

Technology **Arts Sciences**

TH Köln

SWP Vorgaben
Softwarepraktikum

TH Köln - Informatik Labor

Version 10.2, 2024-02-14 10:01:46 UTC

Inhaltsverzeichnis

1. Zweck des Dokuments	1
2. Vorgaben zu den Werkzeugen	2
3. Verwendung von Gitlab Issue Tracker	3
4. Verwendung von Git	4
5. Allgemeine Hinweise	5
5.1. Schichten-Architektur und Kopplung	5
5.2. IntelliJ-Projekte und Maven-Repository des SWP	5
5.3. Paketnamen und Modulnamen	5
5.4. Stringvergleich	6
5.5. Stil des Java-Codes	6
5.6. Konsolenausgaben	6
5.7. Versionsangaben in pom.xml	6
5.8. Qualität des Codes	6
5.9. Verwendung der Gateway-Klasse für EntityManager	7
5.10. Vorgegebene Komponenten-Schnittstellen	7
5.11. Verwendung der Schnittstellen der Datenhaltung	7
6. Hinweise zu MS2	9
6.1. Datenbank-Zugriffe	9
6.2. JUnit-Testfälle	9
7. Vorgaben zur Arbeit nach MS2	11
8. Hinweise zu MS3	12
8.1. Klassendiagramm	12
8.1.1. Gestaltung des Klassendiagramms	12
8.1.2. Verwendung einer geschlossenen 3-Schichten Architektur	13
8.2. Anwendungsfall-Tabellen	14
8.3. Konzeptueller Entwurf der GUI	14
8.4. GUI der Anwendungsfälle	14
8.4.1. Robustheit	14
8.4.2. Rückmeldung an den Benutzer	15
8.4.3. Komfort bei der Bedienung	15
8.4.4. Beschriftungen der GUI-Elemente	15
8.4.5. Anordnung der GUI-Elemente	15
9. Hinweise zu MS4	16
9.1. Entwicklerdokumentation	16
10. Hinweise zu MS5	17

1. Zweck des Dokuments

Dieses Dokument erläutert die verbindlichen Vorgaben im Software-Praktikum. Die korrekte Umsetzung dieser Vorgaben geht in die Note zum Software-Praktikum ein. Die Vorgaben sollen eine qualitäts-orientierte Umsetzung der System-Spezifikation und des Grobentwurfs sicherstellen.

2. Vorgaben zu den Werkzeugen

Es sollen die folgenden Versionen der Werkzeuge verwendet werden:

- IntelliJ IDEA Community Edition in Version 2023.x (Download unter <https://www.jetbrains.com>)
- JDK in Version 21, wir verwenden und empfehlen das Adoptium OpenJDK (<https://adoptium.net/>)
- JavaFX für die Erstellung der Benutzeroberflächen.
- JUnit in Version 4

Zur Erstellung von UML-Diagrammen empfehlen wir:

- PlantUML: (<https://www.plantuml.com>) erstellt UML Diagramme aus Textdateien
- ein gutes freies Programm (online und Desktop-Version): draw.io (<https://www.draw.io>)
- sehr einfaches Werkzeug: violetUML in Version 2.0.1 (siehe <https://violet.sourceforge.net>)
- MS Visio

3. Verwendung von Gitlab Issue Tracker

Kommentare zu den Meilensteinen MS2, MS3, und MS4 und weitere Kommentare werden von uns als Issues im Gitlab Issue Tracker kommuniziert. Die Teams müssen diese Fehler bis zum nächsten Meilenstein korrigieren, d.h.

- alle Fehler, die vor dem MS3-Termin dokumentiert wurden, müssen bis MS3 behoben werden (also insbesondere die Fehler der Abgaben zu MS2).
- alle Fehler, die nach dem dem MS3-Termin und vor dem MS4-Termin dokumentiert wurden, müssen bis MS4 behoben werden.
- alle nach dem dem MS4-Termin dokumentierten Fehler müssen zum MS5-Termin behoben sein.
- nach jeder Fehlerkorrektur muss ein neuer Push in den Master-Branch des Git Remote Repository erfolgen.

Bei der Korrektur soll die Phase der Fehlerbehebung durch den Status des Issues in Gitlab dokumentiert werden. Der Lebenszyklus eines Issues in Gitlab soll im Software-Praktikum folgendermaßen verlaufen:

- Ein neuer Issue erhält von uns den Status **open**.
- Handelt es sich bei diesem Issue um
 - einen Fehler, so wird er mit dem Label **bug** gekennzeichnet,
 - einen Vorschlag oder Hinweis (der keinen Fehler darstellt), so wird er mit dem Label **suggestion** gekennzeichnet. Diese Vorschläge müssen nicht wie Fehler zwingend umgesetzt bzw. korrigiert werden, wir empfehlen es aber.
- Wenn ein Fehler korrigiert wurde, so soll derjenige, der den Fehler korrigiert hat, das Issue in den Status **closed** setzen.
- Falls das Team feststellt, dass ein eingetragener Fehler kein wirklicher Fehler ist oder es sich um eine Duplette handelt, so soll dieser Fehler durch das Team mit dem Label **nobug** gekennzeichnet werden. Zusätzlich soll eine Erklärung hierzu bei dem Issue eingetragen werden.
- Wir kontrollieren regelmäßig den Zustand der Fehler:
 - Falls wir mit der Bearbeitung eines Fehlers im Status **closed** zufrieden sind, dann setzen wir den Fehler auf den Status **approved**. Damit ist die Bearbeitung des Fehlers für das Team erfolgreich abgeschlossen.
 - Falls wir nicht zufrieden sind mit der Lösung des Fehlers, dann kennzeichnen wir den Fehler mit dem Label **reopened** und das Team muss den Fehler erneut bearbeiten und nach Bearbeitung in den Status **closed** setzen.

4. Verwendung von Git

Gitlab wird zur Versionsverwaltung im Software-Praktikum verwendet.

In Gitlab erfolgt auch die Abgabe zu den Meilensteinen MS2, MS3, MS4, MS5. Zu jeder Abgabe (außer MS2) sollen Sie einen entsprechenden Tag setzen: MS3, MS4, MS5. Diese Tags sollen im Git Remote Repository, Branch master erstellt werden, nicht in einem anderen Branch.

Ihre eigenen Branches dürfen nicht die Bezeichnungen MS2, MS3, MS4 oder MS5 besitzen!

Bei Fragen zu Git oder Problemen mit Git, können sich die Teilnehmer an die Tutoren oder die wissenschaftlichen Mitarbeiter wenden.

5. Allgemeine Hinweise

5.1. Schichten-Architektur und Kopplung

Das zu erstellende System im Software-Praktikum besitzt eine geschlossene Schichten-Architektur mit 3 Schichten: Datenhaltung, Fachlogik, Benutzeroberfläche.

Zwischen den Schichten Datenhaltung und Fachlogik muss eine lose Kopplung realisiert werden, d.h.

- Pakete mit den abstrakten Schnittstellen-Klassen der Datenhaltung sollen exportiert und von den Modulen der Fachlogik importiert werden.
- Pakete mit den konkreten Implementierungsklassen der Schnittstellen der Datenhaltung dürfen nicht exportiert werden.
- Die Module der Fachlogik müssen die Implementierungen der Schnittstellen der Datenhaltung mit Hilfe der Klasse `ServiceLoader` ermitteln.
- Es dürfen die Pakete eines Moduls nicht mit `opens` und es darf das Modul nicht mit `open` für Modul-externe Zugriffe geöffnet werden.

Zwischen den Schichten Fachlogik und Benutzeroberfläche soll eine enge Kopplung realisiert werden, d.h.

- Pakete mit den abstrakten Schnittstellen-Klassen der Fachlogik sollen exportiert und von den Modulen der Benutzeroberfläche importiert werden.
- Pakete mit den konkreten Implementierungsklassen der Schnittstellen (also die Steuerungsklassen) der Fachlogik sollen exportiert und von den Modulen der Benutzeroberfläche importiert werden.
- Es dürfen die Pakete eines Moduls nicht mit `opens` und es darf das Modul nicht mit `open` für Modul-externe Zugriffe geöffnet werden.

5.2. IntelliJ-Projekte und Maven-Repository des SWP

Jede Komponente eines Systems (Hinweis: Alle Komponenten eines Systems sind im Grobentwurf definiert.) ist als ein separates IntelliJ-Projekt realisiert. Der Name dieses IntelliJ-Projekts in Gitlab entspricht dem Namen der Komponente (alle Buchstaben sind klein geschrieben).

5.3. Paketnamen und Modulnamen

Bei der Realisierung von Modulen organisieren Sie den Code intern in Paketen, genau wie bei jedem anderen Java-Projekt. Hierbei sollen Sie folgende allgemeine

Namenskonvention beachten: Ein Modulname beginnt mit dem umgekehrten Internet-Domänenamen des Unternehmens oder der Organisation. In unserem Fall ist dieses `de.thkoeln`. Dieser Name wird erweitert um Angaben zur zugehörigen Komponente aus dem Grobentwurf (Bsp.: `swp.bks.admindaten` bzw. `swp.wawi.admindaten`). Ein derart gebildeter Modulname definiert auch die Namen der in ihm vorhandenen Pakete: Alle Paketnamen besitzen den Namen des Moduls als Präfix, d.h. in einem Modul mit dem Namen `de.thkoeln.swp.bks.admindaten` beginnen alle Pakete mit `de.thkoeln.swp.bks.admindaten` und erweitern diesen gegebenenfalls um weitere Ebenen. Weil alle Modulnamen in einem System eindeutig sein müssen, sind somit auch alle Paketnamen eindeutig. Weiterhin kann für jedes Paket sehr leicht ermittelt werden, in welchem Modul es sich befindet.

5.4. Stringvergleich

Der Vergleich zweier Strings auf lexikalische Gleichheit in Java muss im Software-Praktikum durch die Anwendung der Methode `equals()` erfolgen.

Ein Vergleich von Strings auf lexikalische Gleichheit mit `==` ist nicht erlaubt.

5.5. Stil des Java-Codes

Der Stil des Java-Codes im Software-Praktikum muss den Hinweisen aus dem Modul Software Engineering folgen.

5.6. Konsolenausgaben

Das Programm darf grundsätzlich keine Konsolenausgaben machen, die nicht ohne eine Neu-Übersetzung des Programms zu entfernen sind. Hiermit sind explizit `System.out.println()` und andere `print()`-Methoden gemeint, die ihre Meldungen auf die Standardausgabe ausgeben. Eine Ausgabe über einen Logger, der sich auch ohne Neu-Übersetzung des Codes konfigurieren lässt, ist explizit erlaubt.

5.7. Versionsangaben in pom.xml

Die gegebenen Projekte in Git besitzen bereits eine von uns vordefinierte `pom.xml`. Die in dieser `pom.xml` angegebenen Versionen dürfen nicht verändert werden.

5.8. Qualität des Codes

Im Software-Praktikum sollen Sie nicht nur funktionierenden Code erstellen, sondern der erstellte Code soll auch möglichst professionell erstellt werden. Wir achten also bei der Überprüfung des Codes nicht nur darauf, dass der Code funktional korrekt ist, sondern auch darauf, dass er sauber, verständlich, durchdacht, effizient erstellt wurde.

Beispiel: Bei der Implementierung der Methoden der Datenschicht muss der Aufrufer a) den EntityManager setzen und b) oftmals (jedoch nicht immer) ein Objekt als Parameter übergeben. Es soll im Code der Datenschicht unbedingt überprüft werden, ob der EntityManager gesetzt wurde und ob als Parameter übergebene Objekte korrekt instanziert sind. Das heißt, es sind die entsprechenden Überprüfungen im Code vorzunehmen (`em == null` bzw. `objekt == null`).

5.9. Verwendung der Gateway-Klasse für EntityManager

Zur Kapselung der Klasse `EntityManager` der `JPA` ist eine Gateway-Klasse für jedes System von uns bereits implementiert worden:

- BKS: die Klasse `IDatabaseImpl`
- GDS: die Klasse `IDatabaseImpl`
- WAWI: die Klasse `IDatabaseImpl`

Diese Klasse soll von allen Komponenten für den Zugriff auf die Datenbank verwendet werden. Die Verwendung dieser Gateway-Klasse erleichtert den Übergang von der Entwicklungs-Datenbank auf die Produktiv-Datenbank nach MS4 erheblich, da dann nur in dieser Klasse der Name der neuen `PersistenceUnit` geändert werden muss.

Diese Gateway-Klasse verwendet das Entwurfsmuster Singleton, um immer auf den gleichen `EntityManager` zuzugreifen und somit Inkonsistenzen beim Zugriff auf die Datenbank zu vermeiden. Das korrekte Objekt des `EntityManager` erhält man durch den Aufruf der Operation `getEntityManager()`.

Die Interface Klasse `IDatabase` enthält Methoden (`useDevPU`, `useProdPU`) zum Ändern des EntityManagers von dev (:=development) auf prod (:=production) und anders herum. Diese Methoden sollen **nicht** benutzt werden, da diese am Ende während der Integration vom Bootloader (von uns) und somit über alle Komponenten global verwendet werden. Als Default ist während der Entwicklungszeit die `devPU` ausgewählt.

5.10. Vorgegebene Komponenten-Schnittstellen

Die Schnittstellen der Komponenten und die zugehörigen Implementierungsklassen sind in den Dokumenten zum Grobentwurf definiert. Diese vorgegebenen Schnittstellen dürfen nicht verändert werden, d.h. es dürfen keine zusätzlichen Methoden erstellt werden.

5.11. Verwendung der Schnittstellen der Datenhaltung

Die Schnittstellen der Datenhaltung sind in den Dokumenten zum Grobentwurf definiert.

Sie können von anderen Komponenten aus verwendet werden. In der Implementierung der Schnittstellen-Methoden werden oftmals Operationen auf Entitätsobjekten und der Datenbank durchgeführt. Es ist sehr wichtig, dass hierzu der richtige **EntityManager** verwendet wird.

Die Schnittstellen der Datenhaltung sollen in einer bestimmten Art und Weise verwendet werden:

- Die aufrufende Komponente setzt den zu verwendenden **EntityManager** mit der Methode `setEntityManager()`.
- Die aufrufende Komponente ist für die Transaktionsverwaltung zuständig (nicht die aufgerufene Komponenten-Schnittstelle).

Das Sequenz-Diagramm in [Abbildung 1](#) illustriert den vorgegebenen Ablauf zur Verwendung einer Komponenten-Schnittstelle:

Der Klient, d.h. die Operation, die die Schnittstelle verwendet,

1. besorgt sich die korrekte Instanz des **EntityManager** von **IDatabaseImpl** (siehe [Abschnitt 5.9](#)),
2. übergibt diese an die Schnittstelle (`setEntityManager()`),
3. startet die Transaktion (`getTransaction().begin()`),
4. verwendet die gewünschten Operationen der Schnittstelle,
5. beendet die Transaktion (`getTransaction().commit()`).

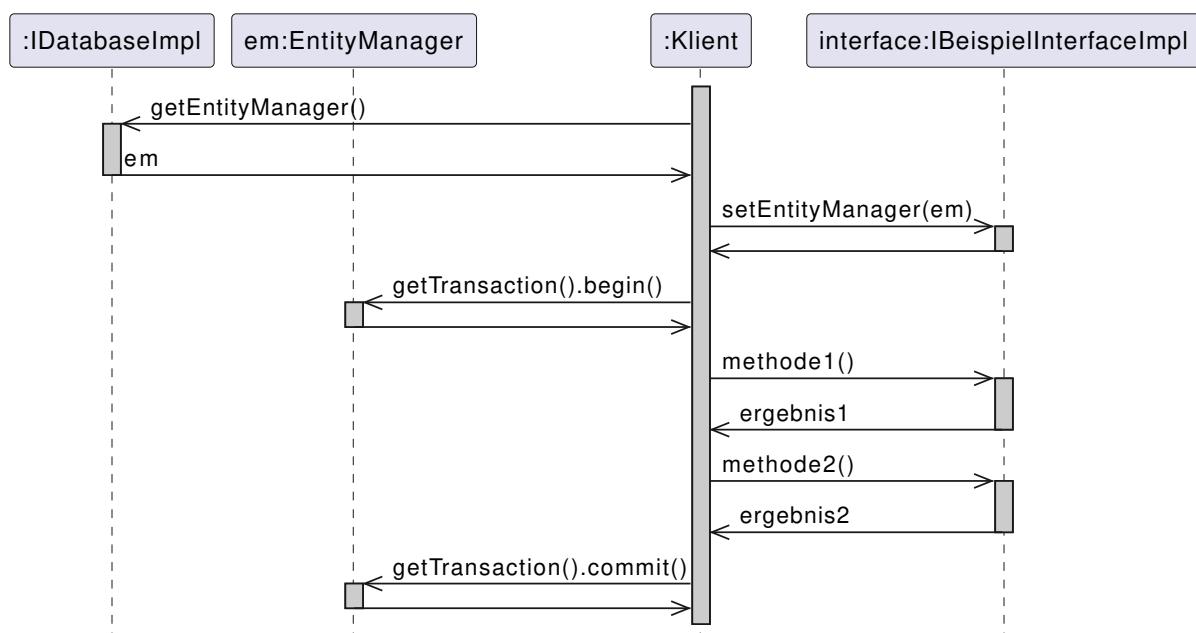


Abbildung 1. Sequenz-Diagramm zur Verwendung einer Komponenten-Schnittstelle

Es ist also immer der Klient, d.h. der Aufrufer einer Operation der Komponenten-Schnittstelle der Datenhaltung, für das Setzen des **EntityManager** und die Transaktionsverwaltung verantwortlich!

6. Hinweise zu MS2

Bis zum MS2 ist zu machen:

- Implementierung der Schnittstellen-Klassen in der Schicht Datenhaltung
 - Das Dokument Arbeitspakete MS2 gibt an, welcher Teilnehmer welche Methoden welcher Schnittstellen-Klasse implementieren soll.
- Implementierung der Testfälle aus der Testspezifikation für die Methoden der implementierten Schnittstellen-Klassen.

Verbindlicher Termin, konkrete Liefergegenstände und genaue Abgabeorte sind in den Folien zur Einführungsveranstaltung angegeben.

6.1. Datenbank-Zugriffe

In den Unterkapiteln des [\[Vorgaben\]](#) finden Sie Informationen für den Zugriff auf die Datenbank unter Verwendung der Gateway-Klasse [IDatabaseImpl](#).

Vor der Nutzung des EntityManagers in einer Implementierung einer Datenhaltungsmethode muss zwingend jedes Mal überprüft werden ob der EntityManager durch die aufrufende Schicht auch korrekt gesetzt wurde.

6.2. JUnit-Testfälle

- Die JUnit-Testfälle müssen entsprechend der Richtlinien zur Pragmatik von JUnit-Testfällen aus dem Modul Software Engineering erstellt werden.
- Die JUnit-Testfälle werden entsprechend der Testspezifikation erstellt. Testspezifikationen existieren für alle Schnittstellen der Komponenten der Schicht "Datenhaltung".
- Ein Testfall darf nicht abhängig von der aktuellen Realisierung der Konstruktoren sein. Falls beispielsweise die Testspezifikation fordert "UND die ID des Testkontos gleich NULL ist", dann muss die ID des für den Testfall erzeugten Objekts explizit auf NULL gesetzt werden. Man darf sich nicht darauf verlassen, dass der Konstruktor dieses übernimmt
 - auch wenn dieses in der aktuellen Implementierung erfolgt.
- Der Text des Testfalls in der Testspezifikation wird zum Java-Kommentar des JUnit-Testfalls, damit man den Zusammenhang leicht erkennen kann. Beispiel:

```
/* @Test: insertKonto_00()
WENN die Methode insertKonto mit einem Testkonto aufgerufen wird,
UND die ID des Testkontos gleich NULL ist,
DANN sollte sie TRUE zurueckliefern,
```

```
UND das Testkonto sollte in der DB existieren.  
*/  
@Test  
public void insertKonto_00()  
...  
...
```

- Der Zustand der Datenbank soll durch die Ausführung der Testfälle nicht persistent verändert werden, d.h. es darf kein `commit` erfolgen, sondern es muss in der `@After`-Methode ein `rollback` vorgenommen werden.
- Die Testfälle dürfen keine Annahmen über den Zustand der Datenbank machen, da alle Entwickler auf der gleichen Datenbank entwickeln und sich somit der Zustand der Datenbank ständig verändert wird. Jeder einzelne Testfall muss vielmehr den gewünschten Zustand der Datenbank selbst herstellen.
- Im Software-Praktikum sollen Sie nicht nur funktionierenden Code erstellen, sondern der erstellte Code soll auch möglichst professionell erstellt werden. Wir achten also bei der Überprüfung des Codes nicht nur darauf, dass der Code funktional korrekt ist, sondern auch darauf, dass er sauber, verständlich, durchdacht, effizient erstellt wurde.

Beispiel: Soll in JUnit beispielsweise geprüft werden, dass eine Variable `result` den Wert `false` besitzt, dann ist folgender Code zwar funktional korrekt, jedoch weder verständlich noch durchdacht: `AssertNotEquals(true, result)`

Verständlich und durchdacht ist dagegen die folgende Überprüfung der Variable `result`: `assertFalse(result)`

- Die Testfälle einer kompletten Komponente sollten insgesamt in einer Zeit von ca. 30 sek maximal durchlaufen. Laufen die Tests länger wurden sie sehr ineffizient programmiert.

7. Vorgaben zur Arbeit nach MS2

Die Datenhaltung sollte eigentlich zu MS2 korrekt implementiert sein. Leider enthält die Datenhaltung aber nach MS2 immer noch Fehler, die zumeist bei der Implementierung der Anwendungsfälle entdeckt werden.

Wird bei der Implementierung der Anwendungsfälle ein Fehler in der Datenhaltung entdeckt, so muss für diesen Fehler im Gitlab Issue Tracker ein Bug erzeugt werden.

Der verantwortliche Implementierer der Datenhaltungs-Methode, die dieser Fehler betrifft, muss diesen Fehler innerhalb einer Woche beheben - oder als unbegründet kennzeichnen. Findet durch den verantwortlichen Implementierer keine fristgerechte Bearbeitung des Fehlers statt, so muss sich der Erzeuger des Bugs im Gitlab Issue Tracker bei den Organisatoren des SWP melden (Assis oder Prof).

Es sollten also alle Teams rechtzeitig vor MS4 mit der Implementierung der Anwendungsfälle beginnen, damit Sie Fehler in den verwendeten Datenhaltungs-Methoden rechtzeitig entdecken. Nur dadurch können Sie sicherstellen, dass Ihre Abgabe zu MS4 auch wirklich lauffähig ist. Ist die Abgabe zu MS4 aufgrund zu spät entdeckter Fehler in den verwendeten Datenhaltungs-Methoden nicht lauffähig, so liegt die Schuld hierfür bei den Implementierern der Anwendungsfälle.

8. Hinweise zu MS3

Bis zum MS3 ist zu machen:

- Klassendiagramm mit den vom Team geplanten Klassen der Fachlogik-Schicht und deren Abhängigkeiten
- Anwendungsfall-Tabelle
- konzeptueller Entwurf der GUI für die zu realisierenden Anwendungsfälle
- Korrektur aller Bugs und Setzen des korrekten Status, inkl.
 - ein neuer Push in den Master-Branch des Git Remote Repository.

Das Dokument Arbeitspakete MS4 gibt an, welcher Teilnehmer welche Anwendungsfälle implementieren soll.

Verbindlicher Termin, konkrete Liefergegenstände und genaue Abgabeorte sind in den Folien zur Einführungsveranstaltung und in den Folien vom MS2 angegeben.

8.1. Klassendiagramm

Das zu MS3 geforderte Klassendiagramm ist für die Komponenten der Schicht "Fachlogik" zu erstellen. Für Komponenten der Schicht "Benutzeroberfläche" sind keine Klassendiagramme zu erstellen. Das Klassendiagramm soll neben den Inhalten der Komponenten der Schicht "Fachlogik" auch die jeweiligen "use"-Beziehungen zu den Schnittstellen-Klassen der Komponenten der Schicht "Datenhaltung" enthalten. Diese Schnittstellen-Klassen müssen jedoch nur mit ihrem Namen dargestellt werden, die Angabe der Methoden ist nicht erforderlich.

Hinweis: Das Klassendiagramm ist während des gesamten Software-Praktikums aktuell zu halten, d.h. es muss immer dem aktuellen Code entsprechen. Eine angepasste Version des Klassendiagramms muss zu MS4 abgegeben werden.

8.1.1. Gestaltung des Klassendiagramms

Folgende Hinweise sind bei der Gestaltung eines Klassendiagramms zu befolgen:

- Das Klassendiagramm folgt dem UML-Standard und ist im pdf-Format abzugeben.
- Die Komponente ist in Pakete strukturiert und alle Klassen befinden sich in einem Paket. Die zu verwendenden Paketbezeichnungen sind im Dokument zum Grobentwurf enthalten.
- Klassennamen beginnen mit einem Großbuchstaben
- Attributnamen beginnen mit einem Kleinbuchstaben
- Methodennamen beginnen mit einem Kleinbuchstaben

- Parameternamen sind aussagekräftig (nicht a, b, c usw.)
- Assoziationen besitzen einen Namen und der Name beginnt mit einem Kleinbuchstaben
- Grenzklassen besitzen den Stereotyp «Grenzklasse»
- Steuerungsklassen besitzen den Stereotyp «Steuerungsklasse»
- Schnittstellenklassen besitzen den Stereotyp «Interface»
- Abhängigkeitsbeziehungen (use-Beziehungen in UML) existieren zwischen zwei Klassen, wenn die eine (abhängige) Klasse eine Methode der anderen (unabhängigen) Klasse verwendet. Use-Beziehungen müssen nur von Steuerungsklassen zu anderen Steuerungsklassen und zu Schnittstellen-Klassen eingetragen werden.
- Realisierungsbeziehungen existieren zwischen Implementierungs- und Schnittstellenklassen.
- Attribute haben einen sinnvollen Datentyp
- Attribute sind privat, falls es keine guten Gründe für eine andere Sichtbarkeit gibt
- Methoden haben einen Ergebnistyp und alle Parameter haben einen Datentyp
- Entitätsklassen müssen im Klassendiagramm nicht aufgeführt werden.

8.1.2. Verwendung einer geschlossenen 3-Schichten Architektur

Alle Systeme im Software-Praktikum sollen in einer geschlossenen 3-Schichten-Architektur umgesetzt werden. Es sollen die drei Schichten Benutzeroberfläche, Fachlogik und Datenhaltung existieren.

Wir wollen weiterhin eine größtmögliche Unabhängigkeit zwischen der GUI und der Datenhaltung herstellen, damit eine Änderung an der Datenbank (z.B. neue Attribute, Aufteilung/Zusammenlegung von Tabellen) nicht zu Änderungen an der GUI führt. Diese Unabhängigkeit soll durch eine strenge Entkopplung der GUI-Klassen von der Datenbank mit Hilfe von Grenzklassen erfolgen:

- Die GUI kennt keine Entitätsklassen. Sie darf also von den Steuerungsklassen niemals Objekte der Entitätsklassen erhalten.
- Die gesamte Kommunikation zwischen GUI und Steuerungsklassen findet mit Hilfe von Grenzklassen statt.

