen.
Beispielsweise kann der `FinanceReportBuilder` über eine Liste von HTMLTag-Elementen iterieren

und sie per `build()` zu HTML umwandeln. Dies ist dabei unabhängig von der konkreten Implementierung.
Dieses Design erfüllt das Open/Closed Principle, da die Funktionalität durch neue Klassen erweitert, aber das bestehende Interface und die Verarbeitung nicht verändert werden müssen.
Warum hier sinnvoll? Das HTML-Reporting soll flexibel auf neue Anforderungen reagieren können (z.B. neue Zeilentypen oder Zellenformate). Mit dem Interface-Ansatz kann dies erfolgen, ohne dass zentrale Komponenten wie der `ReportBuilder` angepasst werden müssen.



Negativ Beispiel

Die Klasse `Localization` ist für die Übersetzung von Texten in verschiedene Sprachen zuständig. Sie verwendet intern `ResourceBundle`, um Übersetzungen aus `.properties`-Dateien basierend auf einem Sprachcode zu laden (`messages_en.properties`, `messages_de.properties`)

Warum wird das OCP hier verletzt?
Die Klasse ist nicht offen für Erweiterungen, sondern muss bei jeder neuen Anforderung direkt geändert werden. Dadurch verstößt sie gegen das Open/Closed Principle:

- Neue Sprachen (z.B. „fr“) müssen im Code zur Liste `supportedLanguages` hinzugefügt werden.
- Neue Übersetzungsquellen (z.B. JSON-Dateien, Datenbank, REST-API) erfordern Änderungen an der Methode `setLocale(...)`, da dort `ResourceBundle` fest eingebunden ist.
- Eine dynamische Erweiterung oder das Nachladen von Übersetzungen zur Laufzeit ist nicht möglich, ohne die bestehende Klasse zu verändern.

Lösungsansatz:
Um das Open/Closed Principle einzuhalten, sollte die eigentliche Übersetzungslogik in ein Interface ausgelagert werden, z.B. `TranslationProvider`. Damit könnten unterschiedliche Implementierungen genutzt werden, ohne `Localization` ändern zu müssen:

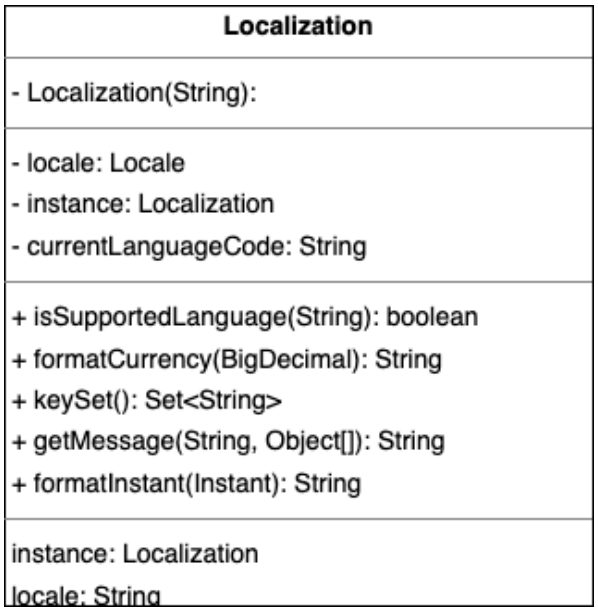
```
1 public interface TranslationProvider {
2     String getMessage(String key, Object... args);
3     Set<String> keySet();
4 }
```

Die Localization-Klasse müsste dann nur noch ein solches Interface nutzen – und wäre offen für Erweiterung, aber geschlossen für Modifikation. Beispielhafte Implementierungen könnten dann die folgenden sein:

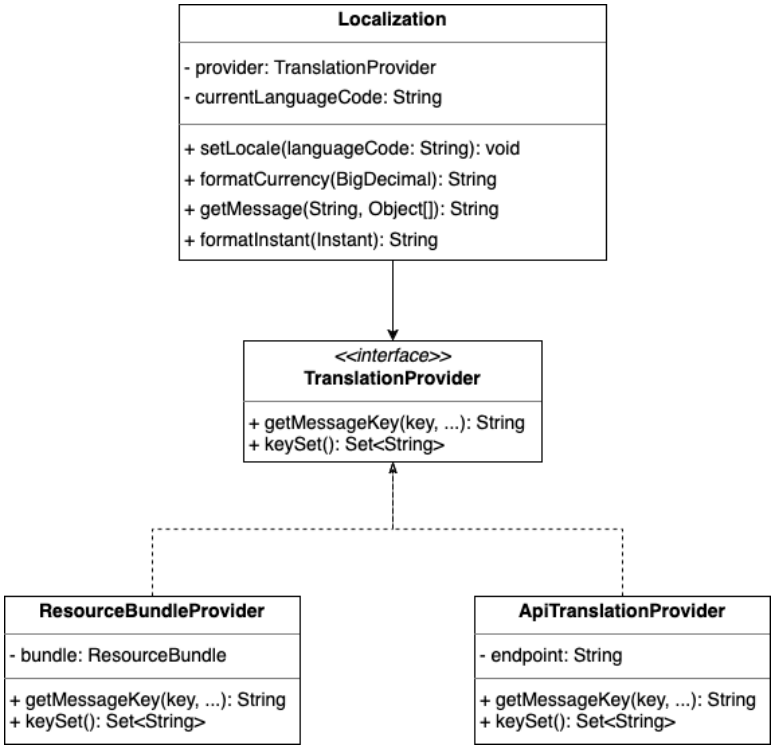
- ResourceBundleTranslationProvider
- JsonTranslationProvider
- ApiTranslationProvider

UML-Vergleich

Ist-Zustand:



Nachher:



3.3 Analyse [LSP/ISP/DIP] (2P)

[jeweils eine Klasse als positives und negatives Beispiel für entweder LSP oder ISP oder DIP; jeweils UML und Begründung, warum hier das Prinzip erfüllt/nicht erfüllt wird; beim Negativ-Beispiel UML einer möglichen Lösung hinzufügen]

[Anm.: es darf nur ein Prinzip ausgewählt werden; es darf NICHT z.B. ein positives Beispiel für LSP und ein negatives Beispiel für ISP genommen werden]

Gewähltes Prinzip: ISP

Positiv-Beispiel

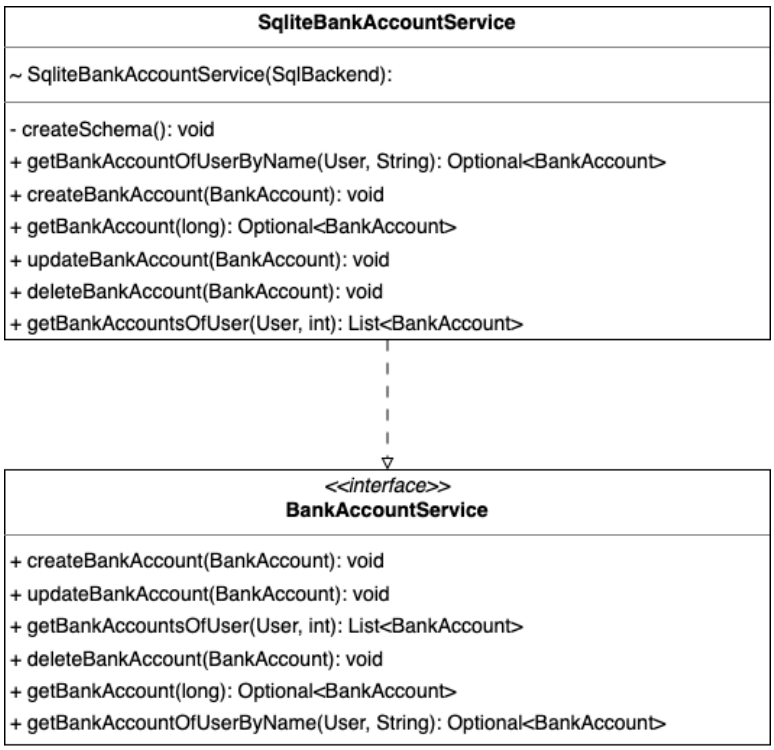
Die Schnittstelle `BankAccountService` definiert ausschließlich Methoden rund um Bankkonten und hat somit eine sehr saubere Trennung: Die Schnittstelle ist spezifisch und auf eine einzige Domäne fokussiert.

```java
public interface BankAccountService {
    void createBankAccount(BankAccount bankAccount);
    Optional<BankAccount> getBankAccount(long id);
    List<BankAccount> getBankAccountsOfUser(User user, int limit);
    void updateBankAccount(BankAccount bankAccount);
    void deleteBankAccount(BankAccount bankAccount);
}
```