Beispiel 1

Daten eines Kunden holen: Die GUI übergibt den Suchparameter (z.B. Name oder ID) an eine Operation der Steuerungsklasse. Diese holt die gewünschten Kundendaten aus der Datenbank, kopiert diese in ein Objekt der Grenzklasse zum Kunden und übergibt dieses an die GUI als Antwort zurück. Somit muss die GUI die Entitätsklassen nicht kennen und kann ausschließlich mit Grenzklassen arbeiten. Eine Änderung der Entitätsklassen erfordert somit auch keine Änderung der GUI-

Klassen, sondern nur der Steuerungsklassen. Hierdurch erlangt man eine weitgehende Entkopplung der GUI von der Datenhaltung.

Beispiel 2

Neuen Kunden anlegen: In einem Formular in der GUI gibt der Benutzer die erforderlichen Daten zum neuen Kunden ein. Die GUI speichert diese eingegebenen Daten in einem Objekt der Grenzklasse zum Kunden und übergibt dieses Objekt als Parameter an die entsprechende Operation der Steuerungsklasse. Diese Operation erstellt ein neues Objekt der Entitätsklasse zum Kunden, trägt dort die Daten aus der Grenzklasse ein und persistiert das Objekt der Entitätsklasse. Auch in diesem Fall muss die GUI nicht die konkrete Struktur der Entitätsklasse kennen und arbeitet lediglich mit der auf ihre Bedürfnisse zugeschnittenen Grenzklasse.

8.2. Anwendungsfall-Tabellen

Eine Vorlage der Anwendungsfall-Tabelle existiert in Ilias.

Die Anwendungsfall-Tabelle enthält alle Anwendungsfälle eines Teams. Für jeden Anwendungsfall ist anzugeben, durch welche Methode welcher Steuerungsklasse er implementiert wird.

8.3. Konzeptueller Entwurf der GUI

Der konzeptuelle Entwurf der GUI hilft dem Kunden zu verstehen, ob die geplante Benutzeroberfläche geeignet ist, die gewünschten Anwendungsfälle aus dem Lastenheft zu unterstützen.

Der konzeptuelle Entwurf der GUI besteht aus Bildern eines GUI-Builders oder aus Zeichnungen der geplanten Oberfläche der einzelnen Anwendungsfälle.

Auf dem Treffen zum MS3 spielen die Teams die Aktionen der Anwendungsfälle mit Hilfe des konzeptuellen Entwurfs der GUI in einer Präsentation für die Betreuer durch. Hierbei lassen sich Lücken und Unklarheiten leicht erkennen.

Das folgende Kapitel liefert Hinweise zur Gestaltung der GUI.

8.4. GUI der Anwendungsfälle

8.4.1. Robustheit

Im Code der GUI sollen falsche Eingaben und fehlende Angaben abgefangen werden und nicht zu einem Absturz oder einer Fehlfunktion (z.B. Zugriff auf leere Liste) führen.

8.4.2. Rückmeldung an den Benutzer

Die GUI soll dem Benutzer eine Rückmeldung nach der Durchführung einer vom Benutzer gewählten Funktion geben, z.B.: Nach dem Auswählen des Buttons zur Speicherung eines neuen Kunden in der Datenbank, soll die GUI eine Nachricht über das Ergebnis der Speicherung ausgeben, d.h. ob die Speicherung erfolgreich war oder nicht.

8.4.3. Komfort bei der Bedienung

Die GUI soll dem Benutzer ein Mindestma's an Komfort bieten. Beispiel: Wenn die GUI dem Benutzer eine Liste von Kunden zur Auswahl anbietet, dann soll die getroffene Auswahl auch automatisch in vorhandene Eingabefelder eingefügt werden (z.B. der Name des Kunden) und der Benutzer soll diese Eingabe nach der Auswahl nicht noch manuell vornehmen müssen.

8.4.4. Beschriftungen der GUI-Elemente

Die Beschriftungen der GUI-Elemente (Reiter, Buttons, Eingabefelder, Ausgabefelder usw.) sollen für den Benutzer aussagekräftig und informativ sein und sollen passend zum realisierten Anwendungsfall gewählt werden. Beispiel: der Name des Buttons zum Anlegen eines neuen Kunden sollte nicht nur "Ok" sondern "Kunde anlegen" heißen.

8.4.5. Anordnung der GUI-Elemente

Die einzelnen Elemente der GUI (Buttons, Listen, Eingabefenster, usw.) sollen derart angeordnet sein, dass die korrekte Bedienung für den Benutzer leicht zu erkennen ist. Oftmals ist es besser, pro Anwendungsfall eine eigene Seite einzurichten, anstatt durch die gleiche Seite mehrere Anwendungsfälle zu unterstützen und dadurch die Zuordnung der Buttons zu den einzelnen Anwendungsfällen zu verlieren.

9. Hinweise zu MS4

Bis zum MS4 ist zu machen:

- Implementierung der verteilten Arbeitspakete in den entsprechenden Komponenten
 - Das Dokument Arbeitspakete MS4 gibt an, welcher Teilnehmer welche Anwendungsfälle (und ggfs. auch andere Aufgaben) implementieren soll.
- Entwicklerdokumentation der erstellten Steuerungsklassen in der Fachlogik-Schicht mit JavaDoc.
- Klassendiagramm anpassen
- Korrektur aller Bugs und Setzen des korrekten Status

Verbindlicher Termin, konkrete Liefergegenstände und genaue Abgabeorte sind in den Folien zur Einführungsveranstaltung angegeben.

Bei der Implementierung der GUI müssen natürlich auch die Hinweise aus [Abschnitt 8.4](#) beachtet werden.

9.1. Entwicklerdokumentation

Die Entwicklerdokumentation erfolgt in javadoc. Es müssen nur die Steuerungs-Klassen in den Komponenten der Schicht "Fachlogik" mit javadoc-Kommentare versehen werden - Die javadoc-Kommentare sollen nicht bei den Interface-Klassen, sondern nur bei den Implementierungen der Interface-Klassen erstellt werden.

Die javadoc-Kommentare sollen aussagekräftig sein. Sie sollen beschreiben, was eine Methode genau macht, welche Parameter erwartet werden und welches Ergebnis in welcher Situation geliefert wird.

10. Hinweise zu MS5

Bis zum MS5 ist zu machen:

- Korrektur aller Bugs und Setzen des korrekten Status.
- Alle Anwendungsfälle und Integrationstestfälle müssen erfolgreich ablaufen.

Verbindlicher Termin, konkrete Liefergegenstände und genaue Abgabeorte sind in den Folien zur Einführungsveranstaltung angegeben.

Auf der Sitzung zum MS5 sollen alle Teams, die gemeinsam an einem System gearbeitet haben, die Lauffähigkeit dieses Systems vorführen. Hierzu werden die Schritte aus dem Integrationstest durchgeführt.



Systemspezifikation für das
Bankverwaltungssystem (BKS)
Softwarepraktikum

TH Köln - Informatik Labor

Inhaltsverzeichnis

1. Daten des Systems	2
2. Verhalten des Systems	8
2.1. Kontoeröffnungs-Antrag (ko)	8
2.2. Kontoschließungs-Antrag (ks)	8
2.3. Kontolimit-Antrag (kl)	8
2.4. Sachbearbeiter-Anlegen - Antrag (so)	9
2.5. Sachbearbeiter-Löschen - Antrag (sd)	9
2.6. Sachbearbeiter-Bearbeiten - Antrag (sb)	9
2.7. Überweisungsstatus	10
2.8. Kontostatus	11
3. Funktionen des Systems	12
4. Schnittstellen des Systems	13
5. Graphische Benutzerschnittstellen (GUI) des Systems	17
6. Ergänzungen	18
6.1. Kontoauszug	18

v10.0, 2024-01-09 11:50:07 UTC

1. Daten des Systems

In diesem Kapitel werden die persistent zu speichernden Daten des Bankverwaltungssystems spezifiziert. Die Abbildung zeigt die Spezifikation der Daten des Bankverwaltungssystems in Form eines UML Klassendiagramms.

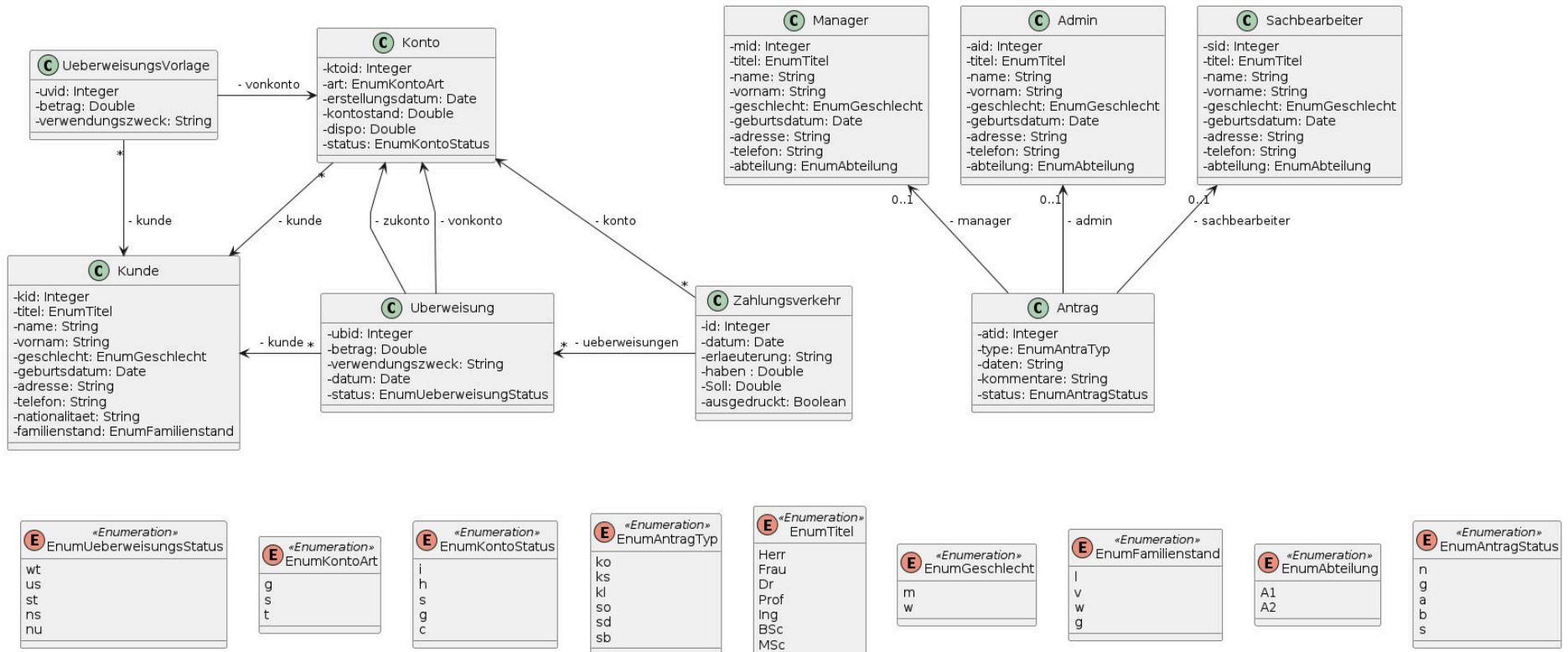


Abbildung 1. Spezifikation der Daten des Bankverwaltungssystems als Klassendiagramm

Es existieren die folgenden Entitäten, Attribute und Beziehungen:

Admin

Ein **Admin** besitzt neben einer Identifikationsnummer (**aid**) persönliche Angaben (**name**, **vorname**, **titel**, **geburtsdatum**, **geschlecht**, **adresse** und **telefon**) und gehört einer **abteilung** an. Einige dieser Angaben sind frei, andere sind aus einer Menge von vorgegebenen Werten auswählbar:

titel:

Herr, Frau, Dr., Prof., Ing-, BSc., MSc.

geschlecht:

m (= männlich), w (= weiblich)

abteilung:

A1 (= Abteilung 1), A2 (= Abteilung 2)

Manager

Ein **Manager** besitzt neben einer Identifikationsnummer (**mid**) persönliche Angaben (**name**, **vorname**, **titel**, **geburtsdatum**, **geschlecht**, **adresse** und **telefon**) und gehört einer **abteilung** an. Einige dieser Angaben sind frei, andere sind aus einer Menge von vorgegebenen Werten auswählbar:

titel:

Herr, Frau, Dr., Prof., Ing-, BSc., MSc.

geschlecht:

m (= männlich), w (= weiblich)

abteilung:

A1 (= Abteilung 1), A2 (= Abteilung 2)

Sachbearbeiter

Ein **Sachbearbeiter** besitzt neben einer Identifikationsnummer (**sid**) persönliche Angaben (**name**, **vorname**, **titel**, **geburtsdatum**, **geschlecht**, **adresse** und **telefon**) und gehört einer **abteilung** an. Einige dieser Angaben sind frei, andere sind aus einer Menge von vorgegebenen Werten auswählbar:

titel:

Herr, Frau, Dr., Prof., Ing-, BSc., MSc.

geschlecht:

m (= männlich), w (= weiblich)

abteilung:

A1 (= Abteilung 1), A2 (= Abteilung 2)

Kunde

Ein **Kunde** besitzt neben einer Identifikationsnummer (**kid**) persönliche Angaben (**name**, **vorname**, **titel**, **geburtsdatum**, **geschlecht**, **adresse**, **telefon** und **familienstand**). Einige dieser Angaben sind frei, andere sind aus einer Menge von vorgegebenen Werten auswählbar:

titel:

Herr, Frau, Dr., Prof., Ing-, BSc., MSc.

geschlecht:

m (= männlich), w (= weiblich)

familienstand:

l (= ledig), v (= verheiratet), w (= verwitwet), g (= geschieden)

Antrag

Ein **Antrag** besitzt neben einer Identifikationsnummer (**atid**) einen **typ**, einen **status** und jeweils ein Feld für **daten** und **kommentare**. Das Feld **daten** enthält dabei alle Daten in Fließtext, die für einen Antrag benötigt werden; das Feld **kommentare** enthält Kommentare zum Antrag des Antragstellers an den Empfänger. Einige dieser Angaben sind frei, andere sind aus einer Menge von vorgegebenen Werten auswählbar:

typ:

- ko (= Konto-Eröffnungsantrag)*
- ks (= Konto-Schließungsantrag)*
- kl (= Konto-Limit-Antrag)*
- so (= Sachbearbeiter-Anlegen-Antrag)*
- sd (= Sachbearbeiter-Löschen-Antrag)*
- sb (= Sachbearbeiter-Bearbeiten-Antrag)*

status:

- n (= neu)*
- g (= genehmigt)*
- a (= abgelehnt)*
- b (= bearbeitet)*
- s (= storniert)*

In das Feld **daten** sollen dabei die für die Bearbeitung wichtige Informationen für den Bearbeiter des Antrags eingefügt werden. Achtung: diese Daten sind unstrukturiert und **nicht** automatisiert auszulesen.

Das Attribut **admin** referenziert auf einen Admin, das Attribut **sachbearbeiter** auf einen Sachbearbeiter und das Attribut **manager** auf einen Manager:

Für Anträge der Typen *ko* und *kl* referenziert hierbei **sachbearbeiter** auf den Sachbearbeiter, der den Antrag erstellt hat und **manager** auf den Manager, an den der Antrag gerichtet ist. Das Attribut **admin** bleibt hierbei jeweils leer.

Für Anträge der Typen *so*, *sd* und *sb* referenziert hierbei **manager** auf den Manager, der den Antrag erstellt hat und **admin** auf den Admin, an den der Antrag gerichtet ist. Das Attribut **sachbearbeiter** bleibt hierbei jeweils leer.

Für Anträge des Typs *ks* referenziert hierbei **sachbearbeiter** auf den Sachbearbeiter, der den Antrag erstellt hat, **manager** auf den Manager, an den der Antrag gerichtet ist und der diesen genehmigen soll, und **admin** auf den Admin, der den Antrag bearbeiten soll. Hierbei bleibt das Attribut **admin** beim Erstellen des Antrags zunächst leer und wird erst durch den Manager beim Genehmigen mit einem Admin befüllt.

Konto

Ein **Konto** besitzt eine Identifikationsnummer (**ktoid**), ein **erstellungsdatum** und ist von einer bestimmten Kontoart (**art**). Darüber hinaus wird der aktuelle **kontostand**, der Dispositionskredit (**dispo**) und der **status** des Kontos gespeichert. Es ist immer einem bestimmten Kunden (**kunde**) zugehörig. Einige dieser Angaben sind frei, andere sind aus einer Menge von vorgegebenen Werten auswählbar:

art:

g (= Girokonto), *s* (= Sparkonto), *t* (= Tageskonto)

status:

i (= init), *h* (= haben), *s* (= soll), *g* (= gesperrt), *c* (= geschlossen)

Ueberweisung

Eine **Ueberweisung** besitzt neben einer Identifikationsnummer (**ubid**) einen **betrag**, der an einem bestimmten **datum** mit einem bestimmten **verwendungszweck** von einem Konto (**vonkonto**) zu einem anderen Konto (**zukonto**) überwiesen werden soll. Darüber hinaus wird der **status** der Überweisung gesichert. Eine Überweisung ist immer einem Kunden (**kunde**) zugehörig. Einige dieser Angaben sind frei, andere sind aus einer Menge von vorgegebenen Werten auswählbar:

status:

wt (= wartend), *us* (= überwiesen), *st* (= storniert), *ns* (= nicht stornierbar), *nu* (= nicht überweisbar)

UeberweisungsVorlage

Eine **UeberweisungsVorlage** dient dem Zweck, eine Überweisung zu erleichtern, indem einige Werte der Überweisung gespeichert und bei Bedarf geladen werden können. Dies verringert die Anzahl an Feldern, die ein Benutzer bei einer Überweisung manuell ausfüllen muss. Die Überweisungs-Vorlage besitzt eine Identifikationsnummer (**uvid**), den **betrag**, der überwiesen werden soll, die Angabe des **verwendungszweck**, und das Konto (**vonkonto**), von dem überwiesen werden soll. Sie ist immer einem Kunden (**kunde**) zugehörig.

Zahlungsverkehr

Bei jeder erfolgreichen Überweisung soll für beide betreffenden Konten ein **Zahlungsverkehr** angelegt werden. Ein Zahlungsverkehr enthält neben einer Identifikationsnummer (**zid**) das **datum**, an dem die Überweisung ausgeführt wurde, eine **erläuterung** (siehe unten), und eines der beiden betreffenden Konten (**konto**). Für das Konto, von dem Geld abgebucht wurde, wird der **betrag** der Überweisung im Feld **soll** vermerkt; für das Konto, auf das Geld gebucht wird, wird der **betrag** im Feld **haben** vermerkt. Das Attribut **ausgedruckt** dient als Hinweis, ob ein Zahlungsverkehr bereits auf einen Kontoauszug gedruckt wurde.

Ebenso soll bei einer Ein-/Auszahlung ein Zahlungsverkehr angelegt werden. Bei einer Einzahlung wird der **betrag** im Feld **haben** vermerkt, bei einer Auszahlung im Feld **soll**. Im Feld **erläuterung** wird der Grund für den Zahlungsverkehr angegeben. Hier sind, je nach Grund des Zahlungsverkehrs, unterschiedliche Angaben zu machen:

- im Falle einer Überweisung, die zu dem Zahlungsverkehr geführt hat: der Text aus dem Attribut **verwendungszweck** der zugehörigen Überweisung.
- im Falle einer Einzahlung, die zu dem Zahlungsverkehr geführt hat: der Text "**Einzahlung**".
- im Falle einer Auszahlung, die zu dem Zahlungsverkehr geführt hat: der Text "**Auszahlung**".

ausgedruckt *false bzw. 0 (= nicht ausgedruckt), true bzw. 1 (= ausgedruckt)*

Einige dieser Angaben sind frei, andere sind aus einer Menge von vorgegebenen Werten auswählbar:

2. Verhalten des Systems

2.1. Kontoeröffnungs-Antrag (ko)

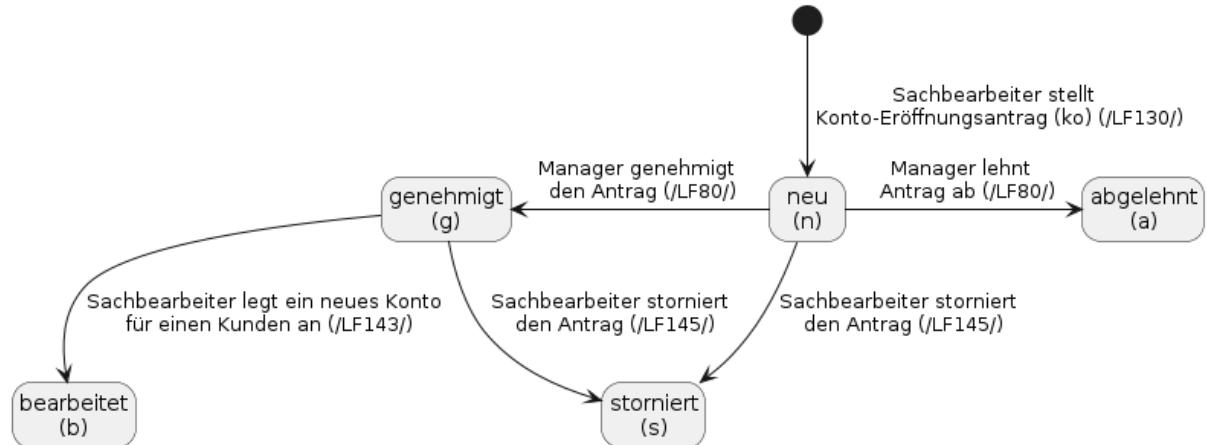


Abbildung 2. Bearbeitung eines Kontoeröffnungs-Antrags

2.2. Kontoschließungs-Antrag (ks)

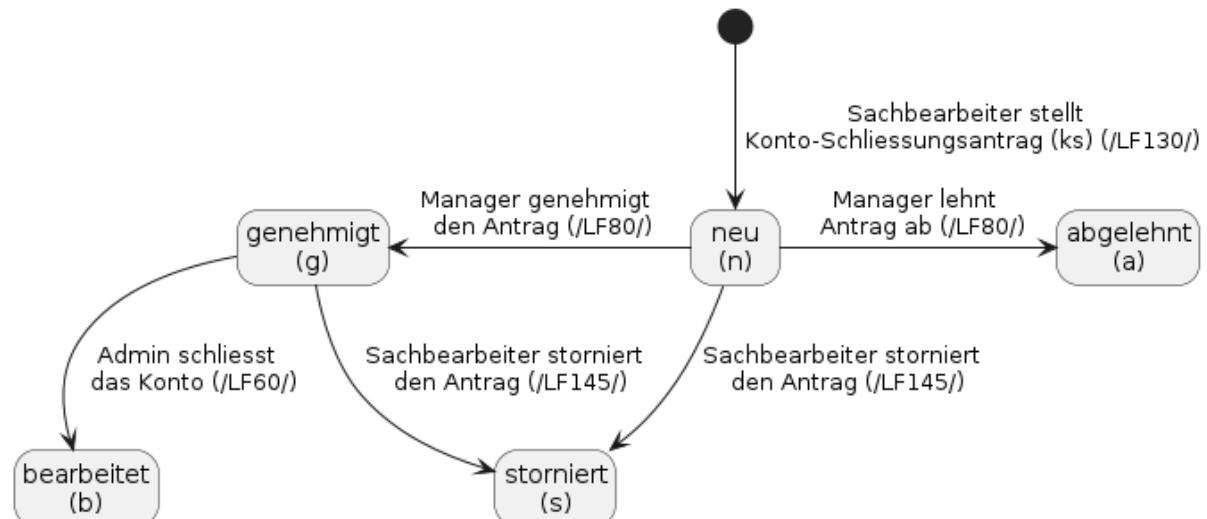


Abbildung 3. Bearbeitung eines Kontoschließungs-Antrags

2.3. Kontolimit-Antrag (kl)

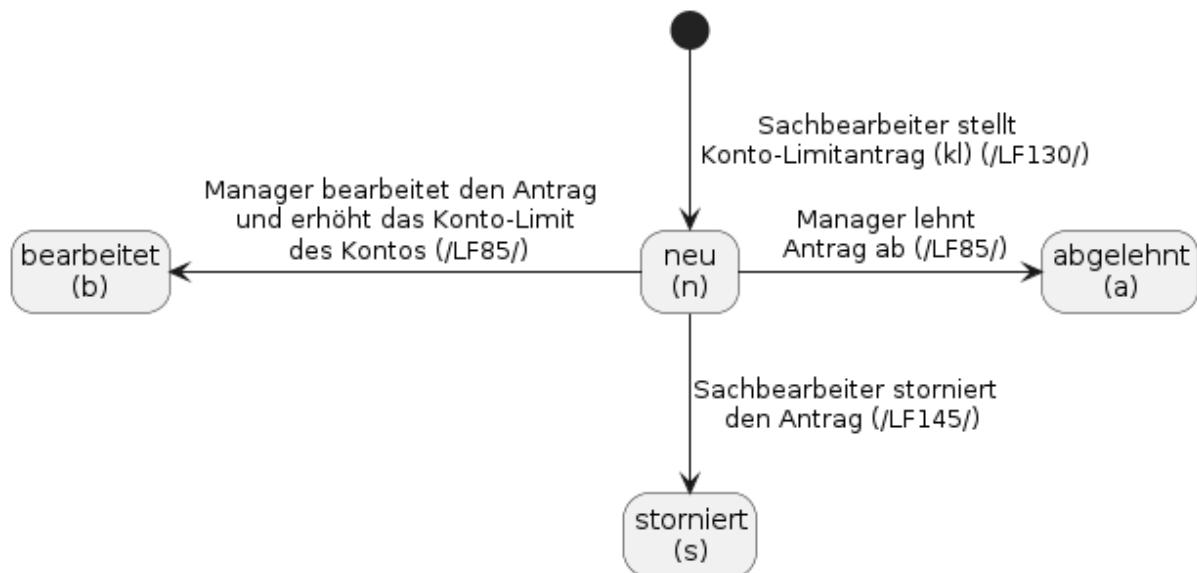


Abbildung 4. Bearbeitung eines Kontolimit-Antrags

2.4. Sachbearbeiter-Anlegen - Antrag (so)

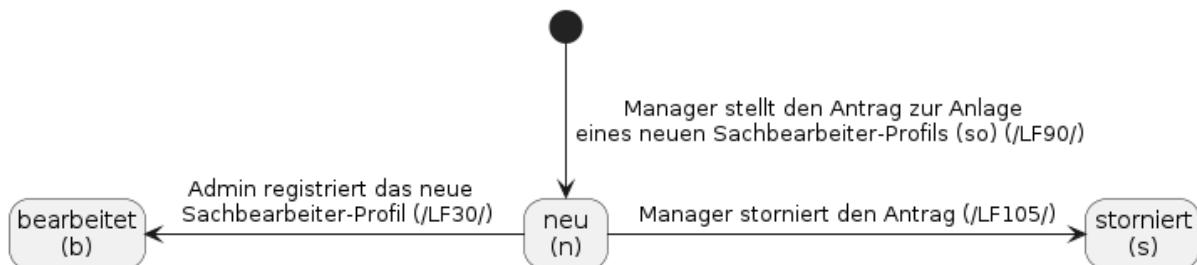


Abbildung 5. Bearbeitung eines Sachbearbeiter-Anlegen-Antrags

2.5. Sachbearbeiter-Löschen - Antrag (sd)

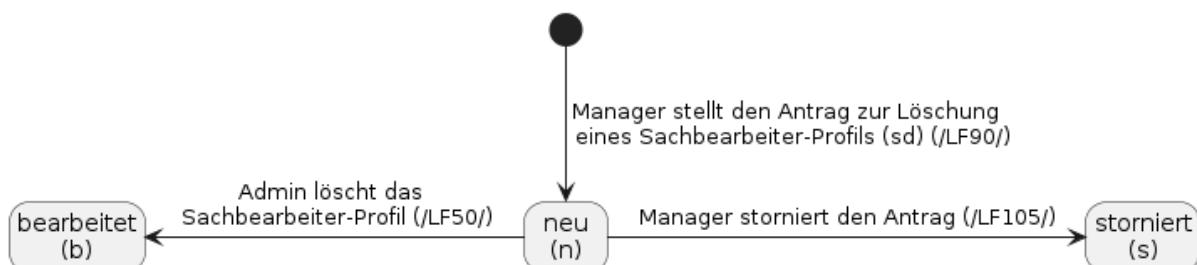


Abbildung 6. Bearbeitung eines Sachbearbeiter-Löschen-Antrags

2.6. Sachbearbeiter-Bearbeiten - Antrag (sb)

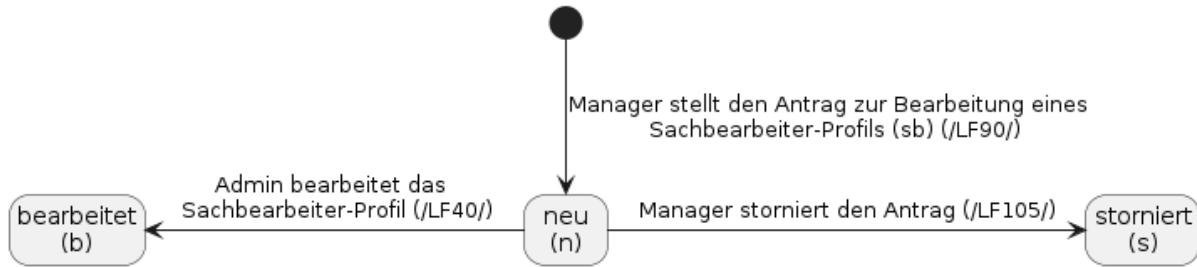


Abbildung 7. Bearbeitung eines Sachbearbeiter-Bearbeiten-Antrags

2.7. Überweisungsstatus

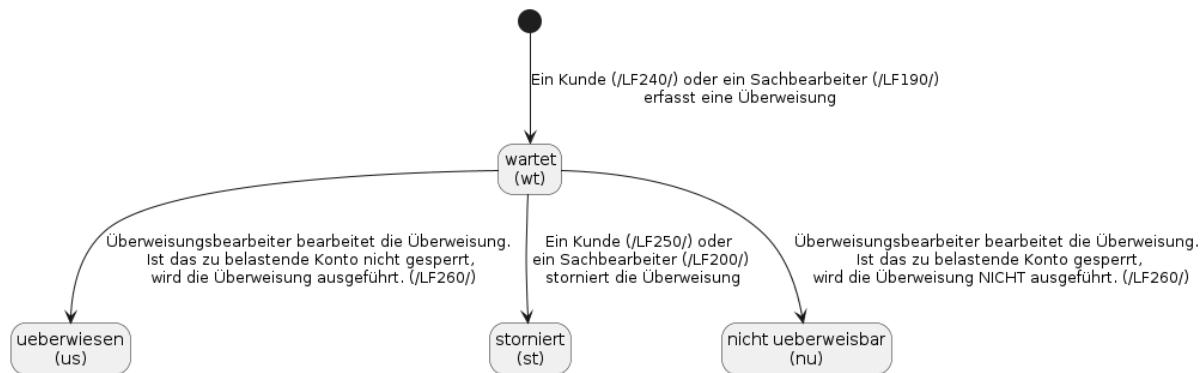


Abbildung 8. Zustände einer Überweisung zur Laufzeit

2.8. Kontostatus

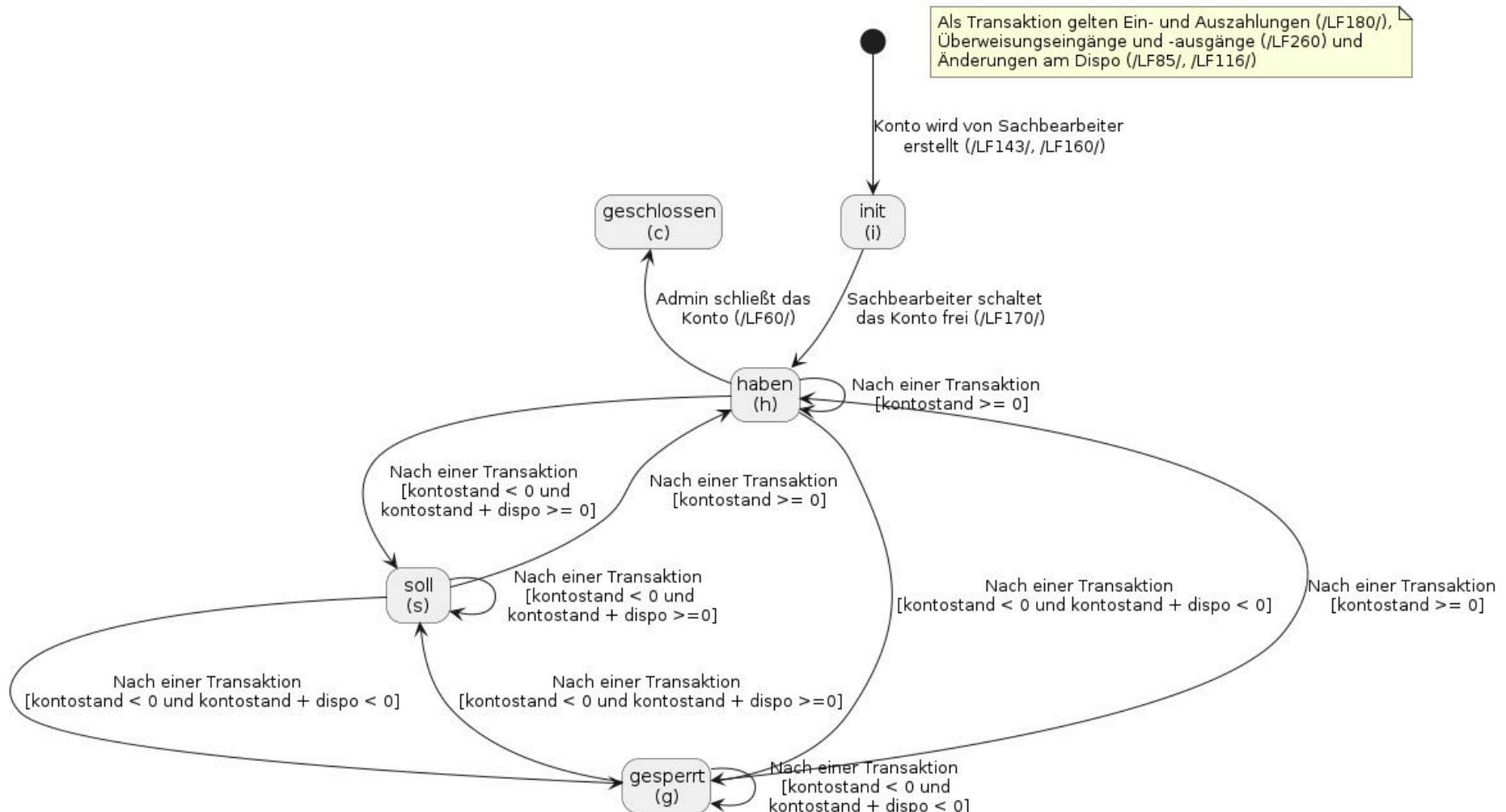


Abbildung 9. Zustände eines Kontos während der Laufzeit

3. Funktionen des Systems

Die Spezifikation der im Lastenheft enthaltenen Anwendungsfälle in Form einer detaillierten Anwendungsfall-Beschreibung oder eines Aktivitätsdiagramms ist normalerweise Teil der Systemspezifikation. Im Software-Praktikum ist auf die Vorgabe dieser Spezifikation bewusst verzichtet worden, damit sich die Teilnehmer des Software-Praktikums selbst intensiv mit den Anwendungsfällen auseinandersetzen und eigene kreative Lösungsideen einbringen können.

4. Schnittstellen des Systems

Anw.-Fall	Involvierte Schnittstelle	Kurzbeschreibung
/LF10/	Dialog Managerprofil	Manager können angezeigt, angelegt, bearbeitet und gelöscht werden
/LF20/	Dialog AdminAntraegeAnzeigen	Liste aller vom Admin zu bearbeitenden Anträge anzeigen; Auswahl eines Antrags zur Darstellung der Antrags-Details; Auswahl eines Antrags zum Start der Bearbeitung (durch /LF30/, /LF40/, /LF50/ oder /LF60/); Auswahl eines Antrags zum Setzen des Status auf bearbeitet.
/LF30/	Dialog SachbearbeiterAnlegen	Anlegen eines Sachbearbeiter-Profil entsprachend des in /LF20/ ausgewählten Antrags
/LF40/	Dialog SachbearbeiterBearbeiten	Bearbeiten eines Sachbearbeiter-Profil entsprachend des in /LF20/ ausgewählten Antrags
/LF50/	Dialog SachbearbeiterLöschen	Löschen eines Sachbearbeiter-Profil entsprachend des in /LF20/ ausgewählten Antrags
/LF60/	Dialog KontoSchließen	Schließen eines Kontos entsprechend des in /LF20/ ausgewählten Antrags
/LF70/	Dialog Wartungsarbeit	Anzeige einer Liste aller Anträge und Löschen stornierter oder bearbeiteter Anträge

Anw.-Fall	Involvierte Schnittstelle	Kurzbeschreibung
/LF80/	Dialog koUndKsAntraegeAnzei gen	Anzeige einer Liste aller Kontoeröffnungs- und Kontoschließungs-Anträge und Anzeigen, Genehmigen oder Ablehnen eines Antrags Besonderheit beim Genehmigen eines Kontoschließungs-Antrags: Anzeigen der Liste aller Admins und Auswahl eines Admins aus dieser Liste
/LF85/	Dialog KontolimitAntraege	Anzeige einer Liste aller Kontolimit-Anträge
/LF85/	Dialog KontolimitBearbeiten	Bearbeiten des Dispos eines Kontos
/LF90/	Dialog SachbearbeiterprofilAn trag	Stellen von Sachbearbeiter-Anträgen (so, sb, sd)
/LF100/	Dialog ManagerAntraege	Liste aller vom Manager zu bearbeitenden Anträge anzeigen
/LF105/	Dialog ManagerAntraegeStornieren	Auswahl und Stornieren eines entsprechenden Antrags
/LF110/	Dialog ÜberzogeneKontenAnz eignen	Anzeige aller überzogenen Konten
/LF113/	Dialog MahnungSchreiben	Auswahl eines entsprechenden Kontos und Schreiben einer Mahnung
/LF113/	Ausgabeerzeugnis Mahnung	Generierung einer Mahnung
/LF116/	Dialog KontoSperrenEntsperre n	Auswahl eines entsprechenden Kontos und Sperren/Entsperren des Kontos
/LF120/	Dialog KundenprofilAnlegenB earbeiten	Anlegen oder Bearbeiten eines Kundenprofils
/LF130/	Dialog KontoAntraegeStellen	Stellen von Konto-Anträgen (ko, ks, kl)
/LF140/	Dialog SachbearbeiterAntraeg eAnzeigen	Anzeige einer Liste aller an den Sachbearbeiter gestellten Anträge

Anw.-Fall	Involvierte Schnittstelle	Kurzbeschreibung
/LF143/	Dialog KontoEroeffnungsantragBearbeiten	Auswahl eines entsprechenden Antrags , Anzeige einer Kundenliste, Auswahl eines Kunden und Anlegen eines neuen Kontos
/LF145/	Dialog SachbearbeiterAntragStornieren	Auswahl und Stornieren eines entsprechenden Antrags
/LF150/	Dialog KontoauszugAnzeigen	Auswahl des Kunden, Auswahl eines Kontos des Kunden und Anzeigen der Kontobewegungen des Kontos innerhalb eines gewünschten Datumsintervalls
/LF170/	Dialog KontoFreischalten	Freischalten eines Kontos
/LF180/	Dialog GeldEinzahlenAbheben	Anzeige einer Kontoliste eines Kunden und Geld Ein-/Auszahlen
/LF190/	Dialog SachbearbeiterÜberweisungErfassen	Anzeige einer Kundenliste, Auswahl eines Kundenkontos, Erfassen einer Überweisung
/LF200/	Dialog ÜberweisungStornieren	Anzeige aller stornierbaren Überweisungen und Stornieren einer Überweisung
/LF210/	Dialog KundenprofilBearbeiten	Bearbeiten der eigenen Profildaten
/LF220/	Dialog KontolisteAnzeigen	Anzeige eigener Kontoliste
/LF225/	Dialog KontoauszugAnzeigen	Anzeigen des Kontoauszugs eines gewünschten Kontos
/LF225/	Ausgabeerzeugnis KontoauszugDrucken	Drucken des Kontoauszugs eines gewünschten Kontos
/LF230/	Dialog ÜberweisungVorlage	Anzeigen, Anlegen, Bearbeiten und Löschen einer UeberweisungsVorlage
/LF240/	Dialog ÜberweisungErfassen	Auswahl einer UeberweisungsVorlage und Erfassen einer Ueberweisung
/LF250/	Dialog ÜberweisungStornieren	Anzeige einer Liste aller stornierbaren Ueberweisungen und Stornieren einer gewählten Ueberweisung
/LF260/	Dialog ÜberweisungAusführen	Anzeige aller wartenden Ueberweisungen und Ausführen von gewählten Ueberweisungen

Anw.-Fall	Involvierte Schnittstelle	Kurzbeschreibung
/LF270/	Dialog ÜberweisungLoeschen	Anzeige aller überwiesenen, nicht überweisbaren und stornierten Ueberweisungen und Löschen dieser

5. Graphische Benutzerschnittstellen (GUI) des Systems

Die Spezifikation der in [Kapitel 4](#) identifizierten graphischen Benutzerschnittstellen ist normalerweise Teil der Systemspezifikation. Im Software-Praktikum ist die Spezifikation der graphischen Benutzerschnittstellen jedoch eine Aufgabe der Teilnehmer. Hierfür bildet die Spezifikation der Schnittstellen in [Kapitel 4](#) die Grundlage.

6. Ergänzungen

6.1. Kontoauszug

Das Bild zeigt ein Beispiel eines Kontoauszugs, wie er im Anwendungsfall [/LF150/](#) erzeugt werden soll.

Muster-Bank			
Konto-Nr. 12345678	BLZ 910 111 21	Muster-Bank Musterstadt	Auszug Nr. 1
Datum	Buchungstext	Betrag	
29.10.2009	Einkauf im Super-Markt	47,93-	
30.10.2009	Mikrowellenherd aus OnlineShopXY	49,99-	
02.11.2009	Lohn 10/2009 von MusterAG	1207,42+	
04.11.2009	Überweisung Tante Herta	100,00+	
Musterstadt, den 06.11.2009 16:52 Uhr		Saldo	1209,50+

Abbildung 10. Beispiel-Kontoauszug



Grobentwurf für das Bankverwaltungssystem (BKS) *Softwarepraktikum*

TH Köln - Informatik Labor

Version 10.3, 2024-02-20 08:23:57 UTC

Inhaltsverzeichnis

1. Speicherstruktur der Daten	2
2. Systemarchitektur	3
3. Komponentenspezifikationen	5
3.1. Bootloader	5
3.1.1. Funktionen	5
3.1.2. Schnittstellen	5
3.1.3. Besonderheiten	5
3.2. ComponentController	5
3.2.1. Funktionen	5
3.2.2. Schnittstellen	5
3.3. BKSDBModel	5
3.3.1. Funktionen	6
3.3.2. Schnittstellen	6
3.4. DatenhaltungAPI	7
3.4.1. Funktionen	7
3.4.2. Schnittstellen	7
3.5. SteuerungAPI	7
3.5.1. Funktionen	7
3.5.2. Schnittstellen	7
3.6. AdminGUI	8
3.6.1. Funktionen	8
3.6.2. Schnittstellen	8
3.7. AdminSteuerung	8
3.7.1. Funktionen	8
3.7.2. Schnittstellen	8
3.8. AdminDaten	10
3.8.1. Funktionen	10
3.8.2. Schnittstellen	10
3.9. KundeGUI	12
3.9.1. Funktionen	12
3.9.2. Schnittstellen	13
3.10. KundeSteuerung	13
3.10.1. Funktionen	13
3.10.2. Schnittstellen	13
3.11. KundeDaten	15
3.11.1. Funktionen	15

3.11.2. Schnittstellen	15
3.12. KontoGUI	16
3.12.1. Funktionen	16
3.12.2. Schnittstellen	16
3.13. KontoSteuerung	16
3.13.1. Funktionen	17
3.13.2. Schnittstellen	17
3.14. KontoDaten	18
3.14.1. Funktionen	18
3.14.2. Schnittstellen	19
3.15. ManagerGUI	22
3.15.1. Funktionen	22
3.15.2. Schnittstellen	22
3.16. ManagerSteuerung	23
3.16.1. Funktionen	23
3.16.2. Schnittstellen	23
3.17. ManagerDaten	25
3.17.1. Funktionen	25
3.17.2. Schnittstellen	25
3.18. SachbearbeiterGUI	27
3.18.1. Funktionen	27
3.18.2. Schnittstellen	28
3.19. SachbearbeiterSteuerung	28
3.19.1. Funktionen	28
3.19.2. Schnittstellen	28
3.20. SachbearbeiterDaten	30
3.20.1. Funktionen	30
3.20.2. Schnittstellen	30

Changelog:

v10.1 : Im Diagramm von IKontoService Rückgabetyp auf List<Konto> geändert von List<KontoGrenz>

v10.2 : <<uses>> durch \<<use>> ersetzt im IDatabase Diagramm

v10.3 : Ein Interface hat kein Bereich für Attribute und Stereotypen werden großgeschrieben

1. Speicherstruktur der Daten

Die konkreten Speicherstrukturen der Daten eines Systems werden typischerweise erst im Feinentwurf festgelegt.

Im Software-Praktikum legen wir diese jedoch bereits im Grobentwurf fest, weil diese Strukturen für alle Teilnehmer vorgegeben sind und somit alle Teilnehmer auf den gleichen Datenstrukturen Ihren eigenen Feinentwurf durchführen können. (Achtung: Nicht im Systementwurfs-Praktikum nachmachen!)

Die [Abbildung 1](#) zeigt die konkrete Umsetzung der Datenspezifikation aus der Systemspezifikation in der BKS-Datenbank.

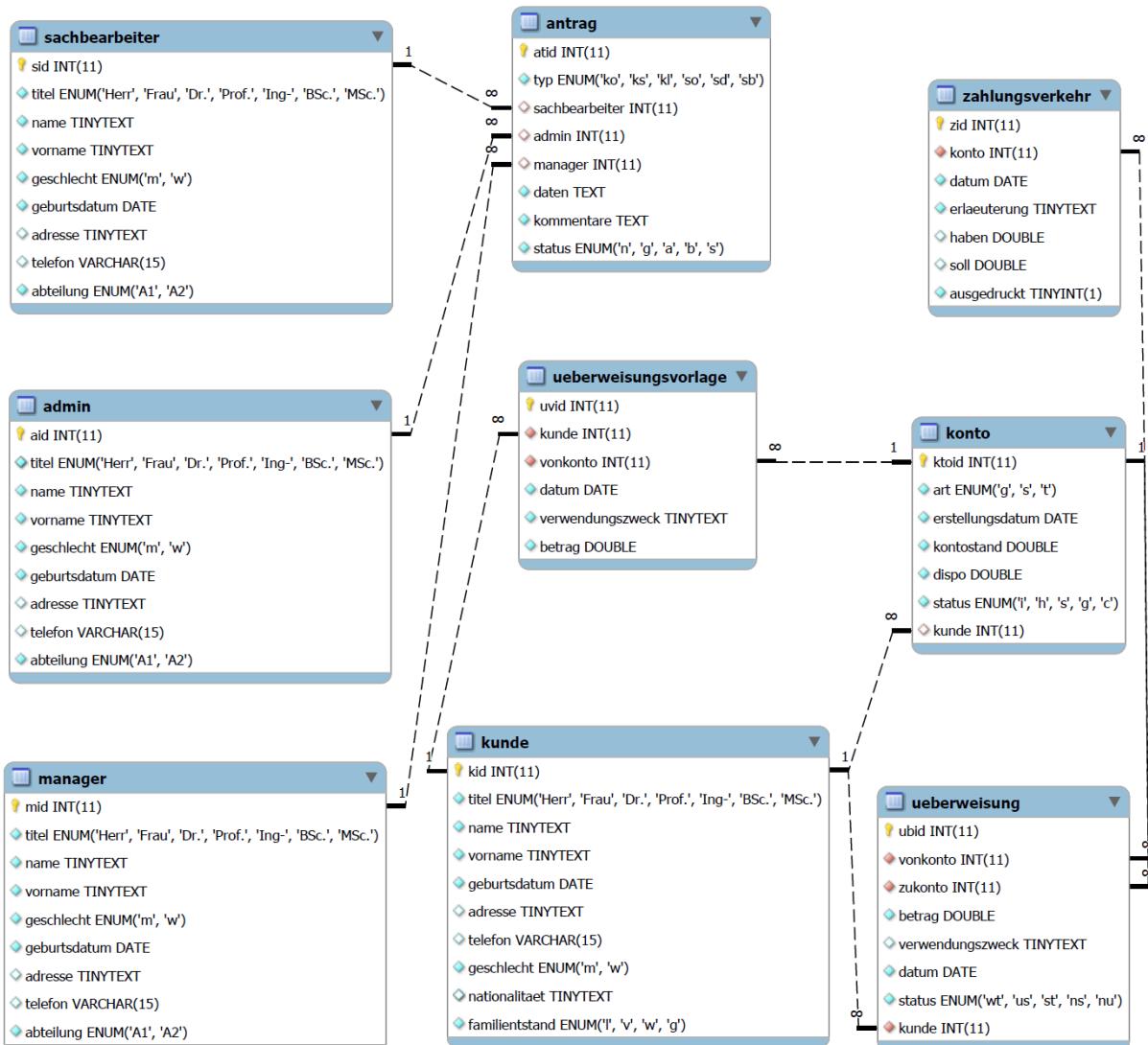


Abbildung 1. Konkrete Umsetzung der Datenspezifikation in der BKS-Datenbank

2. Systemarchitektur

In [Abbildung 2](#) sind alle Komponenten des BKS-Systems zusammen mit ihren angebotenen und zu verwendenden Schnittstellen zu erkennen. Eine Komponente im Softwarepraktikum entspricht genau einem Java-Modul. Jede Komponente wird in genau einem Maven-Projekt entwickelt. Der Name dieses Maven-Projekts in Gitlab entspricht dem Namen der Komponente (jedoch nur Kleinbuchstaben).

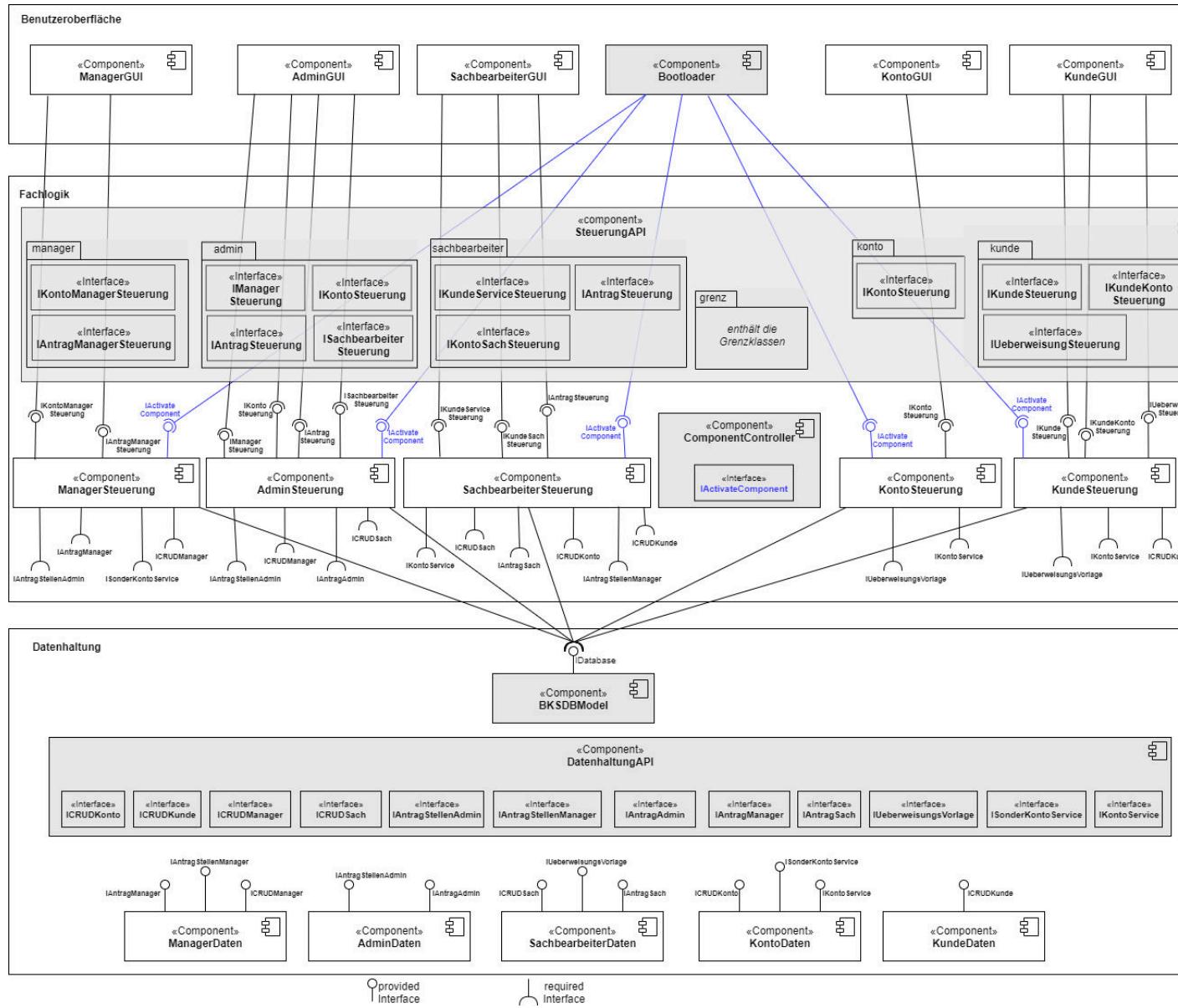


Abbildung 2. BKS Komponentendiagramm mit allen Schnittstellen

3. Komponentenspezifikationen

3.1. Bootloader

3.1.1. Funktionen

Die Bootloader-Komponente stellt das zentrale Startprogramm des BKS-Systems bereit. Es handelt sich dabei um eine Oberfläche, von der aus die speziellen Bedienoberflächen der Komponenten `AdminGUI`, `KundeGUI`, `KontoGUI`, `ManagerGUI` und `SachbearbeiterGUI` gestartet werden können. Der Start der speziellen Bedienoberflächen dieser Komponenten geschieht durch die Verwendung der jeweiligen Implementierung der `IActivateComponent`-Schnittstelle der zugehörigen Komponenten auf der Fachlogik-Schicht.