Begründung

- Interface ist spezialisiert und damit gut für Testbarkeit und Wartung
- Folgeprinzip: niedrige Kopplung, hohe Kohäsion

UML



Negativ-Beispiel

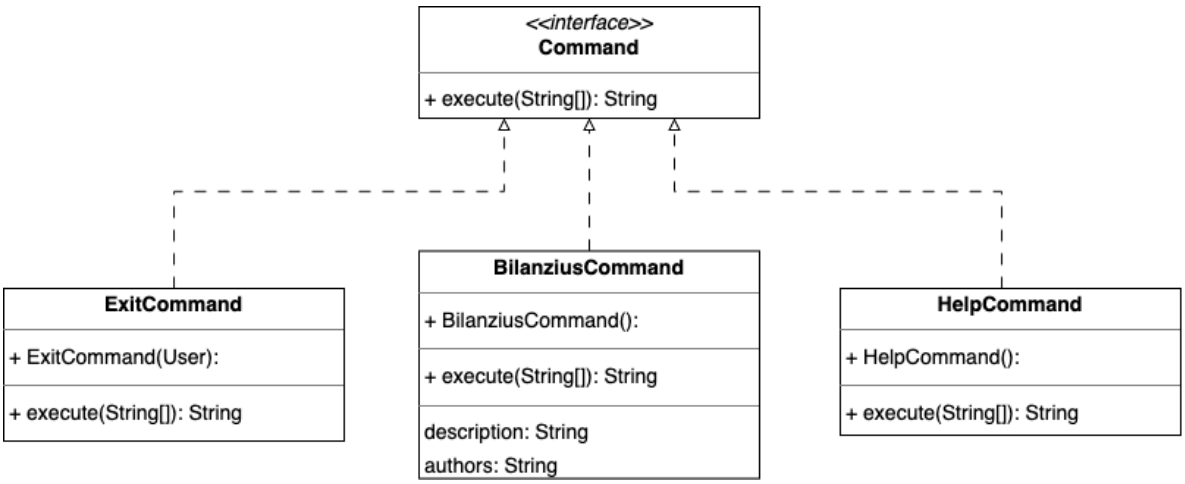
Das Command-Interface wird von allen Befehlen im Terminal verwendet und sieht so aus:

```java
public interface Command {
    String execute(String[] arguments);
}
```

Das wirkt erstmal schlank, aber: In der Praxis gibt es viele Commands, die keine Argumente brauchen, oder deren Verwendung durch das Interface erzwungen wird, obwohl sie sie gar nicht nutzen.
Begründung

- Ein Interface sollte nur das vorschreiben, was für den jeweiligen Fall zwingend nötig ist.
- `Command` ist zu allgemein – spezialisierte Commands (z.B. Info-Commands vs. System-Commands) sollten getrennte Interfaces bekommen.
- Verstöße gegen ISP führen oft zu optionalen, leeren oder irrelevanten Implementierungen → das ist ein Code-Smell.

UML - Ist-Zustand



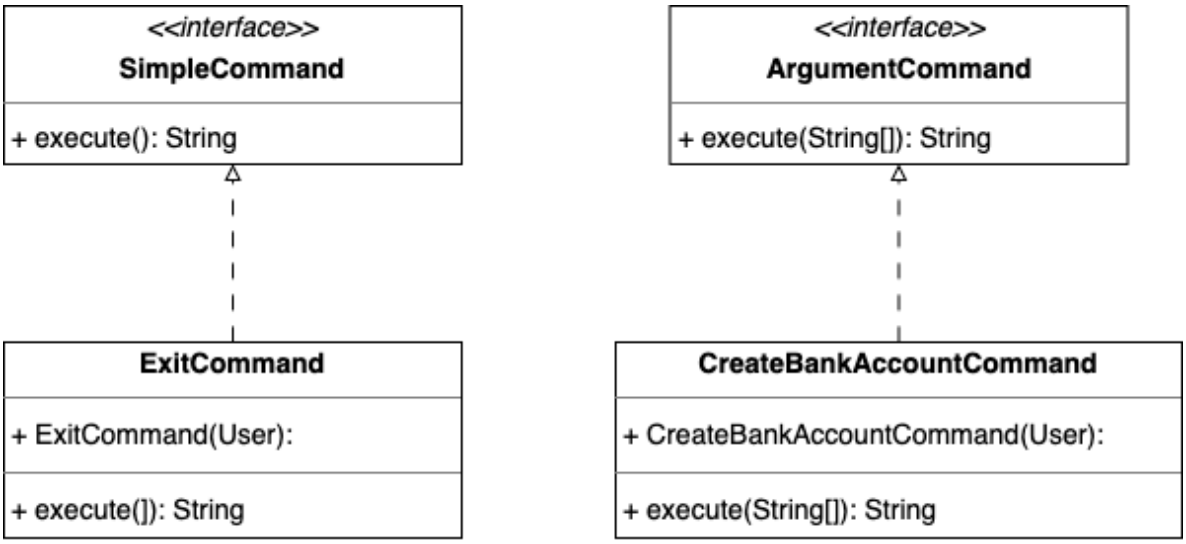
Verbesserungsvorschlag

Trennen von `Command` in spezifischere Interfaces, z.B.:

```java
public interface SimpleCommand {
    String execute();
}

public interface ArgumentCommand {
    String execute(String[] args);
}
```

Dann können z.B. `HelpCommand` und `ExitCommand` `SimpleCommand` implementieren, während z.B. ein **CreateCategoryCommand** ein ArgumentCommand implementiert.

UML Verbesserung

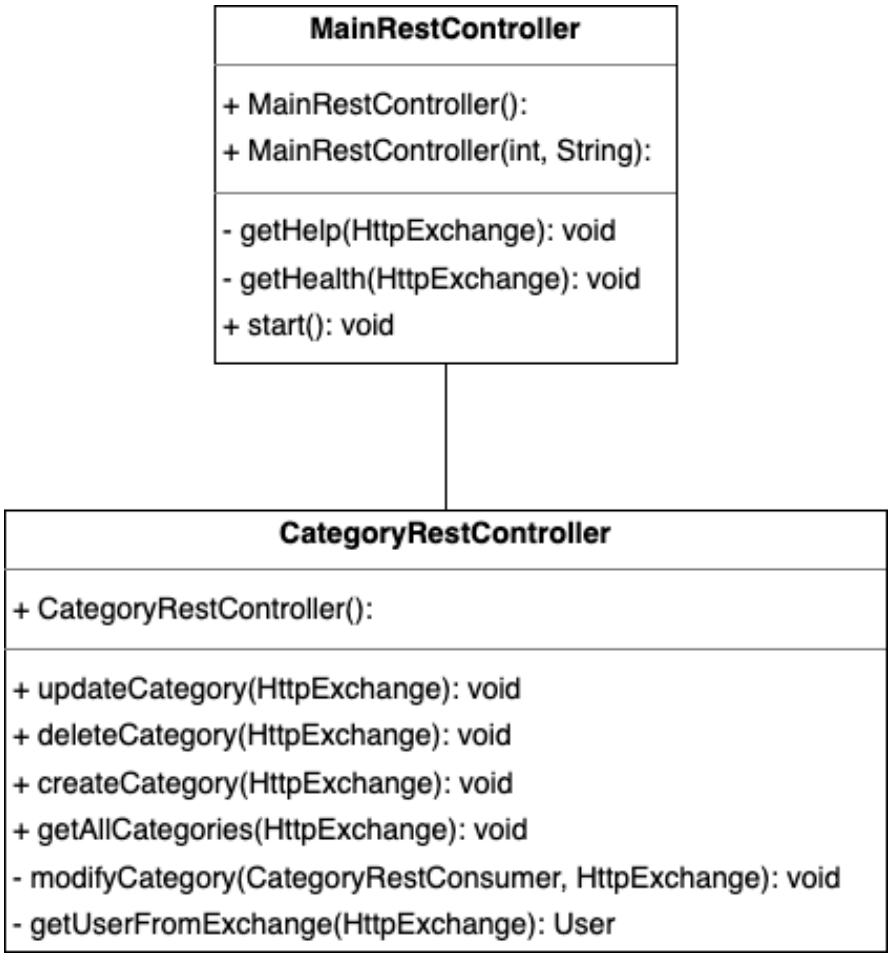

<<interface>>
SimpleCommand
+ execute(): String
ExitCommand
+ ExitCommand(User):
+ execute(]): String
<<interface>>
ArgumentCommand
+ execute(String[]): String
CreateBankAccountCommand
+ CreateBankAccountCommand(User):
+ execute(String[]): String

4 Weitere Prinzipien (8P)

4.1 Analyse GRASP: Geringe Kopplung (3P)

[eine bis jetzt noch nicht behandelte Klasse als positives Beispiel geringer Kopplung; UML mit zusammenspielenden Klassen, Aufgabenbeschreibung der Klasse und Begründung, warum hier eine geringe Kopplung vorliegt; es müssen auch die Aufrufer/Nutzer der Klasse berücksichtigt werden]

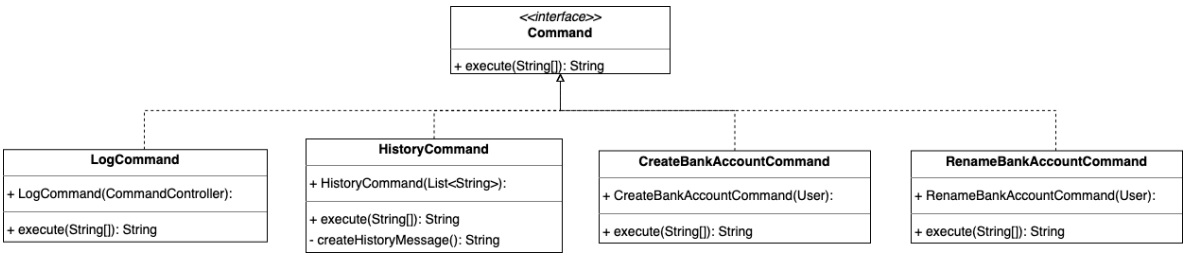
Die Klasse CategoryRestController ist für die Anbindung der CRUD-Endpunkte für den HTTP-Server für alle Kategorien eines Nutzers. Sie hat eine geringe Kopplung da sie nur für CRUD-Endpunkte für Kategorien dar sind. Davor waren alle Endpunkte im MainRestController.



4.2 Analyse GRASP: [Polymorphismus/Pure Fabrication] (3P)

[eine Klasse als positives Beispiel entweder von Polymorphismus oder von Pure Fabrication; UML Diagramm und Begründung, warum es hier zum Einsatz kommt]

Damit alle Commands gleich sind und keine Methoden fehlen wurde ein Interface geschrieben, welches jeder Command erbt.



4.3 DRY (2P)

[ein Commit angeben, bei dem duplizierter Code/duplizierte Logik aufgelöst wurde; Code-Beispiele (vorher/nachher) einfügen; begründen und Auswirkung beschreiben – ggf. UML zum Verständnis ergänzen]

Branch: *main/Commit bc6ad15*

Vorher

```
private String convertToBitcoin()
{

    JsonObject jsonObject = Requests.getRequest(currencyUrl);
    assert jsonObject != null;
    BigDecimal exchangeRate = BigDecimal.valueOf(getCurrencyFromJson(jsonObject, "btc"));
    BigDecimal balance;

    try {
        balance = bankAccountService.getBankAccount(selectedBankAccount.getAccountId()).orElseThrow().
     getBalance();
    } catch (DatabaseException e) {
        return localization.getMessage("database_error", e.toString());
    }

    return localization.getMessage("convert_balance", "Bitcoin", (balance.multiply(exchangeRate)));
}

private String convertToGermanDeutscheMark()
{

    JsonObject jsonObject = Requests.getRequest(currencyUrl);
    assert jsonObject != null;
    BigDecimal exchangeRate = BigDecimal.valueOf(getCurrencyFromJson(jsonObject, "dem"));
    BigDecimal balance;
       try {
        balance = bankAccountService.getBankAccount(selectedBankAccount.getAccountId()).orElseThrow().
     getBalance();
    } catch (DatabaseException e) {
        return localization.getMessage("database_error", e.toString());
    }

    return localization.getMessage("convert_balance", "Deutsche Mark", (balance.multiply(exchangeRate)))
     ;
}

private String convertToSwissFranc()
{

    String currency = "Swiss Franc";
    if (localization.getCurrentLanguageCode().equals("de")) {
        currency = "Schweizer Franken";
    }

    JsonObject jsonObject = Requests.getRequest(currencyUrl);
    assert jsonObject != null;
    BigDecimal exchangeRate = BigDecimal.valueOf(getCurrencyFromJson(jsonObject, "chf"));
    BigDecimal balance;
```

```
46
47      try {
48          balance = bankAccountService.getBankAccount(selectedBankAccount.getAccountId()).orElseThrow().
        getBalance();
49      } catch (DatabaseException e) {
50          return localization.getMessage("database_error", e.toString());
51      }
52
53      return localization.getMessage("convert_balance", currency, (balance.multiply(exchangeRate)));
54  }
55
56  private String convertToDogecoin()
57  {
58
59      JsonObject jsonObject = Requests.getRequest(currencyUrl);
60      assert jsonObject != null;
61      BigDecimal exchangeRate = BigDecimal.valueOf(getCurrencyFromJson(jsonObject, "doge"));
62      BigDecimal balance;
63
64      try {
65          balance = bankAccountService.getBankAccount(selectedBankAccount.getAccountId()).orElseThrow().
        getBalance();
66      } catch (DatabaseException e) {
67          return localization.getMessage("database_error", e.toString());
68      }
69
70      return localization.getMessage("convert_balance", "Deutsche Mark", (balance.multiply(exchangeRate)))
        ;
71  }
72
73  private String convertToSwissFranc()
74  {
75
76      String currency = "Swiss Franc";
77      if (localization.getCurrentLanguageCode().equals("de")) {
78          currency = "Schweizer Franken";
79      }
80
81      JsonObject jsonObject = Requests.getRequest(currencyUrl);
82      assert jsonObject != null;
83      BigDecimal exchangeRate = BigDecimal.valueOf(getCurrencyFromJson(jsonObject, "chf"));
84      BigDecimal balance;
85
86      try {
87          balance = bankAccountService.getBankAccount(selectedBankAccount.getAccountId()).orElseThrow().
        getBalance();
88      } catch (DatabaseException e) {
89          return localization.getMessage("database_error", e.toString());
90      }
91
92      return localization.getMessage("convert_balance", currency, (balance.multiply(exchangeRate)));
93  }
94
95  private String convertToDogecoin()
96  {
97
98      JsonObject jsonObject = Requests.getRequest(currencyUrl);
99      assert jsonObject != null;
100     BigDecimal exchangeRate = BigDecimal.valueOf(getCurrencyFromJson(jsonObject, "doge"));
101     BigDecimal balance;
102
103     try {
104         balance = bankAccountService.getBankAccount(selectedBankAccount.getAccountId()).orElseThrow().
        getBalance();
105     } catch (DatabaseException e) {
106         return localization.getMessage("database_error", e.toString());
107     }
108 }
```

Nachher

```
private String convertCurrency(String currencyCode, String currencyName) {
    JsonObject jsonObject = Requests.getRequest(currencyUrl);
    assert jsonObject != null;
    BigDecimal exchangeRate = BigDecimal.valueOf(getCurrencyFromJson(jsonObject, currencyCode));
    BigDecimal balance;

    try {
        balance = bankAccountService.getBankAccount(selectedBankAccount.getAccountId()).orElseThrow().
     getBalance();
    } catch (DatabaseException e) {
        return localization.getMessage("database_error", e.toString());
    }

    return localization.getMessage("convert_balance", currencyName, (balance.multiply(exchangeRate)));
}

private String convertToBitcoin()
{
    return convertCurrency("btc", "Bitcoin");
}

private String convertToGermanDeutscheMark()
{
    return convertCurrency("dem", "Deutsche Mark");
}

private String convertToSwissFranc()
{
    String currency = localization.getCurrentLanguageCode().equals("de") ? "Schweizer Franken" : "Swiss
     Franc";
    return convertCurrency("chf", currency);
}

private String convertToDogecoin()
{
    return convertCurrency("doge", "Dogecoin");
}

private String convertToEthereum()
{
    return convertCurrency("eth", "Ethereum");
}

private String convertToHongKongDollar()
{
    return convertCurrency("hkd", "Hong Kong Dollar");
}

private String convertToJamaicanDollar()
{
    String currency = localization.getCurrentLanguageCode().equals("de") ? "Jamaikanische Dollar" : "
     Jamaican Dollar";
    return convertCurrency("jmd", currency);
}

private String convertToNorthKoreanWon()
{
    String currency = localization.getCurrentLanguageCode().equals("de") ? "Nordkoreanische Won" : "
     North Korean Won";
    return convertCurrency("kpw", currency);
}

private String convertToRussianRuble()
{
    String currency = localization.getCurrentLanguageCode().equals("de") ? "Russischer Rubel" : "Russian
      Ruble";
    return convertCurrency("rub", currency);
}

private String convertToUsDollar()
{
```

```
67      return convertCurrency("usd", "US Dollar");
68  }
```

Beschreibung

Es gab viele Methoden die, denn selben Code in einer kleinen abgeänderten Art ausgeführt haben. Die Lösung war eine Methode namens *convertCurrency(String currencyCode, String currencyName)* die in den einzelnen Methoden für die jeweilige Währung aufgerufen wird.