Diese Komponente wird von den Organisatoren des Software-Praktikums implementiert.

3.1.2. Schnittstellen

Diese Komponente bietet keine Schnittstellen für andere Komponenten an.

3.1.3. Besonderheiten

Dieser Komponente müssen zur Ausführungszeit alle anderen Komponenten und alle verwendeten Bibliotheken zur Verfügung stehen.

3.2. ComponentController

3.2.1. Funktionen

Die ComponentController-Komponente stellt die Interface-Klasse `IActivateComponent` im Paket `de.thkoeln.swp.bks.componentcontroller.services` zur Verfügung. Diese muss von den Komponenten der Fachlogik-Schicht implementiert werden.

Diese Komponente wird von den Organisatoren des Software-Praktikums implementiert.

3.2.2. Schnittstellen

Diese Komponente bietet keine implementierten Schnittstellen für andere Komponenten an.

3.3. BKSDModel

3.3.1. Funktionen

Die **BKSDBModel**-Komponente besteht aus Entitätsklassen, welche die Tabellen oder Entitäten der Datenbank des Bankverwaltungssystems repräsentieren wie sie in der BKS Systemspezifikation spezifiziert sind. Es existieren zwei identische Instanzen der Datenbank des Bankverwaltungssystems: *bksdb_dev* (Development) und *bksdb_prod* (Produktiv). Entsprechend dazu enthält die **BKSDBModel**-Komponente auch zwei Persistence-Units namens *BKSDBDEVPU* und *BKSDBPRODPU*, mithilfe derer auf die entsprechende Instanz der Datenbank zugegriffen werden kann. Während der gesamten Entwicklungsphase muss die *bksdb_dev*-Datenbank verwendet werden, im Quellcode sollte daher die Persistence-Unit *BKSDBDEVPU* genutzt werden. Erst für die Integration des Gesamtsystems (d.h. zum MS5) soll auf die *bksdb_prod*-Datenbank zugegriffen werden.

Die **BKSDBModel**-Komponente wurde bereits implementiert und ist im Git-Repository verfügbar, darf aber nicht verändert werden!

Die Klassen der Entitäten werden von dieser Komponente im Paket `de.thkoeln.swp.bks.bksdbmodel.entities` bereit gestellt.

3.3.2. Schnittstellen

IDatabase

Die **BKSDBModel**-Komponente bietet die **IDatabase**-Schnittstelle an. Diese wird von allen anderen Komponenten verwendet, um über einen systemweit einzigen **EntityManager** einen konsistenten Zugriff auf die Datenbank zu ermöglichen.

Die Schnittstelle ist in [Abbildung 3](#) spezifiziert.

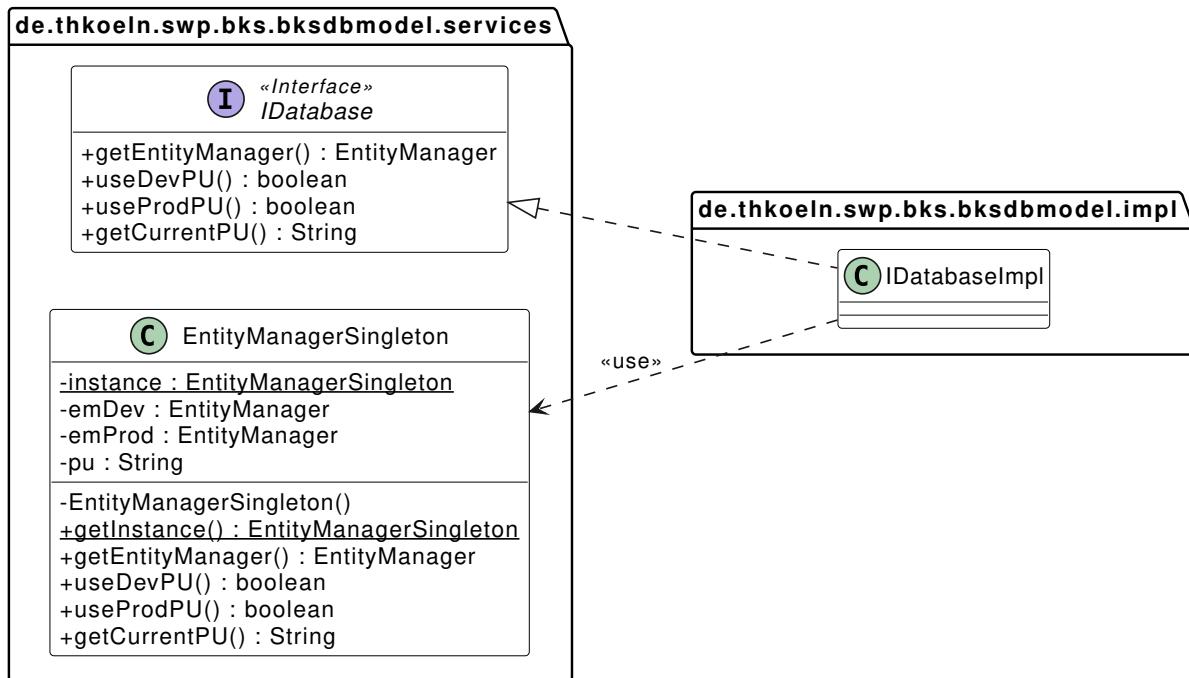


Abbildung 3. **IDatabase**-Schnittstelle der **BKSDBModel**-Komponente

getCurrentPU

liefert eine Information über die aktuell ausgewählte Persistence Unit (PU). Der Rückgabewert ist vom Typ **String** und hat entweder den Wert *BKSDEVPU* oder *BKSPRODPU*.

Diese Schnittstelle ist bereits implementiert und muss von den Teilnehmern des SWP verwendet werden.

3.4. DatenhaltungAPI

3.4.1. Funktionen

Diese Komponente wurde von den Organisatoren des Software-Praktikums bereits implementiert.

Diese Komponente enthält alle Interface-Klassen der Schicht Datenhaltung. Diese Interface-Klassen werden von anderen Komponenten der Schicht Datenhaltung implementiert. Diese Implementierungen werden von den Komponenten der Schicht Fachlogik für Zugriffe auf die gespeicherten Daten verwendet.

3.4.2. Schnittstellen

Diese Komponente bietet keine implementierten Schnittstellen für andere Komponenten an.

3.5. SteuerungAPI

3.5.1. Funktionen

Diese Komponente wurde von den Organisatoren des Software-Praktikums bereits implementiert.

Diese Komponente enthält alle Interface-Klassen der Schicht Steuerung. Diese Interface-Klassen werden von anderen Komponenten der Schicht Steuerung implementiert. Diese Implementierungen werden von den Komponenten der Schicht GUI für Zugriffe auf die Anwendungsfälle benötigt. Zusätzlich bietet diese Komponente die benötigten Grenzklassen an.

3.5.2. Schnittstellen

Diese Komponente bietet keine implementierten Schnittstellen für andere Komponenten an.

3.6. AdminGUI

3.6.1. Funktionen

Die **AdminGUI**-Komponente stellt die Benutzeroberfläche für alle Anwendungsfälle des Admins zur Verfügung.

Sie soll mittels JavaFX realisiert werden.

Die **AdminApp**-Klasse und **AdminController**-Klasse sowie eine **admin.fxml**-Datei werden wie folgt vorgegeben:

- im **src/main/java**-Ordner:
 - `de.thkoeln.swp.bks.admingui.application.AdminApp`
 - `de.thkoeln.swp.bks.admingui.application.AdminController`
- im **src/main/resources**-Ordner:
 - `admin.fxml`

Der weitere Feinentwurf dieser Komponente ist nicht vorgegeben und soll von dem **Admin**-Team eigenständig erstellt werden.

3.6.2. Schnittstellen

Diese Komponente bietet keine Schnittstellen für andere Komponenten an.

3.7. AdminSteuerung

3.7.1. Funktionen

Diese Komponente realisiert die folgenden Anwendungsfälle: /LF10/, /LF20/, /LF30/, /LF40/, /LF50/, /LF60/, /LF70/.

Zur Realisierung der Anwendungsfälle darf nicht direkt auf die Datenbank zugegriffen werden. Es dürfen nur die Schnittstellen verwendet werden, für die in [Abbildung 2](#) der Zugriff definiert wurde.

3.7.2. Schnittstellen

IActivateComponent

Die **AdminSteuerung**-Komponente implementiert die **IActivateComponent**-Schnittstelle (und exportiert diese Implementierung), die von der **Bootloader**-Komponente verwendet wird, um die Komponente zu aktivieren und zu deaktivieren. Die Implementierung muss dafür im Paket `de.thkoeln.swp.bks.adminsteuerung.impl` mit dem Namen `IActivateComponentImpl`

liegen, da diese sonst vom Bootloader nicht gefunden werden kann.

Die Schnittstellen-Klasse **IActivateComponent** wird von der Komponente **ComponentController** bereitgestellt.

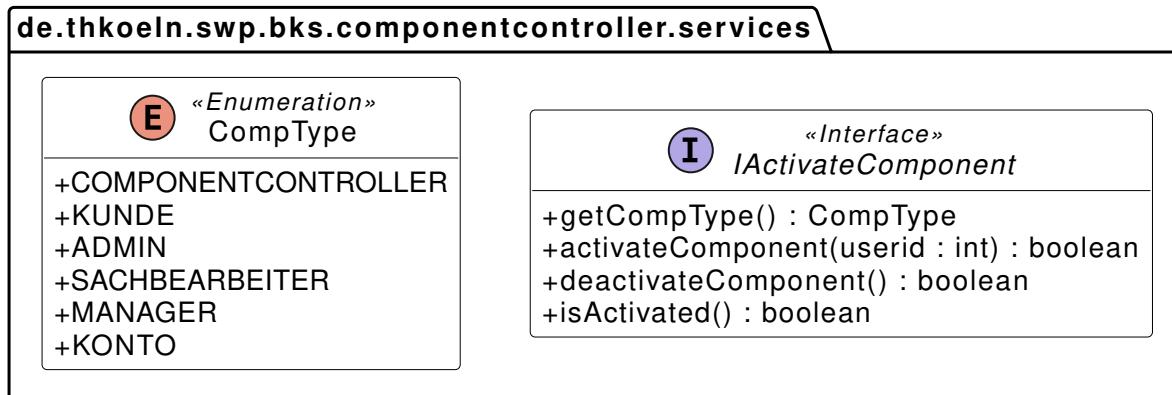


Abbildung 4. **IActivateComponent**-Schnittstelle zur Verwendung in der **AdminSteuerung**-Komponente

Der Parameter der **activateComponent** - Funktion der **IActivateComponent** - Schnittstelle enthält die jeweilige ID des Admins, für den die Komponente aktiviert werden soll. Falls eine falsche ID übergeben wird, sollte die Komponente nicht aktiviert werden und **false** zurückliefern. Zur Aktivierung der Komponente für einen Beispiel-Admin mit **userid=14** und **name=Max Mustermann** wird die **activateComponent**-Methode mit **14** aufgerufen. Die Komponente sollte nur dann aktiviert werden, wenn dieser Admin in der Datenbank bereits existiert.

Die Komponente kann die beiden Zustände *Aktiviert* und *Deaktiviert* annehmen. Die Methoden der Schnittstelle **IActivateComponent** können zu Zustandsübergängen führen:

- Wird die Komponente gestartet, so befindet sie sich zunächst im Zustand *Deaktiviert*.
- Wird im Zustand *Deaktiviert* **activateComponent()** mit zulässiger ID aufgerufen, gibt die Methode ein **true** zurück und geht in den Zustand *Aktiviert*.
- Wird **activateComponent()** mit einer nicht zulässigen ID aufgerufen, wird nur **false** zurück gegeben.
- Wird im Zustand *Aktiviert* **activateComponent()** aufgerufen, wird ein **false** zurück gegeben, die Komponenten bleibt im Zustand *Aktiviert*.
- Wird im Zustand *Aktiviert* **deactivateComponent()** aufgerufen, gibt die Methode ein **true** zurück und geht in den Zustand *Deaktiviert*.
- Wird im Zustand *Deaktiviert* **deactivateComponent()** aufgerufen, wird ein **false** zurück gegeben, die Komponenten bleibt im Zustand *Deaktiviert*.
- Wird im Zustand *Deaktiviert* die Methode **getCompType()** aufgerufen, liefert sie den **CompType ADMIN** als Ergebnis.
- Wird im Zustand *Aktiviert* die Methode **getCompType()** aufgerufen, liefert sie den **CompType ADMIN** als Ergebnis.

- Wird im Zustand *Deaktiviert* die Methode `isActivated()` aufgerufen, wird ein `false` zurück gegeben.
- Wird im Zustand *Aktiviert* die Methode `isActivated()` aufgerufen, wird ein `true` zurück gegeben.

Schnittstellen

Die Schnittstellen-Klassen `IManagerSteuerung`, `IKontoSteuerung`, `IAntragSteuerung` und `ISachbearbeiterSteuerung` werden von der Komponente `SteuerungAPI` bereitgestellt.

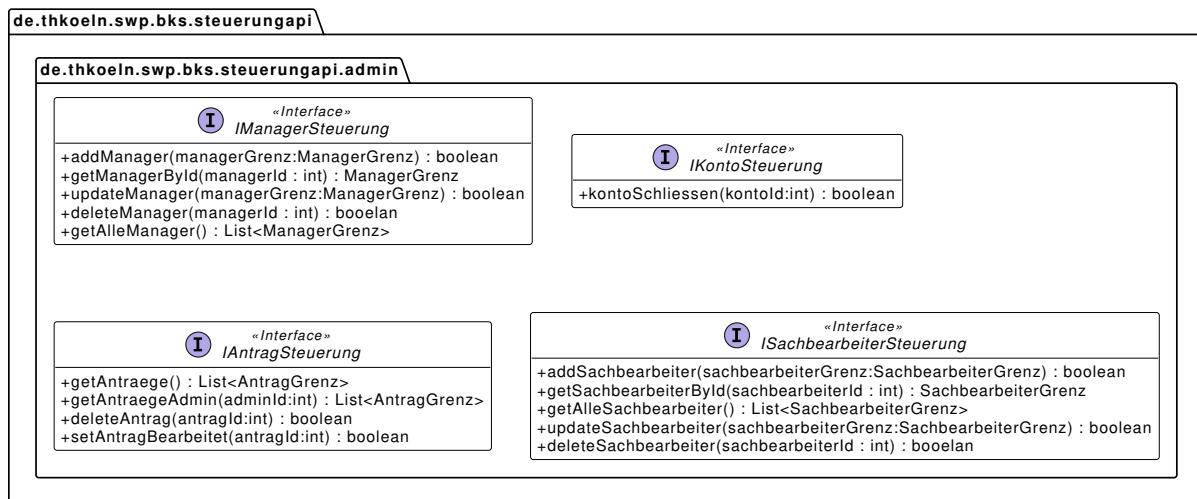


Abbildung 5. Schnittstellen zur Verwendung in der `AdminSteuerung`-Komponente

3.8. AdminDaten

3.8.1. Funktionen

Diese Komponente realisiert einige Schnittstellen für den Zugriff auf die Entitätsklassen.

3.8.2. Schnittstellen



Es existiert keine `ICRUDAdmin`-Schnittstelle, weil Admins direkt in der Datenbank angelegt, verändert und gelöscht werden!

IAntragAdmin

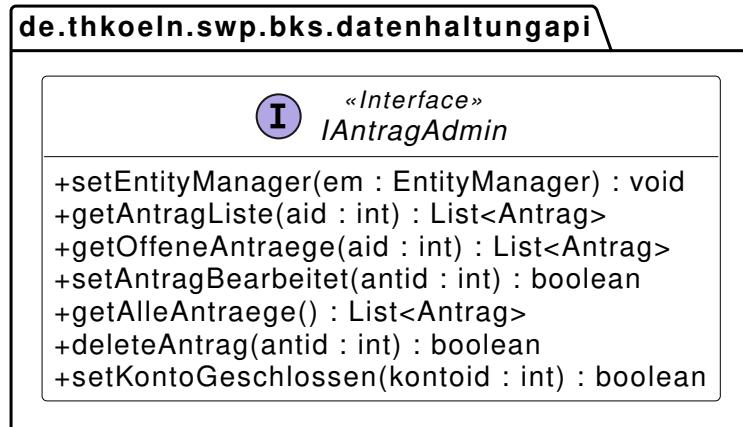


Abbildung 6. `IAntragAdmin`-Schnittstelle zur Verwendung in der `AdminDaten`-Komponente

Spezifikation der Methoden:

`getAntragListe`

liefert alle an den Admin gerichteten Anträge.

`getOffeneAntraege`

liefert alle an den Admin gerichteten Anträge mit Status "genehmigt".

`setAntragBearbeitet`

setzt den Status der vom Admin zu bearbeitenden Antragstypen auf "bearbeitet".

`getAlleAntraege`

liefert alle in der DB vorhandenen Anträge.

`deleteAntrag`

löscht einen Antrag aus der DB, falls der Antrag den Status "storniert" oder "bearbeitet" besitzt.

`setKontoGeschlossen`

setzt den Status des angegebenen Kontos auf "geschlossen".

Die detaillierte Spezifikation des erwarteten Verhaltens der Methoden dieser Schnittstelle sind in der Testspezifikation gegeben.

`IAntragStellenAdmin`

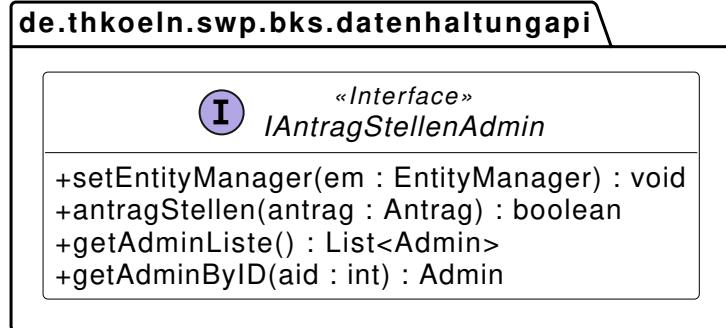


Abbildung 7. `IAntragStellenAdmin`-Schnittstelle zur Verwendung in der `AdminDaten`-Komponente

Spezifikation der Methoden:

`antragStellen`

wird vom Manager zum Stellen seiner Anträge verwendet.

`getAdminListe`

liefert die Liste aller in der DB existierenden Admins.

`getAdminByID`

liefert den Admin mit der angegebenen Id.

Die detaillierte Spezifikation des erwarteten Verhaltens der Methoden dieser Schnittstelle sind in der Testspezifikation gegeben.

3.9. KundegUI

3.9.1. Funktionen

Die `KundegUI`-Komponente stellt die Benutzeroberfläche für alle Anwendungsfälle des Kunden zur Verfügung.

Sie soll mittels JavaFX realisiert werden.

Die `KundeApp`-Klasse und `KundeController`-Klasse sowie eine `kunde.fxml`-Datei werden wie folgt vorgegeben:

- im `src/main/java`-Ordner:
 - `de.thkoeln.swp.bks.kundegui.application.KundeApp`
 - `de.thkoeln.swp.bks.kundegui.application.KundeController`
- im `src/main/resources`-Ordner:
 - `kunde.fxml`

Der weitere Feinentwurf dieser Komponente ist nicht vorgegeben und soll von dem `KundeKonto`-Team eigenständig erstellt werden.

3.9.2. Schnittstellen

Diese Komponente bietet keine Schnittstellen für andere Komponenten an.

3.10. KundeSteuerung

3.10.1. Funktionen

Diese Komponente realisiert die folgenden Anwendungsfälle: /LF210/, /LF220/, /LF225/, /LF230/, /LF240/, /LF250/

Zur Realisierung der Anwendungsfälle darf nicht direkt auf die Datenbank zugegriffen werden. Es dürfen nur die Schnittstellen verwendet werden, für die in [Abbildung 2](#) der Zugriff definiert wurde.

3.10.2. Schnittstellen

IActivateComponent

Die KundeSteuerung-Komponente implementiert die **IActivateComponent** - Schnittstelle (und exportiert diese Implementierung), die von der **Bootloader**-Komponente verwendet wird, um die Komponente zu aktivieren und zu deaktivieren. Die Implementierung muss dafür im Paket `de.thkoeln.swp.bks.kundesteuerung.impl` mit dem Namen `IActivateComponentImpl` liegen, da diese sonst vom Bootloader nicht gefunden werden kann.

Die Schnittstellen-Klasse **IActivateComponent** wird von der Komponente **ComponentController** bereitgestellt.

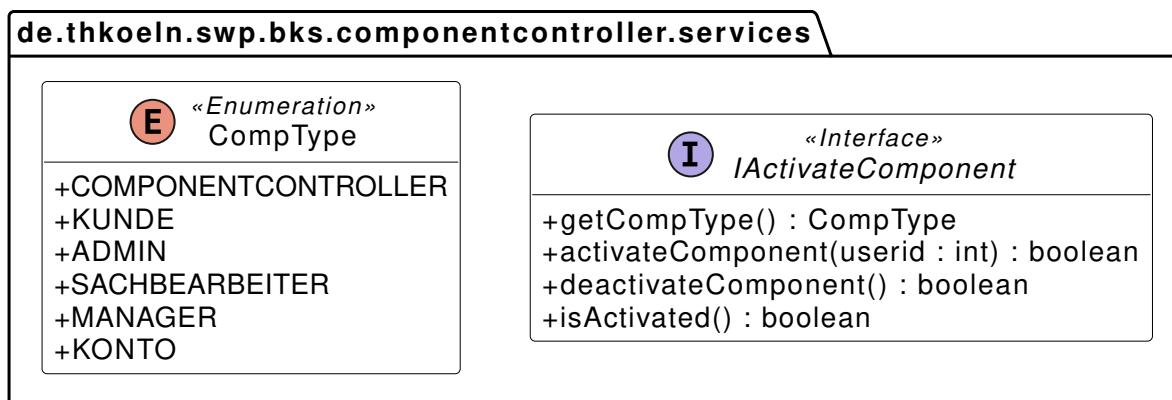


Abbildung 8. IActivateComponent-Schnittstelle zur Verwendung in der KundeSteuerung-Komponente

Der Parameter der `activateComponent` - Funktion der **IActivateComponent** - Schnittstelle enthält die jeweilige ID des Kunden, für den die Komponente aktiviert werden soll. Falls eine falsche ID übergeben wird, sollte die Komponente nicht aktiviert werden und `false` zurückliefern. Zur Aktivierung der Komponente für einen Beispiel-Kunden mit `userid=11` und `name=Max Musterman` wird die `activateComponent`-Methode mit `11` aufgerufen. Die Komponente sollte nur dann aktiviert werden, wenn dieser Kunde in der Datenbank

bereits existiert.

Die Komponente kann die beiden Zustände *Aktiviert* und *Deaktiviert* annehmen. Die Methoden der Schnittstelle **IActivateComponent** können zu Zustandsübergängen führen:

- Wird die Komponente gestartet, so befindet sie sich zunächst im Zustand *Deaktiviert*.
- Wird im Zustand *Deaktiviert* `activateComponent()` mit zulässiger ID aufgerufen, gibt die Methode ein `true` zurück und geht in den Zustand *Aktiviert*.
- Wird `activateComponent()` mit einer nicht zulässigen ID aufgerufen, wird nur `false` zurück gegeben.
- Wird im Zustand *Aktiviert* `activateComponent()` aufgerufen, wird ein `false` zurück gegeben, die Komponenten bleibt im Zustand *Aktiviert*.
- Wird im Zustand *Aktiviert* `deactivateComponent()` aufgerufen, gibt die Methode ein `true` zurück und geht in den Zustand *Deaktiviert*.
- Wird im Zustand *Deaktiviert* `deactivateComponent()` aufgerufen, wird ein `false` zurück gegeben, die Komponenten bleibt im Zustand *Deaktiviert*.
- Wird im Zustand *Deaktiviert* die Methode `getCompType()` aufgerufen, liefert sie den **CompType KUNDE** als Ergebnis.
- Wird im Zustand *Aktiviert* die Methode `getCompType()` aufgerufen, liefert sie den **CompType KUNDE** als Ergebnis.
- Wird im Zustand *Deaktiviert* die Methode `isActivated()` aufgerufen, wird ein `false` zurück gegeben.
- Wird im Zustand *Aktiviert* die Methode `isActivated()` aufgerufen, wird ein `true` zurück gegeben.

Schnittstellen

Die Schnittstellen-Klassen **IKundeSteuerung**, **IKundeKontoSteuerung** und **IIUeberweisungSteuerung** werden von der Komponente **SteuerungAPI** bereitgestellt.

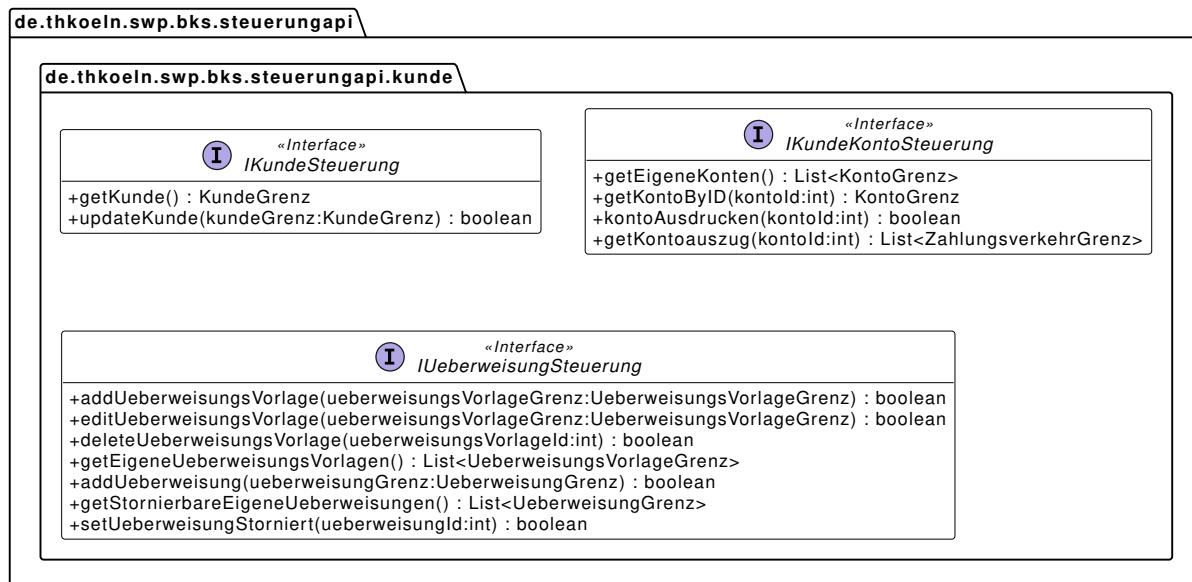


Abbildung 9. Schnittstellen zur Verwendung in der **KundeSteuerung-Komponente**

3.11. KundeDaten

3.11.1. Funktionen

Diese Komponente realisiert einige Schnittstellen für den Zugriff auf die Entitätsklassen.

3.11.2. Schnittstellen

ICRUDKunde



Abbildung 10. **ICRUDKunde**-Schnittstelle zur Verwendung in der **KundeDaten**-Komponente

Spezifikation der Methoden:

getKundeByID

liefert den Kunden mit der angegebenen Id.

getKundenListe

liefert alle in der DB existierenden Kunden.

insertKunde

Fügt einen neuen Kunden zur DB hinzu.

editKunde

modifiziert einen existierenden Kunden.

deleteKunde

löscht einen existierenden Kunden.

Die detaillierte Spezifikation des erwarteten Verhaltens der Methoden dieser Schnittstelle sind in der Testspezifikation gegeben.

3.12. KontoGUI

3.12.1. Funktionen

Die **KontoGUI**-Komponente stellt die Benutzeroberfläche für alle Anwendungsfälle des Überweisungsbearbeiters zur Verfügung.

Sie soll mittels JavaFX realisiert werden.

Die **KontoApp**-Klasse und **KontoController**-Klasse sowie eine **konto.fxml**-Datei werden wie folgt vorgegeben:

- im **src/main/java**-Ordner:
 - **de.thkoeln.swp.bks.kontogui.application.KontoApp**
 - **de.thkoeln.swp.bks.kontogui.application.KontoController**
- im **src/main/resources**-Ordner:
 - **konto.fxml**

Der weitere Feinentwurf dieser Komponente ist nicht vorgegeben und soll von dem **KundeKonto**-Team eigenständig erstellt werden.

3.12.2. Schnittstellen

Diese Komponente bietet keine Schnittstellen für andere Komponenten an.

3.13. KontoSteuerung

3.13.1. Funktionen

Diese Komponente realisiert die folgenden Anwendungsfälle: /LF260/, /LF270/

Zur Realisierung der Anwendungsfälle darf nicht direkt auf die Datenbank zugegriffen werden. Es dürfen nur die Schnittstellen verwendet werden, für die in [Abbildung 2](#) der Zugriff definiert wurde.

3.13.2. Schnittstellen

IActivateComponent

Die **KontoSteuerung**-Komponente implementiert die **IActivateComponent** - Schnittstelle (und exportiert diese Implementierung), die von der **Bootloader**-Komponente verwendet wird, um die Komponente zu aktivieren und zu deaktivieren. Die Implementierung muss dafür im Paket `de.thkoeln.swp.bks.kontosteuierung.impl` mit dem Namen `IActivateComponentImpl` liegen, da diese sonst vom Bootloader nicht gefunden werden kann.

Die Schnittstellen-Klasse **IActivateComponent** wird von der Komponente **ComponentController** bereitgestellt.

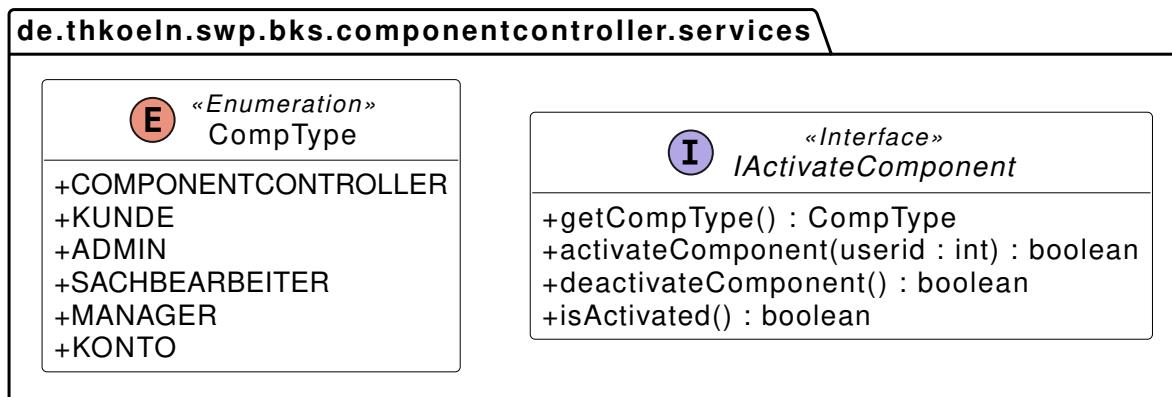


Abbildung 11. IActivateComponent-Schnittstelle zur Verwendung in der KontoSteuerung-Komponente

Der Parameter der `activateComponent` - Funktion der **IActivateComponent** - Schnittstelle enthält die ID des "Überweisungsbearbeiters", für den die Komponente aktiviert werden soll. Falls eine falsche ID übergeben wird, sollte die Komponente nicht aktiviert werden und `false` zurückliefern. Die Komponente darf nur für den Default-"Überweisungsbearbeiter (`userid = 23`) aktiviert werden.

Die Komponente kann die beiden Zustände *Aktiviert* und *Deaktiviert* annehmen. Die Methoden der Schnittstelle **IActivateComponent** können zu Zustandsübergängen führen:

- Wird die Komponente gestartet, so befindet sie sich zunächst im Zustand *Deaktiviert*.
- Wird im Zustand *Deaktiviert* `activateComponent()` mit zulässiger ID aufgerufen, gibt die Methode ein `true` zurück und geht in den Zustand *Aktiviert*.

- Wird `activateComponent()` mit einer nicht zulässigen ID aufgerufen, wird nur `false` zurück gegeben.
- Wird im Zustand *Aktiviert* `activateComponent()` aufgerufen, wird ein `false` zurück gegeben, die Komponenten bleibt im Zustand *Aktiviert*.
- Wird im Zustand *Aktiviert* `deactivateComponent()` aufgerufen, gibt die Methode ein `true` zurück und geht in den Zustand *Deaktiviert*.
- Wird im Zustand *Deaktiviert* `deactivateComponent()` aufgerufen, wird ein `false` zurück gegeben, die Komponenten bleibt im Zustand *Deaktiviert*.
- Wird im Zustand *Deaktiviert* die Methode `getCompType()` aufgerufen, liefert sie den `CompType KONTO` als Ergebnis.
- Wird im Zustand *Aktiviert* die Methode `getCompType()` aufgerufen, liefert sie den `CompType KONTO` als Ergebnis.
- Wird im Zustand *Deaktiviert* die Methode `isActivated()` aufgerufen, wird ein `false` zurück gegeben.
- Wird im Zustand *Aktiviert* die Methode `isActivated()` aufgerufen, wird ein `true` zurück gegeben.

Schnittstellen

Die Schnittstellen-Klasse `IKontoSteuerung` wird von der Komponente `SteuerungAPI` bereitgestellt.

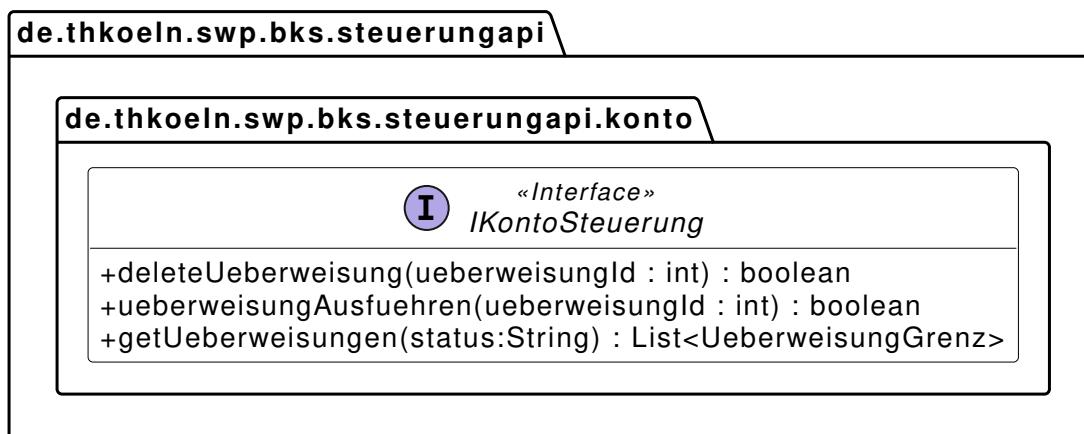


Abbildung 12. Schnittstellen zur Verwendung in der `KontoSteuerung`-Komponente

3.14. KontoDaten

3.14.1. Funktionen

Diese Komponente realisiert einige Schnittstellen für den Zugriff auf die Entitätsklassen.

3.14.2. Schnittstellen

ICRUDKonto

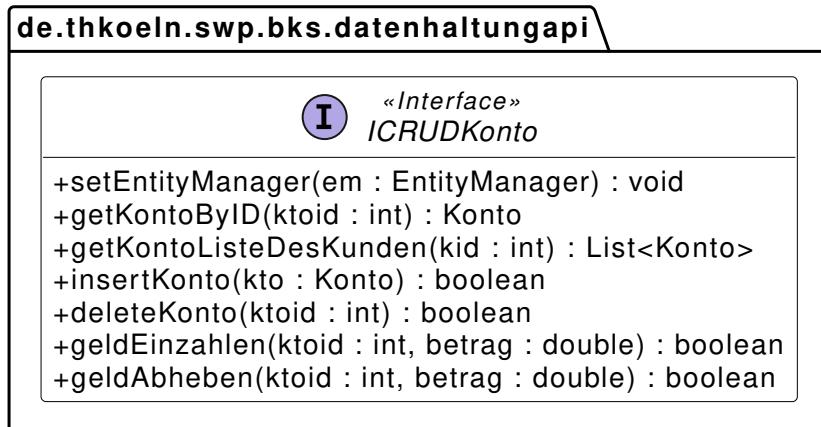


Abbildung 13. **ICRUDKonto**-Schnittstelle zur Verwendung in der **KontoDaten**-Komponente

Spezifikation der Methoden:

getKontoByID

liefert das Konto mit der angegebenen Id.

getKontoListeDesKunden

liefert alle Konten des angegebenen Kunden.

insertKonto

fügt ein neues Konto ein.

deleteKonto

löscht das angegebene Konto.

geldEinzahlen

soll jederzeit möglich sein. Zur Dokumentation der Einzahlung muss ein entsprechender **Zahlungsverkehr** angelegt werden. In das Attribut **erlaeuterung** dieses **Zahlungsverkehr**-Objekts soll "Einzahlung" eingetragen werden. Nach dem erfolgreichen Einzahlen muss der korrekte neue Status des Kontos (haben, soll oder gesperrt) in Abhängigkeit vom aktuellen Kontostand und der Höhe des Dispos gesetzt werden. Siehe hierzu das Zustandsdiagramm zu den Zuständen des Kontos in der Systemspzifikation.

geldAbheben

erlaubt nur dann Abhebungen, wenn das Konto nicht gesperrt ist. Ist ein Konto gesperrt, so liefert die **geldAbheben**-Methode die Rückgabe **false**. Zur Dokumentation der Abhebung muss ein entsprechender **Zahlungsverkehr** angelegt werden. In das Attribut **erlaeuterung** dieses **Zahlungsverkehr**-Objekts soll "Auszahlung" eingetragen werden. Nach dem erfolgreichen Abheben muss der korrekte neue Status des Kontos (haben, soll oder gesperrt) in Abhängigkeit vom aktuellen Kontostand und der Höhe des Dispos

gesetzt werden. Siehe hierzu das Zustandsdiagramm zu den Zuständen des Kontos in der Systemspzifikation.

Die detaillierte Spezifikation des erwarteten Verhaltens der Methoden dieser Schnittstelle sind in der Testspezifikation gegeben.

IKontoService

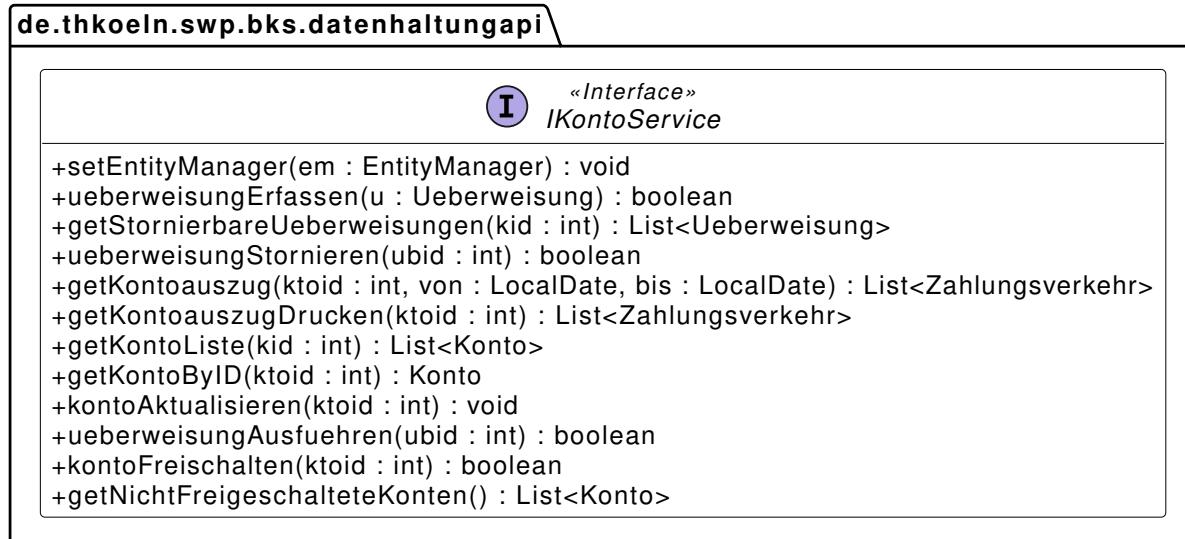


Abbildung 14. IKontoService-Schnittstelle zur Verwendung in der KontoDaten-Komponente

Spezifikation der Methoden:

ueberweisungErfassen

fügt eine neue Überweisung zur DB hinzu.

getStornierbareUeberweisungen

liefert alle stornierbaren Überweisungen des angegebenen Kunden.

ueberweisungStornieren

storniert eine stornierbare Überweisung.

getKontoauszug

liefert alle Zahlungsverkehre des gegebenen Kontos innerhalb des gegebenen Datumsintervalls-sowohl bereits ausgedruckte, wie auch noch nicht ausgedruckte.

getKontoauszugDrucken

liefert alle Zahlungsverkehre des gegebenen Kontos, die noch nicht als ausgedruckt markiert sind-und markiert diese dann anschließend als ausgedruckt.

getKontoListe

liefert alle Konten des angegebenen Kunden.

getKontoByID

liefert das Konto mit der angegebenen Id.

kontoAktualisieren

setzt den Status eines gegebenen Kontos auf den korrekten Wert (haben, soll oder gesperrt) in Abhängigkeit vom aktuellen Kontostand und der Höhe des Dispos. Siehe hierzu das Zustandsdiagramm zu den Zuständen des Kontos in der Systemspzifikation.

ueberweisungAusfuehren

verändert die beiden betroffenen Konten entsprechend der übergebenen Überweisung. Zu beachten ist, dass die Überweisung nicht ausgeführt werden darf (und anschließend *false* als Rückgabe besitzt), falls das zu belastende Konto bereits gesperrt ist. In diesem Fall ist der Zustand der Überweisung auf *nicht überweisbar* zu setzen, ansonsten auf den Zustand *überwiesen*. Zur Dokumentation einer erfolgreichen Überweisung müssen zwei **Zahlungsverkehre** angelegt werden: eine für das zu belastende Konto (= **vonkonto** in der Überweisung) und eine für das empfangende Konto (= **zukonto** in der Überweisung). In das Attribut **erlaeuterung** dieser beiden **Zahlungsverkehr**-Objekte soll jeweils der **verwendungszweck** der Überweisung eingetragen werden. Zum abschließenden korrekten Setzen des Status der betroffenen Konten nach einer erfolgreichen Überweisung, soll für beide Konten die Methode **kontoAktualisieren** der Schnittstelle **IKontoService** aufgerufen werden.

kontoFreischalten

schaltet ein angelegtes Konto frei.

getNichtFreigeschalteteKonten

liefert alle Konten aus der DB, die freigeschaltet werden können.

Die detaillierte Spezifikation des erwarteten Verhaltens der Methoden dieser Schnittstelle sind in der Testspezifikation gegeben.

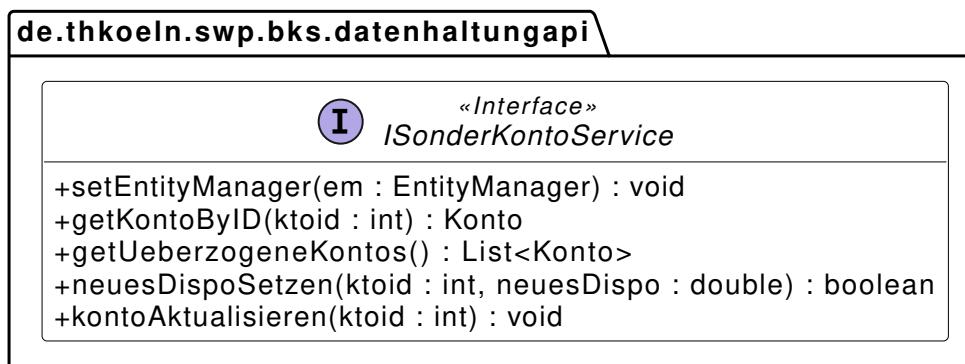
ISonderKontoService

Abbildung 15. **ISonderKontoService**-Schnittstelle zur Verwendung in der **KontoDaten**-Komponente

Spezifikation der Methoden:

getKontoByID

liefert das Konto mit der angegebenen Id.

getUeberzogeneKontos

liefert alle Konten aus der DB, die überzogen sind.

neuesDispoSetzen

setzt den Dispos eines Kontos auf den neuen Wert. Zur Aktualisierung des Status des Kontos soll **neuesDispoSetzen** die Methode **kontoAktualisieren** aufgerufen werden.

kontoAktualisieren

setzt den Status eines gegebenen Kontos auf den korrekten Wert (haben, soll oder gesperrt) in Abhängigkeit vom aktuellen Kontostand und der Höhe des Dispos. Siehe hierzu das Zustandsdiagramm zu den Zuständen des Kontos in der Systemspzifikation.

Die detaillierte Spezifikation des erwarteten Verhaltens der Methoden dieser Schnittstelle sind in der Testspezifikation gegeben.

3.15. ManagerGUI

3.15.1. Funktionen

Die **ManagerGUI**-Komponente stellt die Benutzeroberfläche für alle Anwendungsfälle des Managers zur Verfügung.

Sie soll mittels JavaFX realisiert werden.

Die **ManagerApp**-Klasse und **ManagerController**-Klasse sowie eine **manager.fxml**-Datei werden wie folgt vorgegeben:

- im **src/main/java**-Ordner:
 - **de.thkoeln.swp.bks.managergui.application.ManagerApp**
 - **de.thkoeln.swp.bks.managergui.application.ManagerController**
- im **src/main/resources**-Ordner:
 - **manager.fxml**

Der weitere Feinentwurf dieser Komponente ist nicht vorgegeben und soll von dem **Manager**-Team eigenständig erstellt werden.

3.15.2. Schnittstellen

Diese Komponente bietet keine Schnittstellen für andere Komponenten an.

3.16. ManagerSteuerung

3.16.1. Funktionen

Diese Komponente realisiert die folgenden Anwendungsfälle: /LF80/, /LF85/, /LF90/, /LF105/, /LF110/, /LF113/, /LF116/

Zur Realisierung der Anwendungsfälle darf nicht direkt auf die Datenbank zugegriffen werden. Es dürfen nur die Schnittstellen verwendet werden, für die in [Abbildung 2](#) der Zugriff definiert wurde.

3.16.2. Schnittstellen

IActivateComponent

Die **ManagerSteuerung**-Komponente implementiert die **IActivateComponent** - Schnittstelle (und exportiert diese Implementierung), die von der **Bootloader**-Komponente verwendet wird, um die Komponente zu aktivieren und zu deaktivieren. Die Implementierung muss dafür im Paket `de.thkoeln.swp.bks.managersteuerung.impl` mit dem Namen **IActivateComponentImpl** liegen, da diese sonst vom Bootloader nicht gefunden werden kann.

Die Schnittstellen-Klasse **IActivateComponent** wird von der Komponente **ComponentController** bereitgestellt.

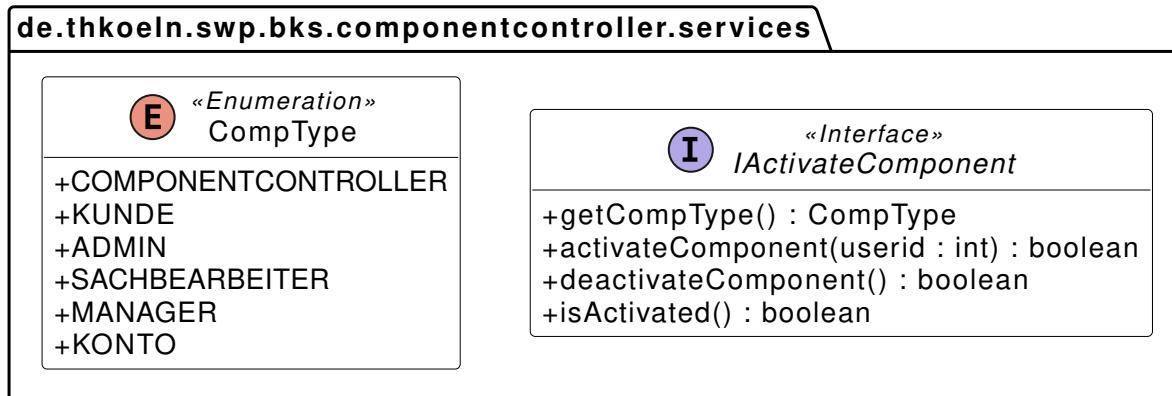


Abbildung 16. IActivateComponent-Schnittstelle zur Verwendung in der ManagerSteuerung-Komponente

Der Parameter der `activateComponent` - Funktion der **IActivateComponent** - Schnittstelle enthält die jeweilige ID des Managers, für den die Komponente aktiviert werden soll. Falls eine falsche ID übergeben wird, sollte die Komponente nicht aktiviert werden und `false` zurückliefern. Zur Aktivierung der Komponente für einen Beispiel-Manager mit `userid=4` und `name=Max Musterman` wird die `activateComponent`-Methode mit `4` aufgerufen. Die Komponente sollte nur dann aktiviert werden, wenn dieser Manager in der Datenbank bereits existiert.

Die Komponente kann die beiden Zustände *Aktiviert* und *Deaktiviert* annehmen. Die Methoden der Schnittstelle **IActivateComponent** können zu Zustandsübergängen führen:

- Wird die Komponente gestartet, so befindet sie sich zunächst im Zustand *Deaktiviert*.
- Wird im Zustand *Deaktiviert* **activateComponent()** mit zulässiger ID aufgerufen, gibt die Methode ein **true** zurück und geht in den Zustand *Aktiviert*.
- Wird **activateComponent()** mit einer nicht zulässigen ID aufgerufen, wird nur **false** zurück gegeben.
- Wird im Zustand *Aktiviert* **activateComponent()** aufgerufen, wird ein **false** zurück gegeben, die Komponenten bleibt im Zustand *Aktiviert*.
- Wird im Zustand *Aktiviert* **deactivateComponent()** aufgerufen, gibt die Methode ein **true** zurück und geht in den Zustand *Deaktiviert*.
- Wird im Zustand *Deaktiviert* **deactivateComponent()** aufgerufen, wird ein **false** zurück gegeben, die Komponenten bleibt im Zustand *Deaktiviert*.
- Wird im Zustand *Deaktiviert* die Methode **getCompType()** aufgerufen, liefert sie den **CompType MANAGER** als Ergebnis.
- Wird im Zustand *Aktiviert* die Methode **getCompType()** aufgerufen, liefert sie den **CompType MANAGER** als Ergebnis.
- Wird im Zustand *Deaktiviert* die Methode **isActivated()** aufgerufen, wird ein **false** zurück gegeben.
- Wird im Zustand *Aktiviert* die Methode **isActivated()** aufgerufen, wird ein **true** zurück gegeben.

Schnittstellen

Die Schnittstellen-Klassen **IKontoManagerSteuerung** und **IAntragManagerSteuerung** werden von der Komponente **SteuerungAPI** bereitgestellt.

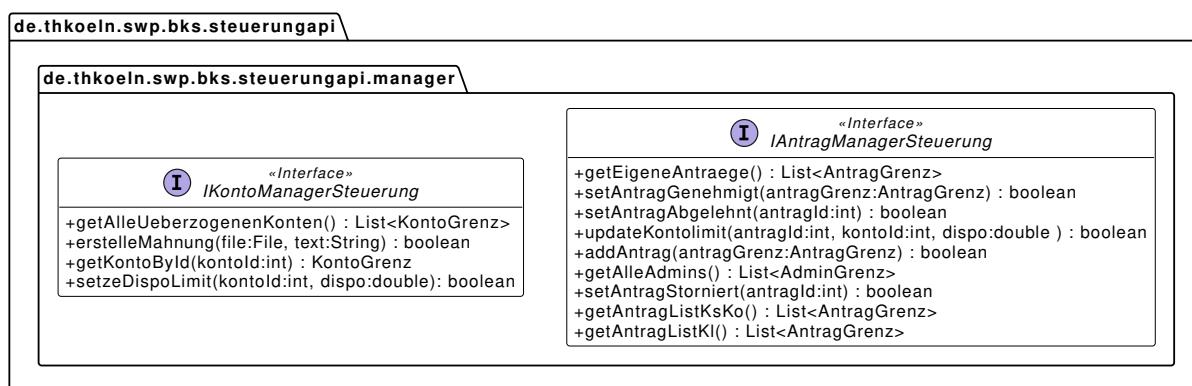


Abbildung 17. Schnittstellen zur Verwendung in der **ManagerSteuerung**-Komponente

3.17. ManagerDaten

3.17.1. Funktionen

Diese Komponente realisiert einige Schnittstellen für den Zugriff auf die Entitätsklassen.