Das hatte zur Folge das der Code einfacherer zu lesen ist und unnötige Codezeilen gespart werden konnten.

5 Unit Tests (8P)

5.1 10 Unit Tests (2P)

[Zeigen und Beschreiben von 10 Unit-Tests und Beschreibung, was getestet wird]

1. **Find User by Name Test (SqliteUserDatabaseServiceTest)**
   Erstelle einen neuen Nutzer mit dem Name Test und prüfe ob dieser über die findUserWithName Funktion gefunden werden kann.

```
@Test
void testFindUserByName()
{
    // Setup test
    var service = userService();
    service.createUser(User.createUser(USERNAME, DEFAULT_PASSWORD));
    var test = "hallo";
    // Find user and validate
    var result = service.findUserWithName(USERNAME).orElseThrow();

    Assertions.assertEquals(1, result.getId());
    Assertions.assertEquals(USERNAME, result.getUsername());
    Assertions.assertEquals(DEFAULT_PASSWORD, result.getHashedPassword());
}
```

2. **Update User Password Test (SqliteUserDatabaseServiceTest)**
   Erstelle einen neuen Nutzer mit dem Name Test und einem Standard-Passwort. Das Passwort wird daraufhin geupdatet. Anschließend wird geprüft das neue Passwort gesetzt wurde.

```
@Test
void testUpdateUserPassword()
{
    // Setup test
    var service = userService();
    var creatingUser = User.createUser(USERNAME, DEFAULT_PASSWORD);

    // user can't be updated before creation
    Assertions.assertFalse(creatingUser.canBeUpdated());

    service.createUser(creatingUser);

    // Find user and update password
    var user = service.findUser(1).orElseThrow();
    Assertions.assertTrue(user.canBeUpdated());
    user.setHashedPassword(OTHER_PASSWORD);
    service.updateUserPassword(user);

    // Find user again
    var sameUser = service.findUser(1).orElseThrow();
    Assertions.assertEquals(OTHER_PASSWORD, sameUser.getHashedPassword());
}
```

3. **Get Localized Message Test (LocalizationTest)**
   Prüfe ob die Übersetzungsfunktion richtig funktioniert. In diesem Test wird geprüft, ob ein existenter Language Key richtig aufgelöst werden kann.

```
@Test
void testGetMessage()
{
    var language = Localization.getInstance();
    language.setLocale("en");

    Assertions.assertEquals("en", language.getCurrentLanguageCode());
    Assertions.assertEquals(EXISTING_KEY_VALUE, language.getMessage(EXISTING_KEY));
    Assertions.assertEquals("MISSING KEY: " + NON_EXISTING_KEY, language.getMessage(
     NON_EXISTING_KEY));
}
```

4. **Get Localized Message with Parameters (LocalizationTest)**
   Prüfe ob eine Übersetzung mit Parametern gefüllt werden kann.

```
@Test
void testGetMessageWithParams()
{
    var language = Localization.getInstance();
    language.setLocale("en");

    Assertions.assertEquals("en", language.getCurrentLanguageCode());
    Assertions.assertEquals(EXISTING_KEY_WITH_PARAMS_VALUE, language.getMessage(
     EXISTING_KEY_WITH_PARAMS, "1"));
}
```

5. **Table Cell Tests (HtmlDataCellTest)**
   Die HtmlDataCell Klasse generiert HTML Tabellen Tags für das Finanzbericht Report Feature. Es wird getestet, ob die Klasse gültige HTML-Tags ausgibt.

```
@Test
void testTableElements()
{
    var headComponent = HtmlDataCell.head("head");
    var dataComponent = HtmlDataCell.cell("data");

    Assertions.assertEquals("<th>head</th>", headComponent.build());
    Assertions.assertEquals("<td>data</td>", dataComponent.build());
}
```

6. **Create and Get Transaction (SqliteTransactionServiceTest)**
   Erstelle eine Zahlungstransaktion für einen Beispielnutzer und prüfe ob diese richtig gespeichert wurde.

```
@Test
void testCreateTransaction()
{
    var service = transactionService();
    service.saveTransaction(new Transaction(1, 1, 1, 1, BigDecimal.ZERO, Instant.now(), "1234"));
    service.saveTransaction(new Transaction(1, 1, 1, -1, BigDecimal.ZERO, Instant.now(), "1234"));

    var transactions = service.getTransactions(ModelUtils.existingUser(), ModelUtils.
     existingBankAccount(), 10, 0);
    Assertions.assertEquals(2, transactions.size());
}
```

7. **Create Transaction for Invalid User (SqliteTransactionServiceTest)**
   Wenn eine Transaktion für einen nicht existenten Benutzer erzeugt wird, soll ein Fehler geworfen werden.

```
@Test
void testCreateTransactionInvalidUser()
{
    var service = transactionService();

    Assertions.assertThrows(DatabaseException.class, () -> service.saveTransaction(new
     Transaction(1, 5, 1, 1, BigDecimal.ZERO, Instant.now(), "1234")));
}
```

8. **Database Provider Test (DatabaseProviderTest)**
   Prüfe ob die SqlDatabaseServiceRepository alle Services richtig erzeugt.

```
@Test
void testDatabaseProvider() throws SQLException
{
    DatabaseProvider.init(new SqlDatabaseServiceRepository(requestBackend()));
    Assertions.assertNotNull(DatabaseProvider.getBankAccountService());
    Assertions.assertNotNull(DatabaseProvider.getUserService());
    Assertions.assertNotNull(DatabaseProvider.getTransactionService());
    Assertions.assertNotNull(DatabaseProvider.getCategoryService());
```

```
9
10      Assertions.assertThrows(IllegalStateException.class, () -> DatabaseProvider.init(new
         SqlDatabaseServiceRepository(requestBackend())));
11  }
```

9. **Create and Get Bank Account (SqliteBankAccountServiceTest)**
   Legt einen Bankaccount für einen Benutzer an und prüft ob die Daten richtig gespeichert wurden.

```
1   @Test
2   void testCreateAndGet()
3   {
4       var service = bankAccountService();
5
6       service.createBankAccount(BankAccount.create(ModelUtils.existingUser(), NAME));
7       var account = service.getBankAccount(1).orElseThrow();
8
9       Assertions.assertEquals(NAME, account.getName());
10      Assertions.assertFalse(service.getBankAccountsOfUser(ModelUtils.existingUser(), 1).isEmpty());
11      Assertions.assertFalse(service.getBankAccountOfUserByName(ModelUtils.existingUser(), NAME).
         isEmpty());
12  }
```

10. **Delete Bank Account (SqliteBankAccountServiceTest)**
    Erzeugt ein Konto und löscht es.

```
1   @Test
2   void testDelete()
3   {
4       var service = bankAccountService();
5
6       service.createBankAccount(BankAccount.create(ModelUtils.existingUser(), NAME));
7       service.deleteBankAccount(service.getBankAccount(1).orElseThrow());
8
9       Assertions.assertTrue(service.getBankAccount(1).isEmpty());
10  }
```

5.2 ATRIP: Automatic, Thorough und Professional (2P)

[je Begründung/Erläuterung, wie 'Automatic', 'Thorough' und 'Professional' realisiert wurde – bei 'Thorough' zusätzlich Analyse und Bewertung zur Testabdeckung]

- **Automatic**
  Als Framework für die automatisierten Tests wird JUnit verwendet. Das Maven Surefire Plugin führt die Softwaretests direkt beim Kompilierungsprozess aus.