3.17.2. Schnittstellen

ICRUDManager

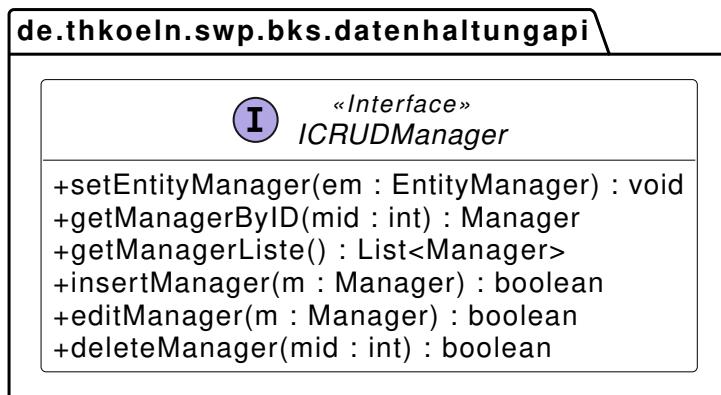


Abbildung 18. `ICRUDManager`-Schnittstelle zur Verwendung in der `ManagerDaten`-Komponente

Grobe Spezifikation der Methoden:

`getManagerByID`

liefert den Manager mit der angegebenen Id.

`getManagerListe`

liefert alle in der DB existierenden Manager.

`insertManager`

fügt einen neuen Manager zur DB hinzu.

`editManager`

modifiziert einen existierenden Manager.

`deleteManager`

löscht einen existierenden Manager.

Die detaillierte Spezifikation des erwarteten Verhaltens der Methoden dieser Schnittstelle sind in der Testspezifikation gegeben.

IAntragManager



Abbildung 19. `IAntragManager`-Schnittstelle zur Verwendung in der `ManagerDaten`-Komponente

Grobe Spezifikation der Methoden:

`getAntragListe`

liefert alle Anträge, die an den Manager `mid` gerichtet sind.

`getListeEigenerAntraege`

liefert alle Anträge, die der Manager `mid` erstellt hat.

`getSachbearbeiterListe`

liefert die Liste aller existierender Sachbearbeiter.

`setAntragGenehmigt`

dient zum Genehmigen eines vom Manager zu genehmigenden Antrags (außer Kontoschließungs-Antrag).

`setKSAntragGenehmigt`

dient zum Genehmigen eines vom Manager zu genehmigenden Kontoschließungs-Antrags und dem gleichzeitigen Festlegen des ausführenden Admins.

`setAntragBearbeitet`

setzt den Status eines vom Manager bearbeiteten Antrags auf "bearbeitet".

`setAntragAbgelehnt`

dient zum Ablehnen eines vom Manager zu genehmigenden Antrags.

`setAntragStorniert`

dient zum Stornieren eines vom Manager `mid` erstellten Antrags.

Die detaillierte Spezifikation des erwarteten Verhaltens der Methoden dieser Schnittstelle sind in der Testspezifikation gegeben.

IAntragStellenManager

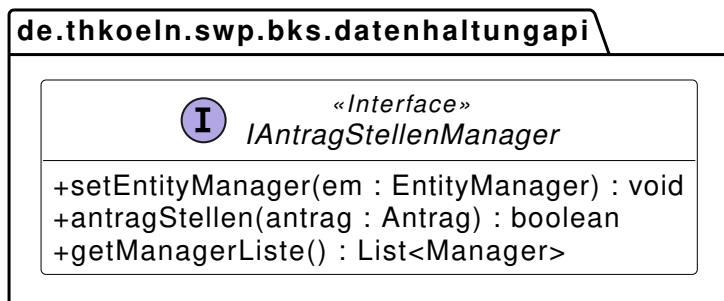


Abbildung 20. `IAntragStellenManager`-Schnittstelle zur Verwendung in der `ManagerDaten`-Komponente

Grobe Spezifikation der Methoden:

`antragStellen`

wird vom Sachbearbeiter zum Stellen eines neuen Antrags verwendet.

`getManagerListe`

liefert alle in der DB existierenden Manager.

Die detaillierte Spezifikation des erwarteten Verhaltens der Methoden dieser Schnittstelle sind in der Testspezifikation gegeben.

3.18. SachbearbeiterGUI

3.18.1. Funktionen

Die `SachbearbeiterGUI`-Komponente stellt die Benutzeroberfläche für alle Anwendungsfälle des Sachbearbeiters zur Verfügung.

Sie soll mittels JavaFX realisiert werden.

Die `SachbearbeiterApp`-Klasse und `SachbearbeiterController`-Klasse sowie eine `sachbearbeiter.fxml`-Datei werden wie folgt vorgegeben:

- im `src/main/java`-Ordner:
 - `de.thkoeln.swp.bks.sachbearbeitergui.application.SachbearbeiterApp`
 - `de.thkoeln.swp.bks.mansachbearbeitergui.application.SachbearbeiterController`
- im `src/main/resources`-Ordner:
 - `sachbearbeiter.fxml`

Der weitere Feinentwurf dieser Komponente ist nicht vorgegeben und soll von dem `Sachbearbeiter`-Team eigenständig erstellt werden.

3.18.2. Schnittstellen

Diese Komponente bietet keine Schnittstellen für andere Komponenten an.

3.19. SachbearbeiterSteuerung

3.19.1. Funktionen

Diese Komponente realisiert die folgenden Anwendungsfälle: /LF120/, /LF130/, /LF140/, /LF143/, /LF145/, /LF150/, /LF170/, /LF180/, /LF190/, /LF200/

Zur Realisierung der Anwendungsfälle darf nicht direkt auf die Datenbank zugegriffen werden. Es dürfen nur die Schnittstellen verwendet werden, für die in [Abbildung 2](#) der Zugriff definiert wurde.

3.19.2. Schnittstellen

IActivateComponent

Die **SachbearbeiterSteuerung** - Komponente implementiert die **IActivateComponent** - Schnittstelle (und exportiert diese Implementierung), die von der **Bootloader**-Komponente verwendet wird, um die Komponente zu aktivieren und zu deaktivieren. Die Implementierung muss dafür im Paket **de.thkoeln.swp.bks.sachbearbeitersteuerung.impl** mit dem Namen **IActivateComponentImpl** liegen, da diese sonst vom Bootloader nicht gefunden werden kann.

Die Schnittstellen-Klasse **IActivateComponent** wird von der Komponente **ComponentController** bereitgestellt.

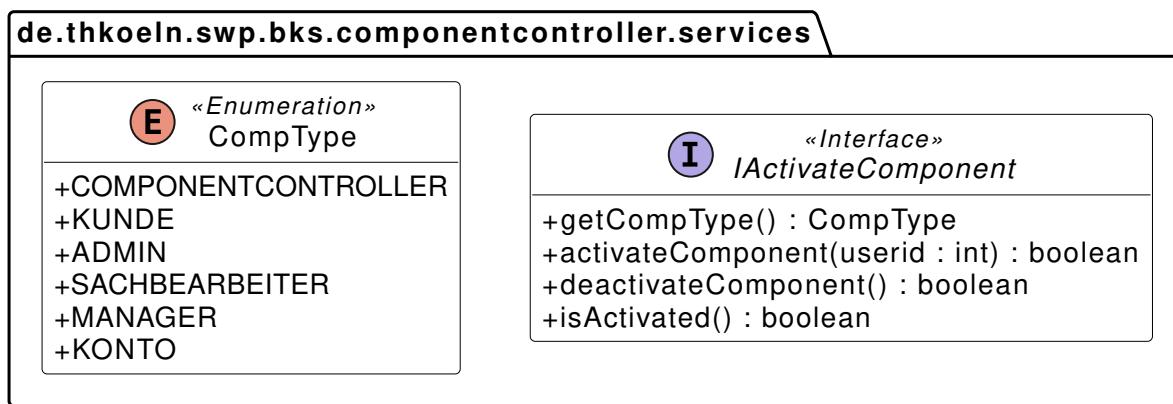


Abbildung 21. IActivateComponent-Schnittstelle zur Verwendung in der SachbearbeiterSteuerung -Komponente

Der Parameter der `activateComponent` - Funktion der **IActivateComponent** - Schnittstelle enthält die jeweilige ID des Sachbearbeiters, für den die Komponente aktiviert werden soll. Falls eine falsche ID übergeben wird, sollte die Komponente nicht aktiviert werden und `false` zurückliefern. Zur Aktivierung der **Sachbearbeiter** - Komponente für einen

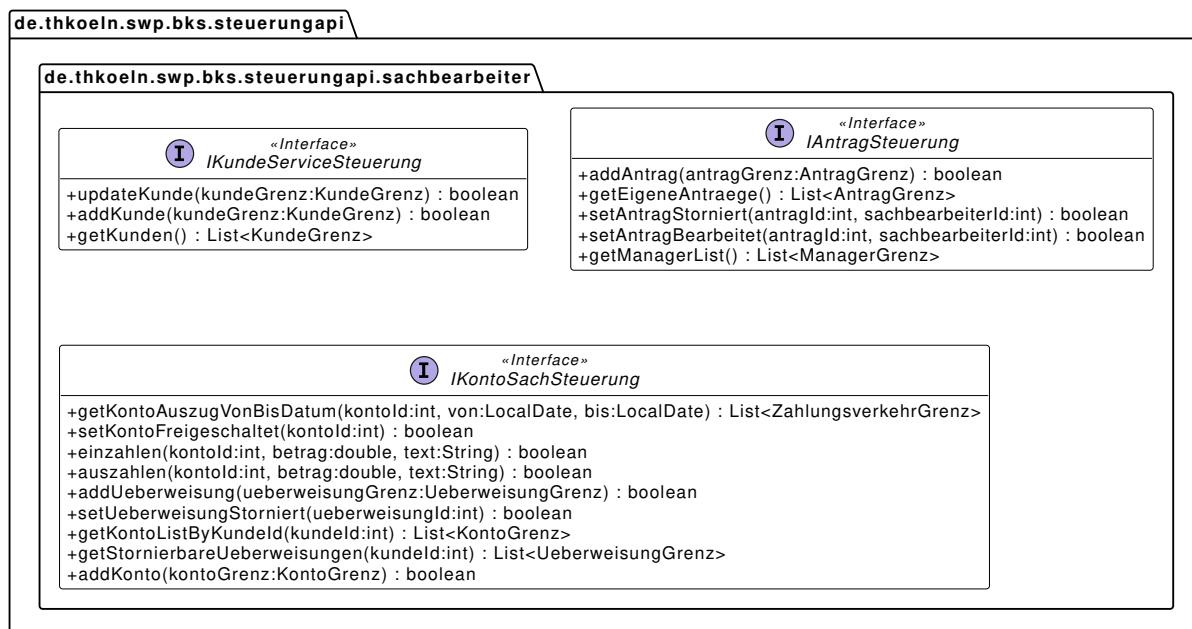
Beispiel-Sachbearbeiter mit `userid=5` und `name=Max Musterman` wird die `activateComponent`-Methode mit `5` aufgerufen. Die `Sachbearbeiter` - Komponente sollte nur dann aktiviert werden, wenn dieser Sachbearbeiter in der Datenbank bereits existiert.

Die Komponente kann die beiden Zustände *Aktiviert* und *Deaktiviert* annehmen. Die Methoden der Schnittstelle `IActivateComponent` können zu Zustandsübergängen führen:

- Wird die Komponente gestartet, so befindet sie sich zunächst im Zustand *Deaktiviert*.
- Wird im Zustand *Deaktiviert* `activateComponent()` mit zulässiger ID aufgerufen, gibt die Methode ein `true` zurück und geht in den Zustand *Aktiviert*.
- Wird `activateComponent()` mit einer nicht zulässigen ID aufgerufen, wird nur `false` zurück gegeben.
- Wird im Zustand *Aktiviert* `activateComponent()` aufgerufen, wird ein `false` zurück gegeben, die Komponenten bleibt im Zustand *Aktiviert*.
- Wird im Zustand *Aktiviert* `deactivateComponent()` aufgerufen, gibt die Methode ein `true` zurück und geht in den Zustand *Deaktiviert*.
- Wird im Zustand *Deaktiviert* `deactivateComponent()` aufgerufen, wird ein `false` zurück gegeben, die Komponenten bleibt im Zustand *Deaktiviert*.
- Wird im Zustand *Deaktiviert* die Methode `getCompType()` aufgerufen, liefert sie den `CompType SACHBEARBEITER` als Ergebnis.
- Wird im Zustand *Aktiviert* die Methode `getCompType()` aufgerufen, liefert sie den `CompType SACHBEARBEITER` als Ergebnis.
- Wird im Zustand *Deaktiviert* die Methode `isActivated()` aufgerufen, wird ein `false` zurück gegeben.
- Wird im Zustand *Aktiviert* die Methode `isActivated()` aufgerufen, wird ein `true` zurück gegeben.

Schnittstellen

Die Schnittstellen-Klassen `IKundeServiceSteuerung`, `IAntragSteuerung` und `IKontoSachSteuerung` werden von der Komponente `SteuerungAPI` bereitgestellt.

Abbildung 22. Schnittstellen zur Verwendung in der **SachbearbeiterSteuerung**-Komponente

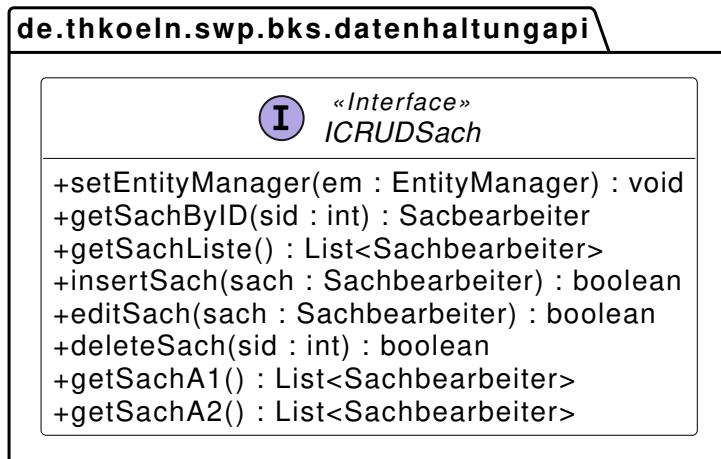
3.20. SachbearbeiterDaten

3.20.1. Funktionen

Diese Komponente realisiert einige Schnittstellen für den Zugriff auf die Entitätsklassen.

3.20.2. Schnittstellen

ICRUDSach

Abbildung 23. **ICRUDSach**-Schnittstelle zur Verwendung in der **SachbearbeiterDaten**-Komponente

Grobe Spezifikation der Methoden:

getSachByID

liefert den Sachbearbeiter mit der angegebenen Id.

getSachListe

liefert alle in der DB existierenden Sachbearbeiter.

insertSach

fügt einen neuen Sachbearbeiter zur DB hinzu.

editSach

modifiziert einen existierenden Sachbearbeiter.

deleteSach

löscht einen existierenden Sachbearbeiter.

getSachA1

liefert alle Sachbearbeiter, die der Abteilung A1 zugeordnet sind.

getSachA2

liefert alle Sachbearbeiter, die der Abteilung A2 zugeordnet sind.

Die detaillierte Spezifikation des erwarteten Verhaltens der Methoden dieser Schnittstelle sind in der Testspezifikation gegeben.

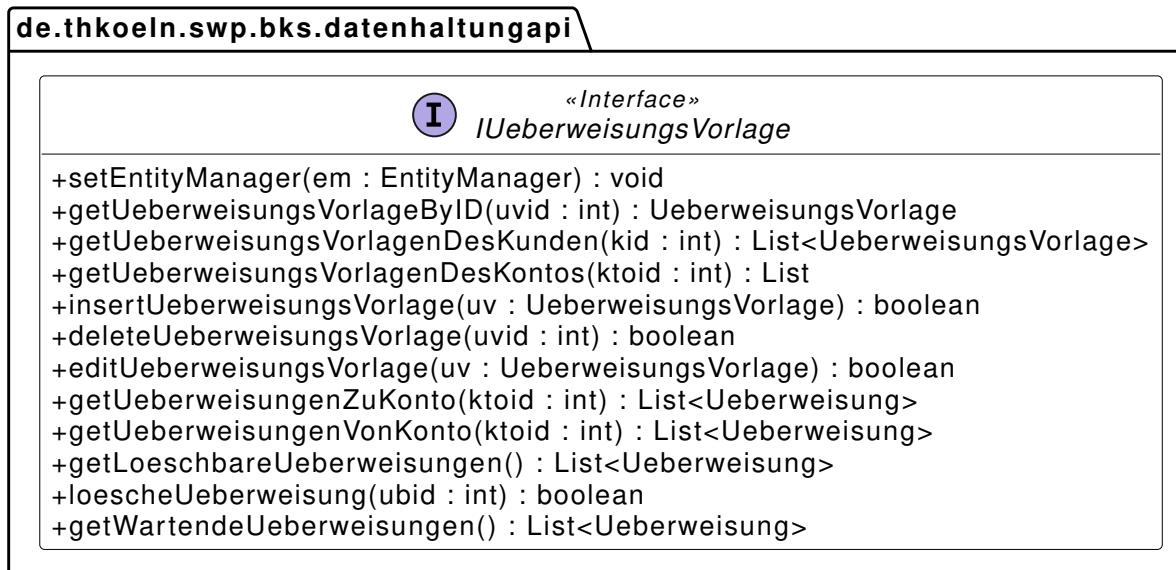
IUeberweisungsVorlage

Abbildung 24. IUeberweisungsVorlage-Schnittstelle zur Verwendung in der SachbearbeiterDaten-Komponente

Grobe Spezifikation der Methoden:

getUeberweisungsVorlageByID

liefert die Überweisungsvorlage mit der angegebenen Id.

getUeberweisungsVorlagenDesKunden

liefert alle Überweisungsvorlagen des als Parameter angegebenen Kunden.

getUeberweisungsVorlagenDesKontos

liefert alle Überweisungsvorlagen die für das als Parameter angegebene Konto vorhanden sind.

insertUeberweisungsVorlage

fügt eine neue Überweisungsvorlage zur DB hinzu.

deleteUeberweisungsVorlage

löscht eine existierende Überweisungsvorlage.

editUeberweisungsVorlage

modifiziert eine existierende Überweisungsvorlage.

getUeberweisungenZuKonto

liefert alle Überweisungen, die auf das als Parameter angegebene Konto überweisen.

getUeberweisungenVonKonto

liefert alle Überweisungen, die von dem als Parameter angegebenen Konto überweisen.

getLoeschbareUeberweisungen

liefert alle Überweisungen, die gelöscht werden können (siehe LF 270).

loescheUeberweisung

entfernt löschrable Überweisungen aus der DB.

getWartendeUeberweisungen

liefert alle wartenden Überweisungen (siehe LF 260).

Die detaillierte Spezifikation des erwarteten Verhaltens der Methoden dieser Schnittstelle sind in der Testspezifikation gegeben.

IAntragSach

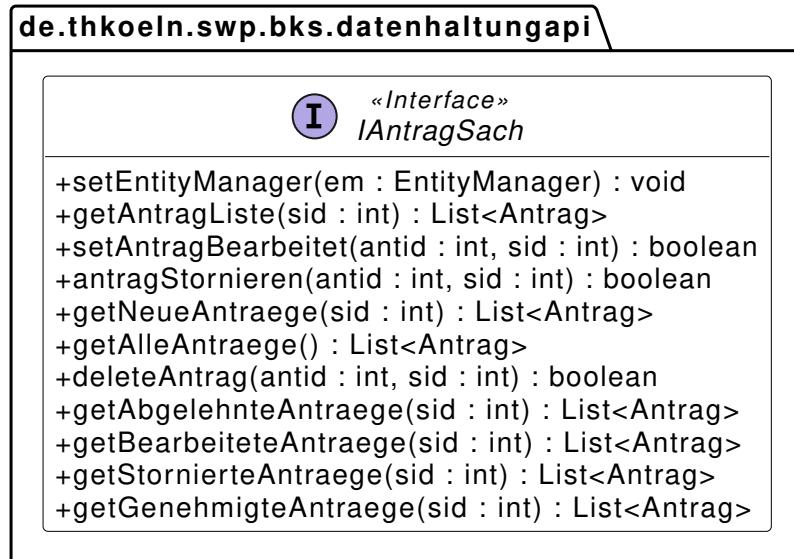


Abbildung 25. `IAntragSach`-Schnittstelle zur Verwendung in der `SachbearbeiterDaten`-Komponente

Grobe Spezifikation der Methoden:

`getAntragListe`

liefert alle Anträge, die der Sachbearbeiter mit der ID `sid` erstellt hat.

`setAntragBearbeitet`

setzt den Status des Antrags auf "bearbeitet", falls dieser vom Sachbearbeiter `sid` erstellt und bearbeitet wurde.

`antragStornieren`

setzt den Status eines vom Sachbearbeiter `sid` stornierbaren Antrags auf "storniert". Ein Sachbearbeiter kann einen Antrag nur dann stornieren, wenn er diesen Antrag auch selbst gestellt hat.

`getNeueAntraege`

liefert alle vom Sachbearbeiter mit der ID `sid` erstellten Anträge, die den Status "neu" besitzen.

`getAlleAntraege`

liefert alle in der DB vorhandenen Anträge.

`deleteAntrag`

löscht den Antrag mit der ID `antid` aus der DB, falls (a) dieser Antrag den Status "abgelehnt" besitzt und (b) der Antrag von dem Sachbearbeiter mit der ID `sid` gestellt wurde.

`getAbgelehnteAntraege`

liefert alle vom Sachbearbeiter mit der ID `sid` erstellten Anträge, die den Status "abgelehnt" besitzen.

getBearbeiteteAntraege

liefert alle vom Sachbearbeiter mit der ID `sid` erstellten Anträge, die den Status "bearbeitet" besitzen.

getStornierteAntraege

liefert alle vom Sachbearbeiter mit der ID `sid` erstellten Anträge, die den Status "storniert" besitzen.

getGenehmigteAntraege

liefert alle vom Sachbearbeiter mit der ID `sid` erstellten Anträge, die den Status "genehmigt" besitzen.

Die detaillierte Spezifikation des erwarteten Verhaltens der Methoden dieser Schnittstelle sind in der Testspezifikation gegeben.



Lastenheft für das
Bankverwaltungssystem (BKS)
Softwarerepraktikum

TH Köln - Informatik Labor

Version 10.0, 2024-01-09 11:43:03 UTC

Inhaltsverzeichnis

1. Produkteinsatz	1
2. Produktübersicht	3
3. Produktfunktionen	5
4. Produktdaten	12
5. Produktleistungen	13
6. Qualitätsanforderungen	14
7. Ergänzungen	15
8. Offene Punkte	16

1. Produkteinsatz

Ein typisches Bankverwaltungssystem ist wesentlich komplexer und besitzt mehr Anwendungsfälle als das unten definierte System. Damit aber das System (BKS) im Rahmen des Software-Praktikums fertiggestellt werden kann und die Entwickler die Entwicklungs-, Test- und Integrationsphasen erleben können, ist der Komplexitätsgrad des Systems angepasst worden. Im Rahmen der Veranstaltung "Software-Praktikum" sollen die Studenten ein Informationssystem, hier ein Bankverwaltungssystem, entwickeln, das die Informationen über Bankkunden, Bankmitarbeitern, Bankkontos und Zahlungsverkehr zur Verfügung stellt.

Es existieren folgende Akteure, die mit dem Bankverwaltungssystem (BKS) interagieren. Die Interaktion zwischen dem System und den Akteuren müssen mittels einer GUI realisiert werden.

Admin

Ein Administrator erhält Anträge vom Manager und vom Sachbearbeiter. Er kann das Profil eines Mitarbeiters (Sachbearbeiter bzw. Manager) einfügen, bearbeiten und löschen. Darüber hinaus ist er für die Wartungsarbeiten (z.B. stornierte und bearbeitete Anträge löschen) an allen Anträgen zuständig.

Sachbearbeiter

Ein Sachbearbeiter kann auf Wunsch des Kunden seine Überweisungen erfassen bzw. stornieren. Er muss für die Konto-Eröffnung, -Schließung oder -Limit zunächst einem Manager einen Antrag stellen und wenn dieser Antrag vom Manager genehmigt wird, dann kann er diesen Antrag bearbeiten. Der Kunde kann auch beim Sachbearbeiter das Geld in sein Konto einzahlen bzw. von seinem Konto abheben. Registrierung von Kunden und Kontos findet ebenfalls beim Sachbearbeiter statt. Schließlich kann er sich die Liste der von ihm gestellten Anträge anschauen und ggf. einen oder mehrere Anträge bearbeiten oder stornieren.

Manager

Ein Manager ist für die Genehmigung aller Anträge zuständig. Er kann die Anträge, die von Sachbearbeitern gestellt wurden, genehmigen oder ablehnen. Falls er ein Konto-Limitantrag genehmigt, muss er diesen Antrag selber bearbeiten und dem Kunden ein neues Konto-Limit (Dispo) zuweisen. Wenn ein neuer Sachbearbeiter in der Bank eingestellt wird, dann stellt er für ihn einen neuen Sachbearbeiterprofil-Antrag an den Admin. Zur Bearbeitung und Löschung des Sachbearbeiterprofils stellt er ebenfalls Anträge an einen Admin. Diese Anträge werden dann vom Admin durchgeführt. Der Manager kann sich die Liste aller überzogenen Kontos anschauen und ggf. ein Konto sperren und dessen Inhaber eine Mahnung zustellen. Schließlich kann der Manager sich die Liste der von ihm gestellten Anträge anschauen und ggf. einen oder mehrere Anträge auswählen und stornieren.

Kunde

Ein Kunde kann seine eigenen Profil-Daten außer Name und Konto-Daten bearbeiten und speichern. Er kann für seine Überweisungen sog. Überweisungsvorlage anlegen und bearbeiten. Anhand einer Überweisungsvorlage kann er seine Überweisungen erfassen. Zum Stornieren kann er eine Überweisung aus der Liste der stornierbaren Überweisungen auswählen und stornieren. Er kann aus seiner eigenen Kontoliste ein Konto auswählen und sich dessen Kontoauszug anschauen und ausdrucken.

Überweisungsbearbeiter

Der Überweisungsbearbeiter ist eine besondere Rolle, die in der Praxis nicht in dieser Form auftritt. Die Aufgaben der Rolle sind die Durchführung der Überweisungen an dem korrekten Datum und die Wartung der abgegebenen Überweisungen. In der Praxis werden diese Aufgaben typischerweise als ein automatisierter, serverseitiger Prozess realisiert. Aber in unserem Bankverwaltungssystem werden diese Aufgaben manuell über eine GUI realisiert.