- **Thorough**
  Es werden Unit und Integrationstests verwendet. Da als Datenbank SQLite eingesetzt wird und die Datenbank dadurch lokal läuft, wird im Code selbst nicht zwischen Unit und Integrationstests unterschieden.

  Die Testabdeckung liegt derzeit bei lediglich 22% und ist somit nicht besonders hoch. Auffällig ist dabei, dass der Datenbank-Layer mit einer Abdeckung von über 95% deutlich besser getestet ist als andere Komponenten. Derzeit werden nur einzelne Befehle automatisiert getestet, was vor allem dem hohen Implementierungsaufwand geschuldet ist. Dieser entsteht insbesondere dadurch, dass für die Tests Datenbank-Services gemockt werden müssen.

- **Professional**
  Unsere automatisierten Softwaretests sind professionell, da diese einer klaren Struktur folgen: Die zu einer Klasse gehörenden Tests befinden sich immer in der entsprechenden Datei Namens *[domain.package.Example]Test*. Die Tests sind sinnvoll benannt und testen immer nur einen Pfad einer Funktionalität. Der Testcode folgt den allgemeinen Regeln zu Clean Code.

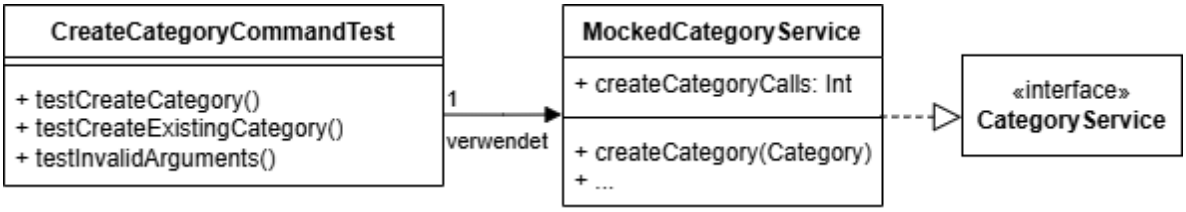
5.3 Fakes und Mocks (4P)

[Analyse und Begründung des Einsatzes von 2 Fake/Mock-Objekten (die Fake/Mocks sind ohne Drittherstellter-Bibliothek/Framework zu implementieren); zusätzlich jeweils UML Diagramm mit Beziehungen zwischen Mock, zu mockender Klasse und Aufrufer des Mocks]

CategoryService Mock

```
@Test
void testCreateCategory()
{
  var mockedService = new MockedCategoryService();
  var command = new CreateCategoryCommand(mockedService, ModelUtils.existingUser());
  command.execute(new String[]{"-n", "test", "-b", "10"});
  Assertions.assertEquals(1, mockedService.getCreatedCategories());
}
```

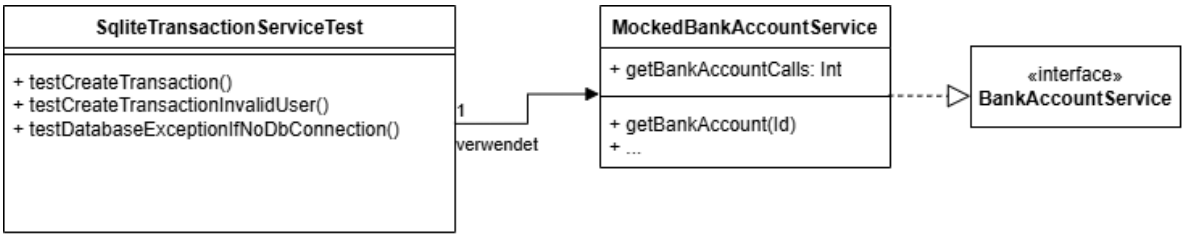
In diesem Unit Test wird der Create Category Command getestet. Um nicht die Datenbanklogik testen zu müssen wird ein Mock Objekt eingesetzt. Der MockedCategoryService ist eine Implementierung des CategoryService Interface.



BankAccountService Mock

```java
@Test
void testCreateTransaction()
{
  var service = transactionService();
  service.saveTransaction(new Transaction(1, 1, 1, 1, BigDecimal.ZERO, Instant.now(), "1234"));
  service.saveTransaction(new Transaction(1, 1, 1, -1, BigDecimal.ZERO, Instant.now(), "1234"));

  var transactions = service.getTransactions(ModelUtils.existingUser(), ModelUtils.existingBankAccount()
     , 10, 0);
  Assertions.assertEquals(2, transactions.size());
}

private SqliteTransactionService transactionService()
{
  try {
    var backend = requestBackend();
    return new SqliteTransactionService(backend, new MockedBankAccountService());
  } catch (SQLException ex) {
    throw new RuntimeException(ex);
  }
}
```

In den Tests der Klasse *SqliteTransactionServiceTest* wird der TransactionService getestet. Dieser benötigt als Abhängigkeit einen BankAccount Service. Da dieser nicht mitgetestet werden soll wird der BankAccount Service gemockt.



6 Domain Driven Design (8P)

6.1 Ubiquitous Language (2P)

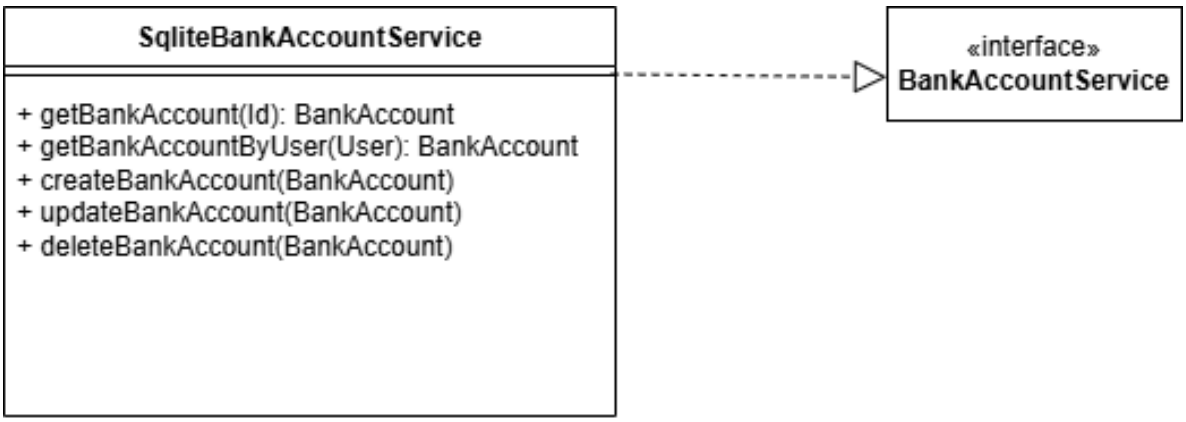
[4 Beispiele für die Ubiquitous Language; jeweils Bezeichung, Bedeutung und kurze Begründung, warum es zur Ubiquitous Language gehört]

Bezeichung	Bedeutung	Begründung
Bank Account	Ein Finanzkonto, das von einem Kunden genutzt wird, um Geld zu speichern und zu verwalten.	Ein "Bank Accountist die zentrale Entität, um Guthaben, Einzahlungen und Abhebungen zu verfolgen. Es ist ein grundlegendes Konzept in Bilanzius.
Transaction	Eine finanzielle Aktivität, bei dem Geld zwischen Konten bewegt wird, dazu gehören Einzahlungen und Abhebungen.	Transaktionen stellen die grundlegenden Operationen dar, die den Zustand eines Kontos verändern.
Balance	Der Betrag an Geld, der aktuell auf einem Bankkonto verfügbar ist.	Die "Balance"gibt den aktuellen Stand eines Kontos an und ist entscheidend, um zu bestimmen, ob weitere Transaktionen durchgeführt werden können.
Report	Ein Dokument, das die Transaktionen eines Bankkontos über einen bestimmten Zeitraum zusammenfasst.	Der Report"gibt dem Kunden eine detaillierte Übersicht über alle Aktivitäten auf seinem Konto, einschließlich Einzahlungen und Abhebungen. Es hilft bei der Überwachung der Finanzen.