2. Produktübersicht

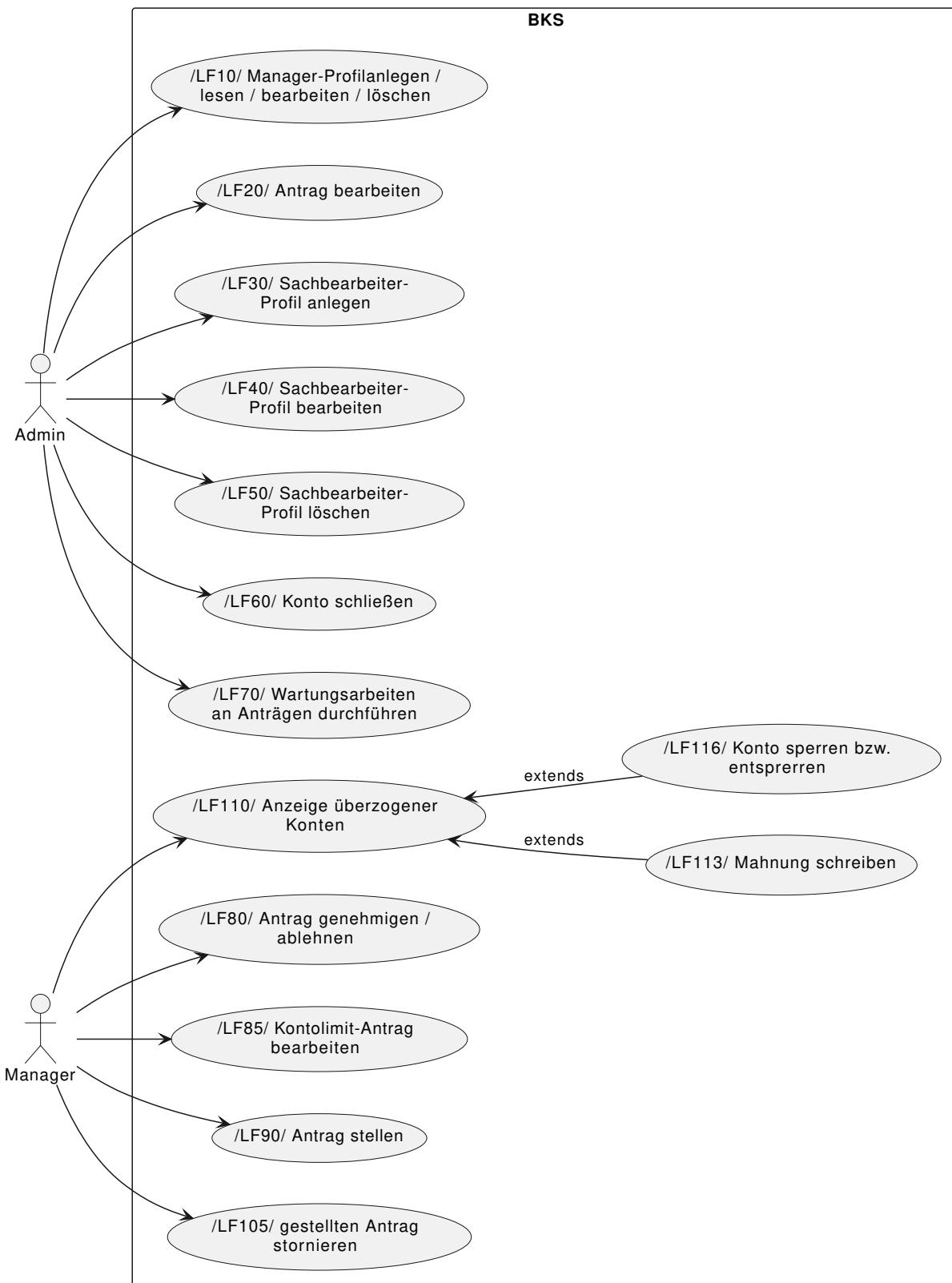


Abbildung 1. Anwendungsfall-Diagramm des Bankverwaltungssystems (nur Admin und Manager)

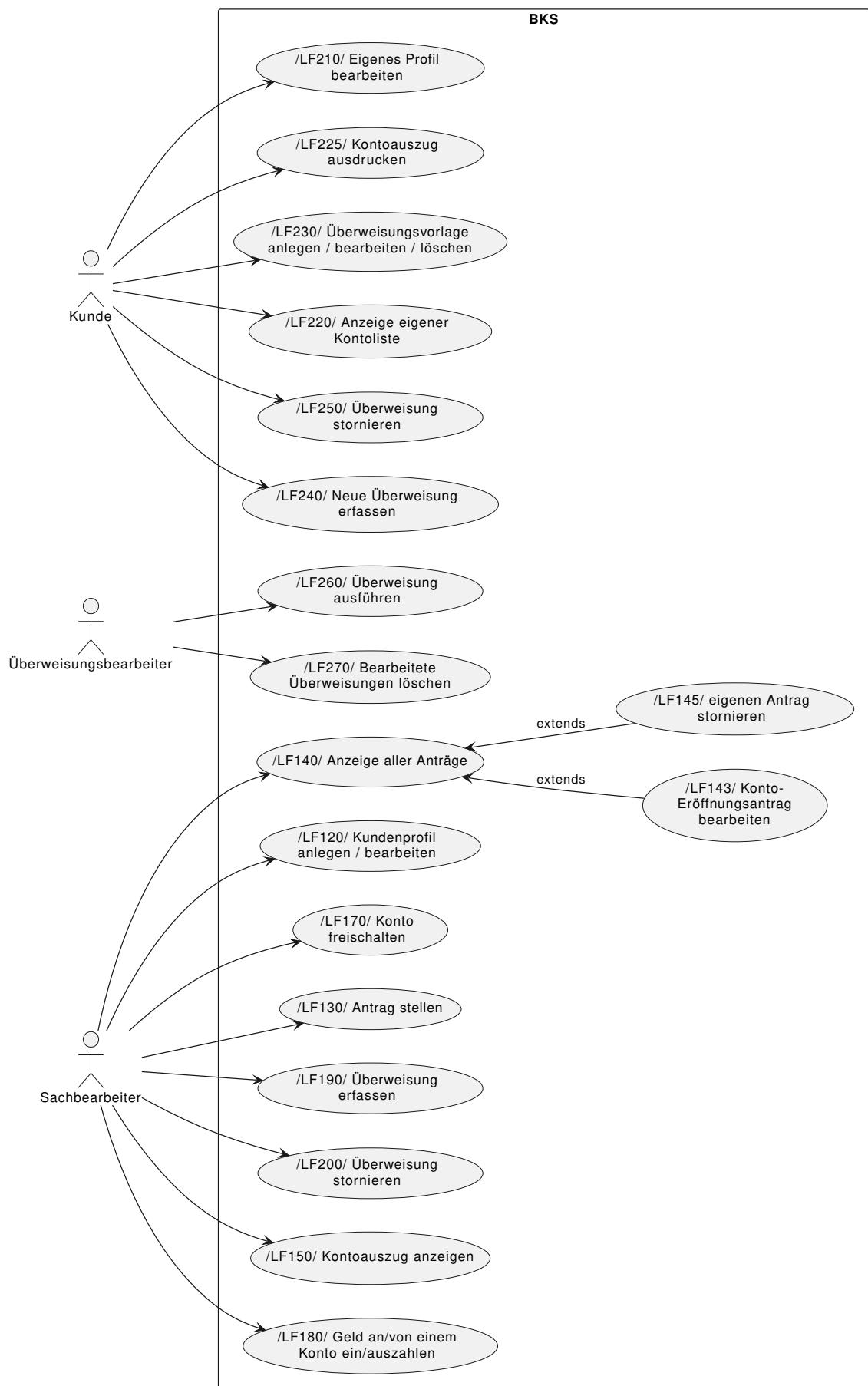


Abbildung 2. Anwendungsfall-Diagramm des Bankverwaltungssystems (nur Sachbearbeiter, Kunde und Überweisungsbeauftragter)

3. Produktfunktionen

/LF10/ Muss

Anwendungsfall: Manager-Profil anlegen / lesen / bearbeiten / löschen

Akteure: Admin

Beschreibung: Der Admin soll Manager-Profildaten registrieren, lesen, bearbeiten und löschen können. Er benötigt hierzu keine Anträge.

/LF20/ Muss

Anwendungsfall: Antrag bearbeiten

Akteure: Admin

Beschreibung: Der Admin soll über eine GUI eine Liste aller Anträge anzeigen lassen können, bei denen er für die Bearbeitung verantwortlich ist. Der Admin ist für die Bearbeitung folgender Anträge verantwortlich:

- Sachbearbeiter-Anlegen
- Sachbearbeiter-Löschen
- Sachbearbeiter-Bearbeiten
- Kontoschließung



Anträge des Typs Kontoschließung können erst dann von einem Admin bearbeitet werden, wenn diese zuvor von einem Manager genehmigt wurden. Diese Anträge dürfen in der Liste nur erscheinen, wenn sie bereits von einem Manager genehmigt wurden.

Dann soll er in der Liste einen Antrag auswählen und sich die Details des Antrags zu seiner Information anzeigen lassen können. Der Admin kann weiterhin einen Antrag in der Liste auswählen und die Bearbeitung dieses Antrags starten: In der GUI sollte dieses durch einen entsprechenden Button erfolgen. Die konkrete Bearbeitung eines selbst ausgewählten Antrags (nicht automatisch durch den Antragstext) wird dann in Abhängigkeit vom Typ des Antrags in einem der Anwendungsfälle /LF30/, /LF40/, /LF50/, /LF60/ erfolgen.

Nach Abschluss der Bearbeitung eines Antrags soll der Admin in der GUI diesen bearbeiteten Antrag auswählen können und seinen Status entsprechend setzen können.

/LF30/ Muss

Anwendungsfall: Sachbearbeiter-Profil anlegen

Akteure: Admin

Beschreibung: Wenn der Admin einen "Sachbearbeiter-Anlegen"-Antrag bearbeitet, soll er ein Profil für einen neuen Sachbearbeiter entsprechend des Inhalts des Antrags anlegen.

/LF40/ Muss

Anwendungsfall: Sachbearbeiter-Profil bearbeiten

Akteure: Admin

Beschreibung: Wenn der Admin einen "Sachbearbeiter-Bearbeiten"-Antrag bearbeitet, soll er das Profil des betroffenen Sachbearbeiters entsprechend den Angaben im Antrag bearbeiten.

/LF50/ Muss

Anwendungsfall: Sachbearbeiter-Profil löschen

Akteure: Admin

Beschreibung: Wenn der Admin einen "Sachbearbeiter-Löschen"-Antrag bearbeitet, soll der Admin das Profil des im Antrag genannten Sachbearbeiters löschen.

/LF60/ Muss

Anwendungsfall: Konto schließen

Akteure: Admin

Beschreibung: Wenn der Admin einen "Kontoschließung"-Antrag bearbeitet, soll der Admin das im Antrag angegebene Konto schließen.

/LF70/ Muss

Anwendungsfall: Wartungsarbeiten an Anträgen durchführen

Akteure: Admin

Beschreibung: Der Admin ist verantwortlich für die Durchführung von Wartungsarbeiten an Anträgen. Dafür soll er sich über eine GUI eine Liste aller Anträge anzeigen lassen können. Dann soll er einen oder mehrere stornierte oder bearbeitete Anträge auswählen und löschen können.

/LF80/ Muss

Anwendungsfall: Antrag genehmigen / ablehnen

Akteure: Manager

Beschreibung: Der Manager soll mithilfe einer GUI die an ihn gestellten Konto-Eröffnungs- und Konto-Schließungs-Anträge als Liste anzeigen lassen können und daraus einen auswählen und lesen können. Aus der Antragsliste soll der Manager dann einen Antrag auswählen und ihn genehmigen oder ablehnen können.

Speziell bei Genehmigung eines Konto-Schließungs-Antrags: Der Manager soll aus der Liste der existierenden Admins einen Admin zur Bearbeitung des Konto-Schließungs-Antrags auswählen können. Dieser Admin soll bei dem Antrag dann als bearbeitender Admin eingetragen werden.

/LF85/ Muss

Anwendungsfall: Kontolimit-Antrag bearbeiten

Akteure: Manager

Beschreibung: Der Manager soll sich alle an ihn gestellten Konto-Limitanträge in einer Liste anzeigen lassen und daraus einen zur Bearbeitung auswählen können. Zur Bearbeitung eines Konto-Limitantrags soll eine separate GUI zur Verfügung stehen, in der die Konto-Daten für den Manager sichtbar sind. Der Manager soll dann den Dispo des Kontos ändern können. Alternativ kann der Manager den Antrag auch ablehnen. Der

Status des entsprechenden Antrags soll entsprechend geändert werden.

/LF90/ Muss

Anwendungsfall: Antrag stellen

Akteure: Manager

Beschreibung: Der Manager soll über eine GUI folgende Anträge an einen bestimmten Admin (durch Auswahl aus einer Adminliste) stellen können:

- Neues Sachbearbeiterprofil Antrag
- Sachbearbeiterprofil-Bearbeitungsantrag
- Sachbearbeiterprofil-Löschantrag

/LF105/ Muss

Anwendungsfall: gestellten Antrag stornieren

Akteure: Manager

Beschreibung: Der Manager soll sich eine Liste der von ihm gestellten Anträge ansehen können. Der Manager soll aus dieser Liste einen oder mehrere Anträge stornieren können.

/LF110/ Muss

Anwendungsfall: Anzeige der überzogenen Konten

Akteure: Manager

Beschreibung: Der Manager soll sich mittels einer GUI eine Liste der überzogenen Konten und den zugehörigen Kunden anschauen können.

/LF113/ Muss

Anwendungsfall: Mahnung schreiben

Akteure: Manager

Beschreibung: Dieser Anwendungsfall erweitert /LF110/. Aus der Liste soll er ein Konto eines Kunden auswählen und ihm eine Mahnung schreiben können. Die Mahnung soll er als eine ".txt"-Datei speichern können.

/LF116/ Muss

Anwendungsfall: Konto sperren bzw. entsperren

Akteure: Manager

Beschreibung: Dieser Anwendungsfall erweitert /LF110/. Aus der Liste soll er ein Konto auswählen und dieses Konto sperren oder entsperren können, indem er das Kredit-Limit (Dispo) des Kontos verändert.

/LF120/ Muss

Anwendungsfall: Kundenprofil anlegen / bearbeiten

Akteure: Sachbearbeiter

Beschreibung: Der Sachbearbeiter soll über eine GUI einen neuen Kunden registrieren und die Profildaten eines existierenden Kunden bearbeiten können. Hierzu benötigt er keinen Antrag.

/LF130/ Muss

Anwendungsfall: Antrag erstellen

Akteure: Sachbearbeiter

Beschreibung: Der Sachbearbeiter soll mittels einer GUI die folgenden Anträge erstellen können und diese an einen bestimmten Manager durch Auswahl aus einer Managerliste richten:

- Konto-Eröffnungsantrag
- Konto-Schließungsantrag
- Konto-Limitantrag

Der Antrag muss den korrekten Zustand erhalten.

/LF140/ Muss

Anwendungsfall: Anzeige eigener Anträge

Akteure: Sachbearbeiter

Beschreibung: Der Sachbearbeiter soll eine Liste der von ihm gestellten Anträge sehen können.

/LF143/ Muss

Anwendungsfall: Konto-Eröffnungsantrag bearbeiten

Akteure: Sachbearbeiter

Beschreibung: Dieser Anwendungsfall erweitert /LF140/. Aus der Liste soll der Sachbearbeiter einen Konto-Eröffnungsantrag zur Bearbeitung auswählen können, falls dieser genehmigt ist. Der Sachbearbeiter soll ein neues Konto für einen existierenden Kunden anlegen können. Hierzu soll dem Sachbearbeiter eine GUI zur Verfügung stehen, in der er einen Kunden aus der Kundenliste auswählen und die Konto-Daten eingeben kann. Der Status des Konto-Eröffnungsantrags soll dementsprechend geändert werden.

/LF145/ Muss

Anwendungsfall: eigenen Antrag stornieren

Akteure: Sachbearbeiter

Beschreibung: Dieser Anwendungsfall erweitert /LF140/. Aus der Liste soll er dann gewünschte Anträge auswählen und stornieren können, falls diese den erforderlichen Zustand aufweisen.

/LF150/ Muss

Anwendungsfall: Kontoauszug anzeigen

Akteure: Sachbearbeiter

Beschreibung: Auf Wunsch eines Kunden soll der Sachbearbeiter die Kontobewegungen des Kunden in eines bestimmten Zeitraums anzeigen lassen können (aber nicht ausdrucken). Dabei soll der Sachbearbeiter ein Datumsintervall (von-bis) und ein Konto aus der Kontoliste des Kunden auswählen können. Die Kontobewegungen werden hierdurch **nicht** als ausgedruckt markiert!

/LF170/ Muss

Anwendungsfall: Konto freischalten

Akteure: Sachbearbeiter

Beschreibung: Der Sachbearbeiter soll auf Wunsch eines Managers (nennt eine KontoId) ein bereits angelegtes Konto freischalten können (d.h. das Konto kann ab jetzt verwendet werden). Der Zustand des Kontos soll entsprechend gesetzt werden.

/LF180/ Muss

Anwendungsfall: Geld an/von einem Konto ein/auszahlen

Akteure: Sachbearbeiter

Beschreibung: Der Sachbearbeiter soll auf Wunsch des Kunden Geld auf dessen Konto einzahlen oder von dessen Konto abheben. Vor der Ein- oder Auszahlung soll der Sachbearbeiter aus der Kontoliste des Kunden ein Konto auswählen können. Auszahlungen sind nur erlaubt, wenn das Konto bisher nicht gesperrt ist.

Ein- und Auszahlungen sollen in System als Zahlungsverkehr für das entsprechende Konto gespeichert werden.

/LF190/ Muss

Anwendungsfall: Überweisung erfassen

Akteure: Sachbearbeiter

Beschreibung: Der Sachbearbeiter soll auf Wunsch des Kunden eine Überweisung erfassen können. Hierzu soll dem Sachbearbeiter eine GUI zur Verfügung stehen, in der er ein Konto eines Kunden aus einer Liste auswählt umso eine Überweisung erstellen zu können. Hierbei ist auf den korrekten Zustand der neu erstellten Überweisung zu achten.

/LF200/ Muss

Anwendungsfall: Überweisung stornieren

Akteure: Sachbearbeiter

Beschreibung: Zum Stornieren einer Überweisung soll sich der Sachbearbeiter eine Liste aller stornierbaren Überweisungen des Kunden anzeigen lassen. Hieraus kann er eine auswählen und diese dann stornieren.

/LF210/ Muss

Anwendungsfall: Eigenes Profil bearbeiten

Akteure: Kunde

Beschreibung: Der Kunde soll über eine GUI seine eigenen Profil-Daten außer Name und Konto-Daten bearbeiten können.

/LF220/ Muss

Anwendungsfall: Anzeige eigener Kontoliste

Akteure: Kunde

Beschreibung: Der Kunde soll sich über eine GUI seine Kontoliste anschauen können.

/LF225/ Muss

Anwendungsfall: Kontoauszug ausdrucken

Akteure: Kunde

Beschreibung: Dieser Anwendungsfall erweitert /LF220/. Der Kunde soll sich ein Konto aus seiner Kontoliste auswählen können. Nach der Auswahl soll er den Kontoauszug des

ausgewählten Kontos anzeigen und ausdrucken lassen können. Der Kontoauszug soll alle Kontobewegungen enthalten, die noch nicht als ausgedruckt markiert sind. Die ausgedruckten Einträge sollen nach dem Anzeigen und Ausdrucken als „ausgedruckt“ in der Datenbank vermerkt werden, damit sie beim nächsten Kontoauszug nicht nochmals erscheinen.



Der Druckauftrag soll nicht an einen echten Drucker gesendet werden sondern in eine ".txt" Datei geschrieben werden. Dabei soll aber auf die korrekte Formatierung Rücksicht genommen werden.

/LF230/ Muss

Anwendungsfall: Überweisungsvorlage anlegen/ bearbeiten / löschen

Akteure: Kunde

Beschreibung: Der Kunde soll über eine GUI eine Überweisungsvorlage anlegen, bereits existierende Überweisungsvorlagen bearbeiten oder löschen können. Zur Bearbeitung oder Löschung soll der Kunde die Überweisungsvorlagen aus einer Liste seiner existierenden Überweisungsvorlagen auswählen können.

/LF240/ Muss

Anwendungsfall: Neue Überweisung erfassen

Akteure: Kunde

Beschreibung: Der Kunde soll aus der Liste seiner Überweisungsvorlagen eine Überweisungsvorlage auswählen und darauf basierend eine Überweisung erfassen können. Er soll auch ohne Überweisungsvorlage eine Überweisung erfassen können.

/LF250/ Muss

Anwendungsfall: Überweisung stornieren

Akteure: Kunde

Beschreibung: Der Kunde soll sich eine Liste aller seiner stornierbaren Überweisungen anzeigen lassen können und eine gewünschte Überweisung auswählen und sie stornieren können.

/LF260/ Muss

Anwendungsfall: Überweisungen ausführen

Akteure: Überweisungsbearbeiter

Beschreibung: Über eine GUI soll es möglich sein, eine Liste aller Überweisungen, die den Status wartet haben, anzeigen zu lassen. Diese Überweisungsliste soll nach ihrem Überweisungsdatum gruppiert sein. Zusätzlich soll die Überweisungsliste nach ihrem Überweisungsdatum gefiltert werden können, d.h. nur die Anzeige der Überweisungen, die das heutige Datum (das Datum, wann die GUI ausgeführt wird) als ihr Überweisungsdatum haben. Das Auswählen einer oder mehrerer Überweisungen soll ebenfalls möglich sein und per Knopfdruck sollen die ausgewählten Überweisungen ausgeführt werden können. Ein Fehlerfall soll in der GUI angezeigt werden.

Ausgeführte Überweisungen sollen im System als Zahlungsverkehr bei beiden betroffenen Konten gespeichert werden.

/LF270/ Muss

Anwendungsfall: Bearbeitete Überweisungen löschen

Akteure: Überweisungsbearbeiter

Beschreibung: Eine GUI soll eine Liste aller überwiesenen, nicht überweisbaren und stornierten Überweisungen bereitstellen. Das Auswählen einer oder mehrerer Überweisungen aus der Liste soll möglich sein und per Knopfdruck sollen die ausgewählten Überweisungen aus der Datenbank gelöscht werden.

4. Produktdaten

/LD10/ Kunde
/LD20/ Sachbearbeiter
/LD30/ Admin
/LD40/ Manager
/LD50/ Antrag
/LD60/ Konto
/LD70/ Überweisungsvorlage
/LD80/ Überweisung
/LD90/ Zahlungsverkehr

5. Produktleistungen

Nicht anwendbar

6. Qualitätsanforderungen

/LQ10/ Funktionalität: sehr gut

/LQ20/ Zuverlässigkeit: sehr gut

/LQ30/ Benutzbarkeit: sehr gut

/LQ40/ Effizienz: gut

/LQ50/ Änderbarkeit und Wartbarkeit: sehr gut - insbesondere soll die Datenquelle ohne viele Änderungen austauschbar sein, z.B. von einer SQL-Datenbank als Datenquelle soll sehr leicht auf ein Dateisystem als Datenquelle umgestellt werden können.

/LQ60/ Portierbarkeit: sehr gut

7. Ergänzungen

Nicht anwendbar

8. Offene Punkte

Nicht anwendbar



BKS Testspezifikation 2023

Softwarepraktikum

TH Köln - Informatik Labor

Version 10.1, 2023-06-29 12:24:46 UTC

Inhaltsverzeichnis

1. Vorgaben zur Testspezifikation im SWP	2
1.1. Strukturierung der Testspezifikation und der Testfälle	2
2. Testspezifikation	5
2.1. Testspezifikation zur Komponente AdminDaten	5
2.1.1. IAntragAdmin	5
2.1.2. IAntragStellenAdmin	8
2.2. Testspezifikation zur Komponente KundeDaten	10
2.2.1. ICRUDKunde	10
2.3. Testspezifikation zur Komponente KontoDaten	11
2.3.1. ICRUDKonto	12
2.3.2. IKontoService	18
2.3.3. ISonderKontoService	24
2.4. Testspezifikation zur Komponente ManagerDaten	27
2.4.1. ICRUDManager	27
2.4.2. IAntragManager	29
2.4.3. IAntragStellenManager	35
2.5. Testspezifikation zur Komponente SachbearbeiterDaten	37
2.5.1. ICRUDSach	37
2.5.2. IUeberweisungsVorlage	38
2.5.3. IAntragSach	43

Changelog:

v10.1: kontoAktualisieren_08 geupdated

1. Vorgaben zur Testspezifikation im SWP

Das Testen einer Software ist ein immens wichtiger Teil während eines Software Entwicklungsprozesses. Viele Fehler lassen sich nur durch ausgiebiges Testen finden und beseitigen. Je größer die Software ist umso umfangreicher und auch wichtiger werden die Tests.

Ein Test prüft, ob das Verhalten einer Methode zu einem gewünschten Ergebnis führt. Dabei ist zu beachten, dass Tests unabhängig von persistent abgelegten Daten durchgeführt werden, d.h. zum Beispiel unabhängig von Werten die in einer Datenbank gespeichert sind. Dies kann man garantieren, indem man selbstständig vor Durchlauf eines eigentlichen Testfalls sicher stellt, dass der gewünschte Ausgangszustand hergestellt ist. Eine mögliche Strategie könnte dabei sein, zu Beginn eines Tests die Daten einer betreffenden Entität einer Datenbank zu löschen.

Da die Datenbank durch einen durchgeföhrten Test nicht verändert werden soll, muss zwingend nach Beenden eines Test ein Datenbank-Rollback durchgeföhrt werden. Hierbei werden alle Aktionen, die innerhalb der letzten Transaktion durchgeföhrt wurden, rückgängig gemacht. Niemals sollte ein **commit** innerhalb eines Testfalls erfolgen.

Im Software Praktikum werden nur die Schnittstellen-Implementierungen getestet. Die Methoden der Steuerungsklassen können/sollten auf Grund des zu großen Aufwands außer acht gelassen werden.

Diese Testspezifikation ist ein Dokument, das die zu erstellenden Testfälle in einer strukturierten Weise kurz und prägnant beschreibt.

Die Testspezifikation umfasst alle Testfälle für die Implementierung der Methoden der Schnittstellen der Komponenten der Schicht "Datenhaltung".

Die in dieser Testspezifikation beschriebenen Testfälle sollen in JUnit implementiert werden. Die Gestaltung der Testspezifikation ist deshalb bereits an JUnit angepasst, um eine leichte Umsetzung in JUnit zu ermöglichen. Der Text der Testspezifikation soll als Java-Kommentar des konkreten JUnit-Testfalls verwendet werden.