6.2 Repositories (1,5P)

[UML, Beschreibung und Begründung des Einsatzes eines Repositories; falls kein Repository vorhanden: ausführliche Begründung, warum es keines geben kann/hier nicht sinnvoll ist – NICHT, warum es nicht implementiert wurde]

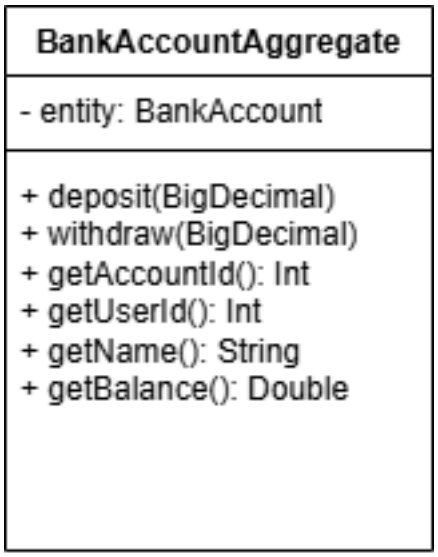
Der BankAccountService ist ein, obwohl er ungünstig benannt wurde, Repository. Dieses ermöglicht den Zugriff auf die BankAccount Datenschicht. Es stellt einfache CRUD Operationen bereit.



6.3 Aggregates (1,5P)

[UML, Beschreibung und Begründung des Einsatzes eines Aggregates; falls kein Aggregate vorhanden: ausführliche Begründung, warum es keines geben kann/hier nicht sinnvoll ist– NICHT, warum es nicht implementiert wurde]

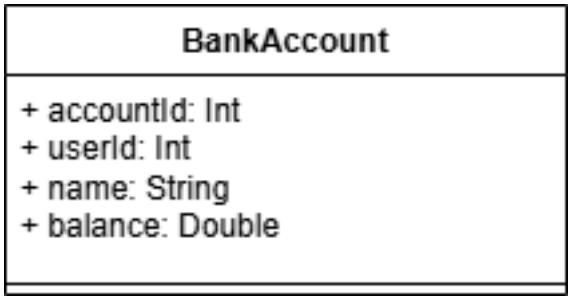
Ein Aggregate ist eine Gruppe von eng zusammengehörigen Domain-Objekten, die im System als eine logische Einheit betrachtet und behandelt wird. Im Zentrum steht der sogenannte Aggregate Root, der als einziger Zugangspunkt zu den enthaltenen Objekten dient. Das BankAccountAggregate erlaubt den Zugriff auf ein BankAccount Entity. Es kapselt die *setBalance* Funktion und ermöglicht so eine bessere Integritätsprüfung.



6.4 Entities (1,5P)

[UML, Beschreibung und Begründung des Einsatzes einer Entity; falls keine Entity vorhanden: ausführliche Begründung, warum es keine geben kann/hier nicht sinnvoll ist– NICHT, warum es nicht implementiert wurde]

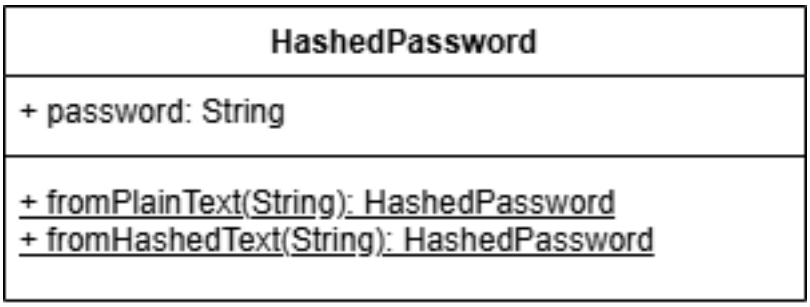
Das Entity BankAccount repräsentiert ein Java-Objekt, das der Datenbanktabelle BankAccount entspricht. Es ermöglicht einen unkomplizierten und direkten Zugriff auf die darin gespeicherten Daten aus dem Java-Code heraus.



6.5 Value Objects (1,5P)

[UML, Beschreibung und Begründung des Einsatzes eines Value Objects; falls kein Value Object vorhanden: ausführliche Begründung, warum es keines geben kann/hier nicht sinnvoll ist– NICHT, warum es nicht implementiert wurde]

Die Klasse *HashedPassword* ist als Value Object für Passwörter konzipiert. Sie macht deutlich, dass das darin enthaltene Passwort ausschließlich in gehashter Form vorliegt und nicht im Klartext gespeichert sind. Dadurch soll sichergestellt werden, dass es während der Entwicklung nicht zu Missverständnissen im Umgang mit Passwörtern kommt.



7 Refactoring (8P)

7.1 Code Smells (2P)

[jeweils 1 Code-Beispiel zu 2 unterschiedlichen Code Smells (die benannt werden müssen) aus der Vorlesung; jeweils Code-Beispiel und einen möglichen Lösungsweg bzw. den genommen Lösungsweg beschreiben (inkl. (Pseudo-)Code)]

Code Smell 1: Duplicated Code *main/Commit bc6ad15*

Sollte eigentlich OK sein, aber ich bin kein Fän davon, dass wir hier das gleiche Beispiel von DRY nutzen

Vorher

```
1  private String convertToBitcoin()
2  {
3
4      JsonObject jsonObject = Requests.getRequest(currencyUrl);
5      assert jsonObject != null;
6      BigDecimal exchangeRate = BigDecimal.valueOf(getCurrencyFromJson(jsonObject, "btc"));
7      BigDecimal balance;
8
9      try {
10         balance = bankAccountService.getBankAccount(selectedBankAccount.getAccountId()).orElseThrow().
       getBalance();
11     } catch (DatabaseException e) {
12         return localization.getMessage("database_error", e.toString());
13     }
14
15     return localization.getMessage("convert_balance", "Bitcoin", (balance.multiply(exchangeRate)));
16 }
17
18 private String convertToGermanDeutscheMark()
19 {
20
21     JsonObject jsonObject = Requests.getRequest(currencyUrl);
22     assert jsonObject != null;
23     BigDecimal exchangeRate = BigDecimal.valueOf(getCurrencyFromJson(jsonObject, "dem"));
24     BigDecimal balance;
25        try {
26         balance = bankAccountService.getBankAccount(selectedBankAccount.getAccountId()).orElseThrow().
       getBalance();
27     } catch (DatabaseException e) {
28         return localization.getMessage("database_error", e.toString());
29     }
30
31     return localization.getMessage("convert_balance", "Deutsche Mark", (balance.multiply(exchangeRate)))
       ;
32 }
33
34 private String convertToSwissFranc()
35 {
36
37     String currency = "Swiss Franc";
38     if (localization.getCurrentLanguageCode().equals("de")) {
39         currency = "Schweizer Franken";
40     }
41
42     JsonObject jsonObject = Requests.getRequest(currencyUrl);
43     assert jsonObject != null;
44     BigDecimal exchangeRate = BigDecimal.valueOf(getCurrencyFromJson(jsonObject, "chf"));
45     BigDecimal balance;
46
47     try {
48         balance = bankAccountService.getBankAccount(selectedBankAccount.getAccountId()).orElseThrow().
       getBalance();
49     } catch (DatabaseException e) {
```

```
50          return localization.getMessage("database_error", e.toString());
51      }
52
53      return localization.getMessage("convert_balance", currency, (balance.multiply(exchangeRate)));
54  }
55
56  private String convertToDogecoin()
57  {
58
59      JsonObject jsonObject = Requests.getRequest(currencyUrl);
60      assert jsonObject != null;
61      BigDecimal exchangeRate = BigDecimal.valueOf(getCurrencyFromJson(jsonObject, "doge"));
62      BigDecimal balance;
63
64      try {
65          balance = bankAccountService.getBankAccount(selectedBankAccount.getAccountId()).orElseThrow().
        getBalance();
66      } catch (DatabaseException e) {
67          return localization.getMessage("database_error", e.toString());
68      }
69
70      return localization.getMessage("convert_balance", "Deutsche Mark", (balance.multiply(exchangeRate)))
        ;
71  }
72
73  private String convertToSwissFranc()
74  {
75
76      String currency = "Swiss Franc";
77      if (localization.getCurrentLanguageCode().equals("de")) {
78          currency = "Schweizer Franken";
79      }
80
81      JsonObject jsonObject = Requests.getRequest(currencyUrl);
82      assert jsonObject != null;
83      BigDecimal exchangeRate = BigDecimal.valueOf(getCurrencyFromJson(jsonObject, "chf"));
84      BigDecimal balance;
85
86      try {
87          balance = bankAccountService.getBankAccount(selectedBankAccount.getAccountId()).orElseThrow().
        getBalance();
88      } catch (DatabaseException e) {
89          return localization.getMessage("database_error", e.toString());
90      }
91
92      return localization.getMessage("convert_balance", currency, (balance.multiply(exchangeRate)));
93  }
94
95  private String convertToDogecoin()
96  {
97
98      JsonObject jsonObject = Requests.getRequest(currencyUrl);
99      assert jsonObject != null;
100     BigDecimal exchangeRate = BigDecimal.valueOf(getCurrencyFromJson(jsonObject, "doge"));
101     BigDecimal balance;
102
103     try {
104         balance = bankAccountService.getBankAccount(selectedBankAccount.getAccountId()).orElseThrow().
        getBalance();
105     } catch (DatabaseException e) {
106         return localization.getMessage("database_error", e.toString());
107     }
108 }
```

Lösung

Der Code Smell wurde gelöst, indem die Methode *convertCurrency(String currencyCode, String currencyName)* eingeführt wurde. Diese wird in den einzelnen Methoden für die jeweilige Währung aufgerufen.

```
1  private String convertCurrency(String currencyCode, String currencyName) {
2      JsonObject jsonObject = Requests.getRequest(currencyUrl);
```

```
        assert jsonObject != null;
        BigDecimal exchangeRate = BigDecimal.valueOf(getCurrencyFromJson(jsonObject, currencyCode));
        BigDecimal balance;

        try {
            balance = bankAccountService.getBankAccount(selectedBankAccount.getAccountId()).orElseThrow().
        getBalance();
        } catch (DatabaseException e) {
            return localization.getMessage("database_error", e.toString());
        }

        return localization.getMessage("convert_balance", currencyName, (balance.multiply(exchangeRate)));
    }

    private String convertToBitcoin()
    {
        return convertCurrency("btc", "Bitcoin");
    }

    private String convertToGermanDeutscheMark()
    {
        return convertCurrency("dem", "Deutsche Mark");
    }

    private String convertToSwissFranc()
    {
        String currency = localization.getCurrentLanguageCode().equals("de") ? "Schweizer Franken" : "Swiss
        Franc";
        return convertCurrency("chf", currency);
    }

    private String convertToDogecoin()
    {
        return convertCurrency("doge", "Dogecoin");
    }

    private String convertToEthereum()
    {
        return convertCurrency("eth", "Ethereum");
    }

    private String convertToHongKongDollar()
    {
        return convertCurrency("hkd", "Hong Kong Dollar");
    }

    private String convertToJamaicanDollar()
    {
        String currency = localization.getCurrentLanguageCode().equals("de") ? "Jamaikanische Dollar" : "
        Jamaican Dollar";
        return convertCurrency("jmd", currency);
    }

    private String convertToNorthKoreanWon()
    {
        String currency = localization.getCurrentLanguageCode().equals("de") ? "Nordkoreanische Won" : "
        North Korean Won";
        return convertCurrency("kpw", currency);
    }

    private String convertToRussianRuble()
    {
        String currency = localization.getCurrentLanguageCode().equals("de") ? "Russischer Rubel" : "Russian
          Ruble";
        return convertCurrency("rub", currency);
    }

    private String convertToUsDollar()
    {
        return convertCurrency("usd", "US Dollar");
    }
```

Code Smell 2: Switch-Statement *main/Commit c46c962*

Vorher

```
public CommandController()
{
}

public String handleInput(String input)
{

    String output;

    switch (input) {
        case "/exit":
            System.exit(0);
            output = "Goodbye User";
            break;
        case "/help":
            HelpCommandService helpCommandService = new HelpCommandService();
            output = helpCommandService.getAllCommands();
            break;
        default:
            output = "Something went wrong :( ";
            break;
    }

    return output;
}
```

Lösung

Dadurch das bei weiteren Commands das Switch-Statement komplexer geworden wäre, wurde es durch eine Hash-Map ersetzt für mehr flexibilität. Dazu wurde nun auch der Code für die Commands in eigene Klassen ausgelagert. Das hat generll auch zu einer verbesserten lesbarkeit gewirkt.

```
// CommandController.java
private final Map<Commands, CommandService> commandMap;

public CommandController()
{

    commandMap = new HashMap<>();

    commandMap.put(Commands.EXIT, new ExitCommandService());
    commandMap.put(Commands.HELP, new HelpCommandService());
    commandMap.put(Commands.BILANZIUS, new BilanziusCommandService());

}

public String handleInput(String input)
{

    String[] parts = input.split(" ", 2);
    String commandStr = parts[0];
    String[] arguments = parts.length > 1 ? parts[1].split(" ") : new String[0];

    Commands command = Commands.fromString(commandStr);
    CommandService commandService = commandMap.get(command);

    if (commandService != null) {
        return commandService.execute(arguments);
    }

    return "Unknown command :( . Type /help for a list of commands.";
}

// Commands.java
public enum Commands
{
    EXIT("/exit", "Exit the application", null),
```

```
36      HELP("/help", "Show all commands", null),
37      BILANZIUS("/bilanzius", "Get information about the application", BilanziusCommandArguments.
         getAllArguments());
38
39      // Hier werden die einzelnen Befehle hinzugefügt
40
41      private final String command;
42      private final String description;
43      private final String arguments;
44
45      Commands(String command, String description, String arguments) {
46          this.command = command;
47          this.description = description;
48          this.arguments = arguments;
49      }
50
51      public String getCommand() {
52          return command;
53      }
54
55      public String getDescription() {
56          return description;
57      }
58
59      public String getArguments() {
60          return arguments;
61      }
62
63      public static String getAllCommands() {
64
65          return Arrays.stream(Commands.values()).map(
66                  c -> c.getCommand()
67                          + " - " +
68                          c.getDescription()
69                          +
70                          (
71                                  c.getArguments()
72                                          != null ?
73                                          " | " + (c.getArguments())
74                                          : ""
75                          )
76                  ).reduce(
77                          (a, b) -> a + "\n" + b
78                  ).orElse("");
79      }
80
81      public static Commands fromString(String command) {
82          for (Commands c : Commands.values()) {
83              if (c.command.equals(command)) {
84                  return c;
85              }
86          }
87          return null;
88      }
89  }
90
91  public interface CommandService
92  {
93      String execute(String[] arguments);
94  }
```

7.2 2 Refactorings (6P)

[2 unterschiedliche Refactorings aus der Vorlesung jeweils benennen, anwenden, begründen, sowie UML vorher/nachher liefern; jeweils auf die Commits verweisen – die Refactorings dürfen sich nicht mit den Beispielen der Code überschneiden]

Refactoring 1: Replace Error Code with Exceptions *main/Commit 2cc1fce*

Alle Methoden von UserService werfen eine DatabaseException die nicht abgefangen wurde. Daher wurden try-catch-Block um die jeweiligen Methoden hinzugefügt.

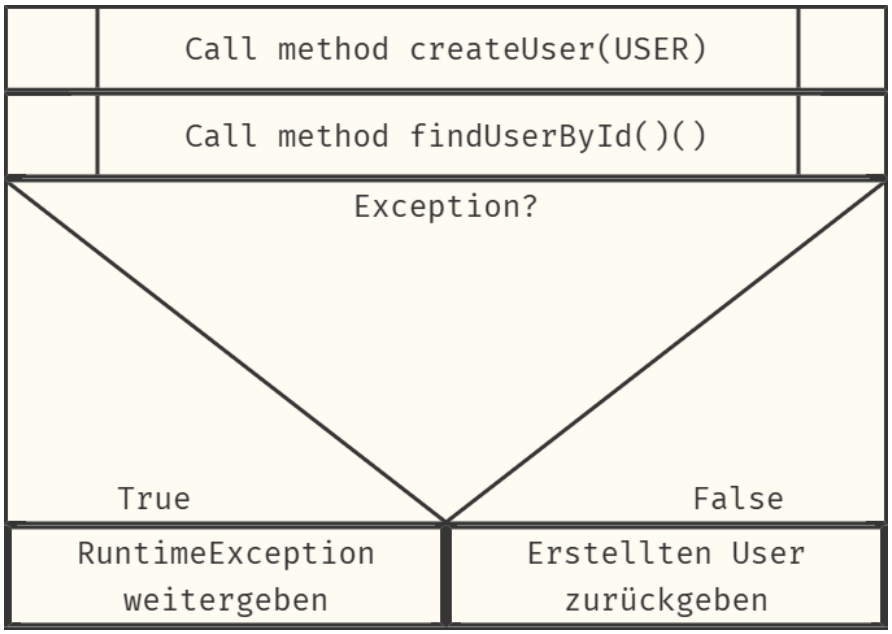


Abbildung 1: SignUp#register vorher

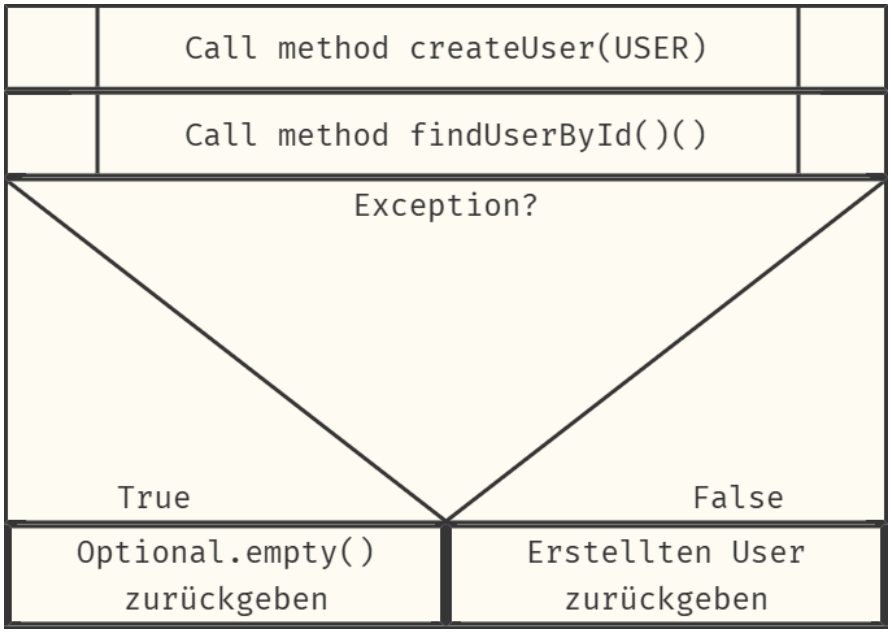
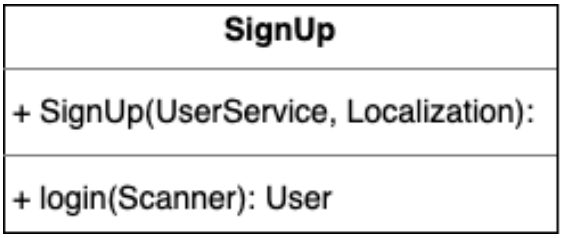
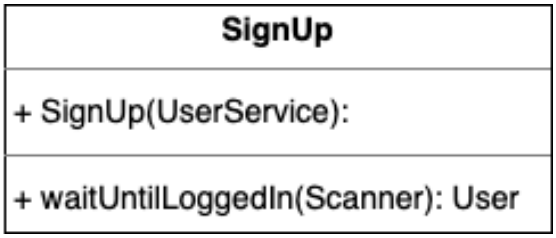


Abbildung 2: SignUp#register nachher

Refactoring 2: Rename Methods *main/Commit d3be454*

Die Methode login() in der Klasse SignUp wurde zu waitUntilLoggedIn() umbenannt da in der Methode erwartet wird bis der Benutzter eingeloggt ist und daraufhin dann entscheidet.





8 Entwurfsmuster (8P)

[2 unterschiedliche Entwurfsmuster aus der Vorlesung (oder nach Absprache auch andere) jeweils benennen, sinnvoll einsetzen, begründen und UML-Diagramm]

8.1 Entwurfsmuster : Singleton (Erzeugungsmuster) (4P)

Einsatz im Projekt: `Localization`-Klasse (`org.bilanzius.utils.Localization`)

```
1  private static Localization instance;
2
3  public static Localization getInstance()
4  {
5      if (instance == null) {
6          instance = new Localization("en"); // Standard auf Englisch
7      }
8      return instance;
9  }
```

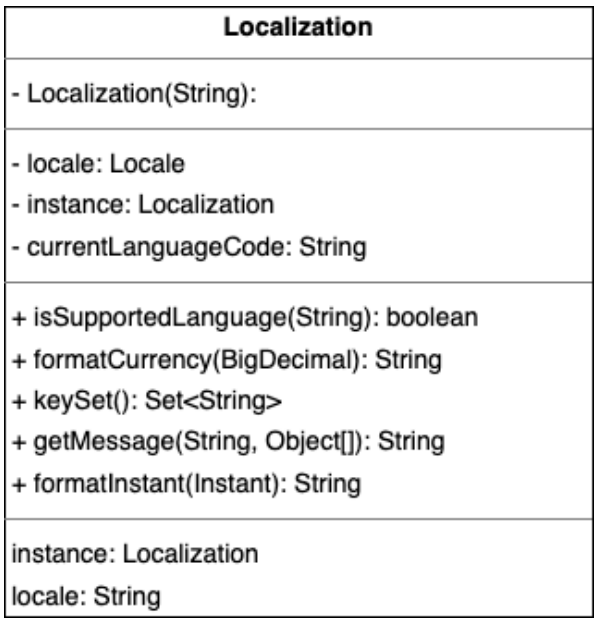
Sinnvoller Einsatz

Die `Localization`-Klasse wird einmalig instanziiert und stellt eine globale Instanz bereit, die für das Abrufen von sprachabhängigen Texten zuständig ist. So kann das gesamte System konsistent auf Sprachdaten zugreifen, ohne dass mehrere Objekte unterschiedliche Zustände haben.

Begründung

1. Nur eine Instanz notwendig für zentrale Konfiguration
2. Globale Zugänglichkeit
3. Minimierung von Speicherverbrauch und Inkonsistenzen

UML-Diagramm



8.2 Entwurfsmuster: Factory/Builder (Erzeugungsmuster) (4P)

Bezeichnung: Factory / Builder (statische Factorymethode)
Einsatz im Projekt: Klassen wie `BankAccount`, `Transaction`, `User` (z. B. `BankAccount.create()`)

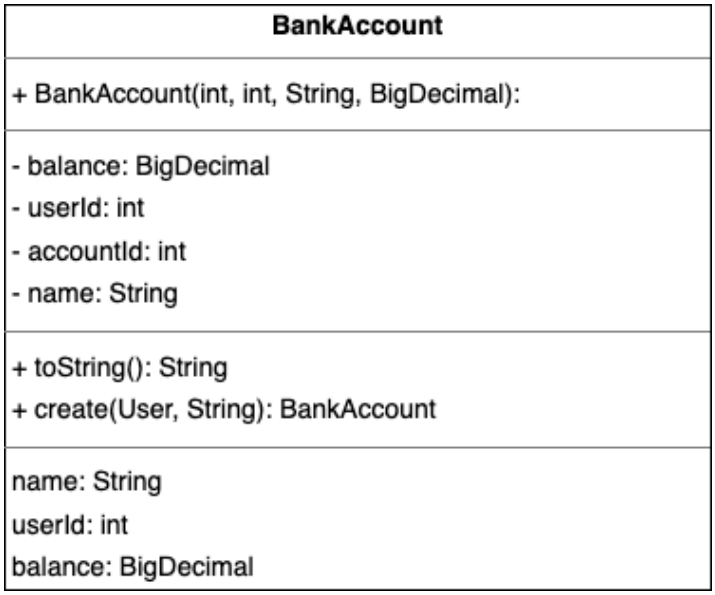
Sinnvoller Einsatz

Objekte wie Bankkonten oder Transaktionen sollen immer im konsistenten Zustand erstellt werden. Statt viele verschiedene Konstruktoren zu nutzen, wird eine kontrollierte `create()`-Methode verwendet. Dadurch wird die Instanziierung vereinfacht und typische Fehler bei der Objekterstellung werden vermieden.

Codebeispiel: Statische Factory-Methode `create()`

```java
package org.bilanzius.persistence.models;

import java.math.BigDecimal;

public class BankAccount {

    private long accountId;
    private long userId;
    private String name;
    private BigDecimal balance;

    private BankAccount(long accountId, long userId, String name, BigDecimal balance) {
        this.accountId = accountId;
        this.userId = userId;
        this.name = name;
        this.balance = balance;
    }

    public static BankAccount create(User user, String name) {
        return new BankAccount(0, user.getId(), name, new BigDecimal("0.0"));
    }

    //Getter  Setter
    //...
}
```

UML-Diagramm



Begründung

- Einheitliche Instanziierung über statische Fabrikmethoden
- Vermeidung fehleranfälliger Konstruktoraufrufe mit falschen Parametern
- Möglichkeit zur Vorbelegung von Standardwerten (z. B. `balance = 0`)
- Erhöhte Lesbarkeit und Wartbarkeit