1.1. Strukturierung der Testspezifikation und der Testfälle

- In der Testspezifikation sind Testfälle für alle Methoden aller Implementierungs-Klassen der Schnittstellen der Komponente definiert.
- Es werden also die Implementierungsklassen getestet, **nicht** die Schnittstellen-Klassen.
- Für die Testfälle einer Klasse kann eine **@Before-**, **@BeforeClass-**, **@After-** und **@AfterClass**

-Methode definiert werden

- Jeder einzelne Testfall besitzt als Überschrift **@Test** und den Namen der später zu implementierenden JUnit-Testmethode
- Ein Testfall ist in Form einer Wenn-Dann-Regel definiert.
 - **WENN**, **UND** und **DANN** sind hierbei Schlüsselwörter.
 - Der **WENN**-Teil definiert die Voraussetzung für die Durchführung des Testfalls und den Aufruf der zu testenden Methode:
 - Dieses kann z.B. das Vorhandensein eines bestimmten Objekts sein
 - oder das Vorhandensein eines bestimmten Datenbank-Eintrags
 - oder das Nicht-Vorhandensein eines bestimmten Datenbank-Eintrags
 - oder ein bestimmter Status eines Antrags oder einer Überweisung o.ä.
 - oder ...
 - Aufruf der zu testenden Methode: hier werden die Parameterwerte vorgegeben.
 - Innerhalb des **WENN**-Teils werden die unterschiedlichen Angaben durch das Schlüsselwort **UND** voneinander getrennt. Es wird jeweils eine neue Zeile begonnen.
 - Der **DANN**-Teil definiert die erwartete Rückgabe und alle sonstigen Seiteneffekte der Methodenausführung, wie bspw. erwartete Rückgabe, Änderung des Zustands eines Objekts, Eintrag/Update/Lösung in der DB, Setzen eines Attributwertes.
 - Es gibt keine ODER Verknüpfung. ODER Verknüpfungen können nur durch mehrere Tests durchgeführt werden.
 - Für den überwiegenden Teil der Testfälle existiert ein Positiv- und ein Negativtestfall.
- Beispiel 1 eines Testfalls in der Testspezifikation:
`@Test insertKunde_00()`
WENN die Methode **insertKunde** mit einem Testkunden aufgerufen wird,
UND die ID des Testkunden gleich **null** ist,
DANN sollte sie **TRUE** zurückliefern,
UND der Testkunde sollte im Persistence Context existieren.

`@Test insertKunde_01()`
WENN die ID eines Testkunden mit einem Wert **ungleich null** besetzt ist,
UND die Methode **insertKunde** mit dem Testkunden aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern,
UND und die DB wurde nicht verändert.
- Beispiel 2 eines Testfalls in der Testspezifikation:
`@Test getKundeById_00()`
WENN ein Testkunde bereits in der DB existiert,
UND die Methode **getKundeById** mit der Id des Testkunden aufgerufen wird,
DANN sollte sie den **Testkunden** zurückliefern.

@Test getKundeById_01()

WENN ein Testkunde nicht in der DB existiert,

UND die Methode **getKundeById** mit der Id des Testkunden aufgerufen wird,

DANN sollte sie **NULL** zurückliefern.

- Beispiel 3 eines Testfalls in der Testspezifikation:

@Test getKundenListe_00()

WENN x ($x > 0$) Kunden in der DB existieren,

UND die Methode **getKundenListe** aufgerufen wird,

DANN sollte sie eine Liste mit x Kunden zurückliefern.

@Test getKundenListe_01()

WENN keine Kunden in der DB existieren,

UND die Methode **getKundenListe** aufgerufen wird,

DANN sollte sie eine **leere Liste** zurückliefern.

- Beispiel 4 eines Testfalls in der Testspezifikation:

@Test deleteKunde_00()

WENN ein Testkunde in der DB existiert,

UND die Methode **deleteKunde** mit der ID des Testkunden aufgerufen wird,

DANN sollte sie **TRUE** zurückliefern,

UND der Testkunde sollte nicht mehr in der DB existieren.

@Test deleteKunde_01()

WENN ein Testkunde nicht in der DB existiert,

UND die Methode **deleteKunde** mit der ID des Testkunden aufgerufen wird,

DANN sollte sie **FALSE** zurückliefern.

2. Testspezifikation

2.1. Testspezifikation zur Komponente AdminDaten

2.1.1. IAntragAdmin

- @Before: angenommen()

Angenommen der EntityManager wird korrekt geholt,
UND die Implementierung der IAntragAdmin Schnittstelle wird als classUnderTest instanziert,
UND der EntityManager em wird per setEntityManager Methode der classUnderTest gesetzt,
UND die Transaktion von em wird gestartet,
UND die Daten der betreffenden Entitäten wurden im Persistence Context gelöscht.
- @After: amEnde()

Am Ende wird die Transaktion zurück gesetzt.
- @Test: getAntragListe_00()

WENN ein Testadmin bereits in der Datenbank existiert,
UND x ($x > 0$) Anträge in der DB existieren, die an diesen Testadmin gerichtet sind,
UND die Methode **getAntragListe** mit der ID des Testadmins aufgerufen wird,
DANN sollte sie die Liste mit diesen x Anträgen zurückliefern.
- @Test: getAntragListe_01()

WENN ein Testadmin bereits in der Datenbank existiert,
UND keine Anträge in der DB existieren, die an diesen Testadmin gerichtet sind,
UND die Methode **getAntragListe** mit der ID des Testadmins aufgerufen wird,
DANN sollte sie eine leere Liste zurückliefern.
- @Test: getAntragListe_02()

WENN ein Testadmin nicht in der Datenbank existiert,
UND die Methode **getAntragListe** mit der ID des Testadmins aufgerufen wird,
DANN sollte sie eine **leere Liste** zurückliefern.
- @Test: getOffeneAntraege_00()

WENN ein Testadmin bereits in der Datenbank existiert,
UND x ($x > 0$) Anträge in der DB existieren, die an diesen Testadmin gerichtet sind und den Status genehmigt (g) besitzen,
UND die Methode **getOffeneAntraege** mit der ID des Testadmins aufgerufen wird,
DANN sollte sie die Liste mit diesen x Anträgen zurückliefern.
- @Test: getOffeneAntraege_01()

WENN ein Testadmin bereits in der Datenbank existiert,
UND keine Anträge in der DB existieren, die an diesen Testadmin gerichtet sind und den Status genehmigt (g) besitzen,

UND die Methode `getOffeneAntraege` mit der ID des Testadmins aufgerufen wird,
DANN sollte sie eine leere Liste zurückliefern.

- @Test: `getOffeneAntraege_02()`
WENN ein Testadmin nicht in der Datenbank existiert,
UND die Methode `getOffeneAntraege` mit der ID des Testadmins aufgerufen wird,
DANN sollte sie eine **leere Liste** zurückliefern.
- @Test: `setAntragBearbeitet_00()`
WENN ein Testantrag bereits in der DB existiert,
UND der Testantrag den Typ "Sachbearbeiter-Anlegen-Antrag" besitzt,
UND der Testantrag den Status "neu" besitzt,
UND die Methode `setAntragBearbeitet` mit der Id des Testantrags aufgerufen wird,
DANN sollte sie den Status des **Testantrags** auf "bearbeitet" setzen,
UND TRUE zurückliefern
- @Test: `setAntragBearbeitet_01()`
WENN ein Testantrag bereits in der DB existiert,
UND der Testantrag den Typ "Sachbearbeiter-Löschen-Antrag" besitzt,
UND der Testantrag den Status "neu" besitzt,
UND die Methode `setAntragBearbeitet` mit der Id des Testantrags aufgerufen wird,
DANN sollte sie den Status des **Testantrags** auf "bearbeitet" setzen,
UND TRUE zurückliefern
- @Test: `setAntragBearbeitet_02()`
WENN ein Testantrag bereits in der DB existiert,
UND der Testantrag den Typ "Sachbearbeiter-Bearbeiten-Antrag" besitzt,
UND der Testantrag den Status "neu" besitzt,
UND die Methode `setAntragBearbeitet` mit der Id des Testantrags aufgerufen wird,
DANN sollte sie den Status des **Testantrags** auf "bearbeitet" setzen,
UND TRUE zurückliefern
- @Test: `setAntragBearbeitet_03()`
WENN ein Testantrag bereits in der DB existiert,
UND der Testantrag den Typ "Kontoschließungs-Antrag" besitzt,
UND der Testantrag den Status "genehmigt" besitzt,
UND die Methode `setAntragBearbeitet` mit der Id des Testantrags aufgerufen wird,
DANN sollte sie den Status des **Testantrags** auf "bearbeitet" setzen,
UND TRUE zurückliefern
- @Test: `setAntragBearbeitet_04()`
WENN ein Testantrag bereits in der DB existiert,
UND der Testantrag den Typ "Kontoeröffnungs-Antrag" besitzt,
UND die Methode `setAntragBearbeitet` mit der Id des Testantrags aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern,
UND der Testantrag wurde nicht verändert.
- @Test: `setAntragBearbeitet_05()`
WENN ein Testantrag bereits in der DB existiert,

UND der Testantrag den Typ "Kontolimit-Antrag" besitzt,
UND die Methode **setAntragBearbeitet** mit der Id des Testantrags aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern,
UND der Testantrag wurde nicht verändert.

- @Test: setAntragBearbeitet_06()

WENN ein Testantrag bereits in der DB existiert,
UND der Testantrag den Status "bearbeitet" besitzt,
UND die Methode **setAntragBearbeitet** mit der Id des Testantrags aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern,
UND der Testantrag wurde nicht verändert.
- @Test: setAntragBearbeitet_07()

WENN ein Testantrag nicht in der DB existiert,
UND die Methode **setAntragBearbeitet** mit der Id des Testantrags aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern.
- @Test: getAlleAntraege_00()

WENN x ($x > 0$) Anträge in der DB existieren,
UND die Methode **getAlleAntraege** aufgerufen wird,
DANN sollte sie die Liste mit diesen **x** Anträgen zurückliefern.
- @Test: getAlleAntraege_01()

WENN keine Anträge in der DB existieren,
UND die Methode **getAlleAntraege** aufgerufen wird,
DANN sollte sie eine **leere Liste** zurückliefern.
- @Test: deleteAntrag_00()

WENN ein Testantrag in der DB existiert,
UND der Testantrag den Status "storniert" besitzt,
UND die Methode **deleteAntrag** mit der ID des Testantrags aufgerufen wird,
DANN sollte sie **TRUE** zurückliefern,
UND der Testantrag sollte nicht mehr im Persistence Context existieren.
- @Test: deleteAntrag_01()

WENN ein Testantrag in der DB existiert,
UND der Testantrag den Status "bearbeitet" besitzt,
UND die Methode **deleteAntrag** mit der ID des Testantrags aufgerufen wird,
DANN sollte sie **TRUE** zurückliefern,
UND der Testantrag sollte nicht mehr im Persistence Context existieren.
- @Test: deleteAntrag_02()

WENN ein Testantrag in der DB existiert,
UND der Testantrag den Status "genehmigt" besitzt,
UND die Methode **deleteAntrag** mit der ID des Testantrags aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern,
UND der Testantrag sollte im Persistence Context existieren.
- @Test: deleteAntrag_03()

WENN ein Testantrag nicht in der DB existiert,

UND die Methode `deleteAntrag` mit der ID des Testantrags aufgerufen wird,
DANN sollte sie `FALSE` zurückliefern.

- @Test: `setKontoGeschlossen_00()`
WENN ein Testkonto in der DB existiert,
UND das Testkonto den Status "haben" besitzt,
UND die Methode `setKontoGeschlossen` mit der ID des Testkontos aufgerufen wird,
DANN sollte sie `TRUE` zurückliefern,
UND das Testkonto sollte im Persistennce Context den Status "geschlossen" besitzen.
- @Test: `setKontoGeschlossen_01()`
WENN ein Testkonto in der DB existiert,
UND das Testkonto den Status "init" besitzt,
UND die Methode `setKontoGeschlossen` mit der ID des Testkontos aufgerufen wird,
DANN sollte sie `FALSE` zurückliefern,
UND das Testkonto sollte im Persistennce Context immer noch den Status "init" besitzen.
- @Test: `setKontoGeschlossen_02()`
WENN ein Testkonto in der DB existiert,
UND das Testkonto den Status "soll" besitzt,
UND die Methode `setKontoGeschlossen` mit der ID des Testkontos aufgerufen wird,
DANN sollte sie `FALSE` zurückliefern,
UND das Testkonto sollte im Persistennce Context immer noch den Status "soll" besitzen.
- @Test: `setKontoGeschlossen_03()`
WENN ein Testkonto in der DB existiert,
UND das Testkonto den Status "gesperrt" besitzt,
UND die Methode `setKontoGeschlossen` mit der ID des Testkontos aufgerufen wird,
DANN sollte sie `FALSE` zurückliefern,
UND das Testkonto sollte im Persistennce Context immer noch den Status "gesperrt" besitzen.
- @Test: `setKontoGeschlossen_04()`
WENN ein Testkonto in der DB existiert,
UND das Testkonto den Status "geschlossen" besitzt,
UND die Methode `setKontoGeschlossen` mit der ID des Testkontos aufgerufen wird,
DANN sollte sie `FALSE` zurückliefern,
UND das Testkonto sollte im Persistennce Context immer noch den Status "geschlossen" besitzen.
- @Test: `setKontoGeschlossen_05()`
WENN ein Testkonto nicht in der DB existiert,
UND die Methode `setKontoGeschlossen` mit der ID des Testkontos aufgerufen wird,
DANN sollte sie `FALSE` zurückliefern.

2.1.2. IAntragStellenAdmin

- @Before: `angenommen()`
Angenommen der EntityManager wird korrekt geholt,

UND die Implementierung der IAntragStellenAdmin Schnittstelle wird als classUnderTest instanziert,

UND der EntityManager em wird per setEntityManager Methode der classUnderTest gesetzt,

UND die Transaktion von em wird gestartet,

UND die Daten der betreffenden Entitäten wurden im Persistence Context gelöscht.

- @After: amEnde()

Am Ende wird die Transaktion zurück gesetzt.

- @Test: antragStellen_000

WENN die Methode **antragStellen** mit einem TestAntrag aufgerufen wird,

UND die ID des TestAntrags gleich **null** ist,

UND der Typ des Antrags "so" ist,

DANN sollte sie **TRUE** zurückliefern,

UND der TestAntrag sollte im Persistennce Context existieren,

UND der Status des Testantrags sollte "neu" sein.

- @Test: antragStellen_010

WENN die Methode **antragStellen** mit einem TestAntrag aufgerufen wird,

UND die ID des TestAntrags gleich **null** ist,

UND der Typ des Antrags "sd" ist,

DANN sollte sie **TRUE** zurückliefern,

UND der TestAntrag sollte im Persistennce Context existieren,

UND der Status des Testantrags sollte "neu" sein.

- @Test: antragStellen_020

WENN die Methode **antragStellen** mit einem TestAntrag aufgerufen wird,

UND die ID des TestAntrags gleich **null** ist,

UND der Typ des Antrags "sb" ist,

DANN sollte sie **TRUE** zurückliefern,

UND der TestAntrag sollte im Persistennce Context existieren,

UND der Status des Testantrags sollte "neu" sein.

- @Test: antragStellen_030

WENN die Methode **antragStellen** mit einem TestAntrag aufgerufen wird,

UND die ID des TestAntrags gleich **null** ist,

UND der Typ des Antrags "ks" ist,

DANN sollte sie **FALSE** zurückliefern,

UND der Persistence Context wurde nicht verändert.

- @Test: antragStellen_040

WENN die Methode **antragStellen** mit einem TestAntrag aufgerufen wird,

UND die ID des TestAntrags **ungleich null** ist,

DANN sollte sie **FALSE** zurückliefern,

UND der Persistence Context wurde nicht verändert.

- @Test: getAdminListe_000

WENN x (x>0) Admins in der DB existieren,

UND die Methode `getAdminListe` aufgerufen wird,
DANN sollte sie die Liste mit diesen `x` Admins zurückliefern.

- @Test: `getAdminListe_01()`
WENN keine Admins in der DB existieren,
UND die Methode `getAdminListe` aufgerufen wird,
DANN sollte sie eine `leere Liste` zurückliefern.
- @Test: `geAdminByID_00()`
WENN ein TestAdmin bereits in der DB existiert,
UND die Methode `geAdminByID` mit der Id des TestAdmins aufgerufen wird,
DANN sollte sie den `TestAdmin` zurückliefern.
- @Test: `geAdminByID_01()`
WENN ein TestAdmin nicht in der DB existiert,
UND die Methode `geAdminByID` mit der Id des TestAdmins aufgerufen wird,
DANN sollte sie `NULL` zurückliefern.

2.2. Testspezifikation zur Komponente KundeDaten

2.2.1. ICRUDKunde

- @Before: `angenommen()`
Angenommen der EntityManager wird korrekt geholt,
UND die Implementierung der ICRUDKunde Schnittstelle wird als `classUnderTest` instanziert,
UND der EntityManager em wird per `setEntityManager` Methode der `classUnderTest` gesetzt,
UND die Transaktion von em wird gestartet,
UND die Daten der betreffenden Entitäten wurden im Persistence Context gelöscht.
- @After: `amEnde()`
Am Ende wird die Transaktion zurück gesetzt.
- @Test: `getKundeByID_00()`
WENN ein Testkunde bereits in der DB existiert,
UND die Methode `getKundeByID` mit der Id des Testkunden aufgerufen wird,
DANN sollte sie den `Testkunden` zurückliefern.
- @Test: `getKundeByID_01()`
WENN ein Testkunde nicht in der DB existiert,
UND die Methode `getKundeByID` mit der Id des Testkunden aufgerufen wird,
DANN sollte sie `NULL` zurückliefern.
- @Test: `getKundenListe_00()`
WENN $x (x>0)$ Kunden in der DB existieren,
UND die Methode `getKundenListe` aufgerufen wird,

DANN sollte sie eine Liste mit **x** Kunden zurückliefern.

- @Test: getKundenListe_01()

WENN keine Kunden in der DB existieren,
UND die Methode **getKundenListe** aufgerufen wird,
DANN sollte sie eine **leere Liste** zurückliefern.
- @Test: insertKunde_00()

WENN die Methode **insertKundet** mit einem Testkunden aufgerufen wird,
UND die ID des Testkunden gleich **null** ist,
DANN sollte sie **TRUE** zurückliefern,
UND der Testkunde sollte im Persistence Context existieren.
- @Test: insertKunde_01()

WENN die Methode **insertKunde** mit einem Testkunden aufgerufen wird,
UND die ID des Testkunden **ungleich null** ist,
DANN sollte sie **FALSE** zurückliefern,
UND der Persistence Context wurde nicht verändert.
- @Test: editKunde_00()

WENN ein Testkunde in der DB existiert,
UND die Methode **editKunde** mit einem veränderten Testkunden (aber gleicher ID) aufgerufen wird,
DANN sollte sie **TRUE** zurückliefern,
UND der Testkunde sollte im Persistence Context verändert sein.
- @Test: editKunde_01()

WENN ein Testkunde nicht in der DB existiert,
UND die Methode **editKunde** mit dem Testkunden aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern,
UND der Testkunde sollte nicht im Persistence Context existieren.
- @Test: deleteKunde_00()

WENN ein Testkunde in der DB existiert,
UND die Methode **deleteKunde** mit der ID des Testkunden aufgerufen wird,
DANN sollte sie **TRUE** zurückliefern,
UND der Testkunde sollte nicht mehr im Persistence Context existieren.
- @Test: deleteKunde_01()

WENN ein Testkundet nicht in der DB existiert,
UND die Methode **deleteKunde** mit der ID des Testkunden aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern.

2.3. Testspezifikation zur Komponente KontoDaten

2.3.1. ICRUDKonto

- @Before: angenommen()

Angenommen der EntityManager wird korrekt geholt,
UND die Implementierung der ICRUDKonto Schnittstelle wird als classUnderTest instanziert,
UND der EntityManager em wird per setEntityManager Methode der classUnderTest gesetzt,
UND die Transaktion von em wird gestartet,
UND die Daten der betreffenden Entitäten wurden im Persistence Context gelöscht.
- @After: amEnde()

Am Ende wird die Transaktion zurück gesetzt.
- @Test: getKontoByID_00()

WENN ein Testkonto bereits in der DB existiert,
UND die Methode **getKontoByID** mit der Id des Testkontos aufgerufen wird,
DANN sollte sie das **Testkonto** zurückliefern.
- @Test: getKontoByID_01()

WENN ein Testkonto nicht in der DB existiert,
UND die Methode **getKontoByID** mit der Id des Testkontos aufgerufen wird,
DANN sollte sie **NULL** zurückliefern.
- @Test: getKontoListeDesKunden_00()

WENN ein Testkunde bereits in der DB existiert,
UND für diesen Testkunden x ($x > 0$) Konten in der DB existieren,
UND die Methode **getKontoListeDesKunden** mit der ID des Testkunden aufgerufen wird,
DANN sollte sie die Liste seiner **x** Konten zurückliefern.
- @Test: getKontoListeDesKunden_01()

WENN ein Testkunde bereits in der DB existiert,
UND für diesen Testkunden keine Konten in der DB existieren,
UND die Methode **getKontoListeDesKunden** mit der ID des Testkunden aufgerufen wird,
DANN sollte sie eine **leere Liste** zurückliefern.
- @Test: getKontoListeDesKunden_02()

WENN ein Testkunde nicht in der DB existiert,
UND die Methode **getKontoListeDesKunden** mit der ID des Testkunden aufgerufen wird,
DANN sollte sie eine **leere Liste** zurückliefern.
- @Test: insertKonto_00()

WENN die Methode **insertKonto** mit einem Testkonto aufgerufen wird,
UND die ID des Testkontos gleich **null** ist,
DANN sollte sie **TRUE** zurückliefern,
UND das Testkonto sollte im Persistence Context existieren.
- @Test: insertKonto_01()

WENN die Methode **insertKonto** mit einem Testkonto aufgerufen wird,
UND die ID des Testkontos **ungleich null** ist,

DANN sollte sie **FALSE** zurückliefern,
UND der Persistence Context wurde nicht verändert.

- @Test: deleteKonto_00()

WENN ein Testkonto in der DB existiert,
UND die Methode **deleteKonto** mit der ID des Testkontos aufgerufen wird,
DANN sollte sie **TRUE** zurückliefern,
UND das Testkonto sollte nicht mehr im Persistence Context existieren.
- @Test: deleteKonto_01()

WENN ein Testkonto nicht in der DB existiert,
UND die Methode **deleteKonto** mit der ID des Testkontos aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern.
- @Test: geldEinzahlen_000

WENN ein Testkonto bereits in der DB existiert,
UND das Testkonto den Status "haben" besitzt,
UND die Methode **geldEinzahlen** mit der ID des Testkontos, und dem Betrag b aufgerufen wird,
DANN sollte sie **TRUE** zurückliefern.
UND der Kontostand des Testkontos ist um den Betrag b erhöht worden,
UND der Status des Testkontos in der DB den Wert "haben" besitzen,
UND es wurde ein neues Objekt der Klasse Zahlungsverkehr angelegt,
UND dieses Objekt besitzt die ID von Testkonto als Wert des Attributs **konto**,
UND dieses Objekt besitzt das korrekte Datum als Wert des Attributs **datum**,
UND dieses Objekt besitzt den Text "Einzahlung" als Wert des Attributs **erlaeuetung**,
UND dieses Objekt besitzt den Betrag b als Wert des Attributs **haben**,
UND dieses Objekt besitzt 0 als Wert des Attributs **soll**,
UND dieses Objekt besitzt **FALSE** als Wert des Attributs **ausgedruckt**.
- @Test: geldEinzahlen_01()

WENN ein Testkonto bereits in der DB existiert,
UND das Testkonto den Status "soll" besitzt,
UND für Betrag b gilt und Kontostand des Testkontos: $\text{Kontostand} + b \geq 0$,
UND die Methode **geldEinzahlen** mit der ID des Testkontos, und dem Betrag b aufgerufen wird,
DANN sollte sie **TRUE** zurückliefern.
UND der Kontostand des Testkontos ist um den Betrag b erhöht worden,
UND der Status des Testkontos in der DB den Wert "haben" besitzen,
UND es wurde ein neues Objekt der Klasse Zahlungsverkehr angelegt,
UND dieses Objekt besitzt die ID von Testkonto als Wert des Attributs **konto**,
UND dieses Objekt besitzt das korrekte Datum als Wert des Attributs **datum**,
UND dieses Objekt besitzt den Text "Einzahlung" als Wert des Attributs **erlaeuetung**,
UND dieses Objekt besitzt den Betrag b als Wert des Attributs **haben**,
UND dieses Objekt besitzt 0 als Wert des Attributs **soll**,
UND dieses Objekt besitzt **FALSE** als Wert des Attributs **ausgedruckt**.
- @Test: geldEinzahlen_020

WENN ein Testkonto bereits in der DB existiert,
UND das Testkonto den Status "soll" besitzt,
UND für Betrag b gilt und Kontostand des Testkontos: Kontostand + b < 0,
UND für Betrag b und Kontostand und Dispo des Testkontos gilt: Kontostand + dispo + b
 ≥ 0 ,
UND die Methode `geldEinzahlen` mit der ID des Testkontos, und dem Betrag b aufgerufen wird,
DANN sollte sie `TRUE` zurückliefern.
UND der Kontostand des Testkontos ist um den Betrag b erhöht worden,
UND der Status des Testkontos in der DB den Wert "soll" besitzen,
UND es wurde ein neues Objekt der Klasse Zahlungsverkehr angelegt,
UND dieses Objekt besitzt die ID von Testkonto als Wert des Attributs `konto`,
UND dieses Objekt besitzt das korrekte Datum als Wert des Attributs `datum`,
UND dieses Objekt besitzt den Text "Einzahlung" als Wert des Attributs `erlaeuetung`,
UND dieses Objekt besitzt den Betrag b als Wert des Attributs `haben`,
UND dieses Objekt besitzt 0 als Wert des Attributs `soll`,
UND dieses Objekt besitzt `FALSE` als Wert des Attributs `ausgedruckt`.

- @Test: `geldEinzahlen_03()`

WENN ein Testkonto bereits in der DB existiert,
UND das Testkonto den Status "gesperrt" besitzt,
UND für Betrag b und Kontostand des Testkontos gilt: Kontostand + b ≥ 0 ,
UND die Methode `geldEinzahlen` mit der ID des Testkontos, und dem Betrag b aufgerufen wird,
DANN sollte sie `TRUE` zurückliefern.
UND der Kontostand des Testkontos ist um den Betrag b erhöht worden,
UND der Status des Testkontos in der DB den Wert "haben" besitzen,
UND es wurde ein neues Objekt der Klasse Zahlungsverkehr angelegt,
UND dieses Objekt besitzt die ID von Testkonto als Wert des Attributs `konto`,
UND dieses Objekt besitzt das korrekte Datum als Wert des Attributs `datum`,
UND dieses Objekt besitzt den Text "Einzahlung" als Wert des Attributs `erlaeuetung`,
UND dieses Objekt besitzt den Betrag b als Wert des Attributs `haben`,
UND dieses Objekt besitzt 0 als Wert des Attributs `soll`,
UND dieses Objekt besitzt `FALSE` als Wert des Attributs `ausgedruckt`.

- @Test: `geldEinzahlen_04()`

WENN ein Testkonto bereits in der DB existiert,
UND das Testkonto den Status "gesperrt" besitzt,
UND für Betrag b und Kontostand des Testkontos gilt: Kontostand + b < 0,
UND für Betrag b und Kontostand und Dispo des Testkontos gilt: Kontostand + dispo + b
 ≥ 0 ,
UND die Methode `geldEinzahlen` mit der ID des Testkontos, und dem Betrag b aufgerufen wird,
DANN sollte sie `TRUE` zurückliefern.
UND der Kontostand des Testkontos ist um den Betrag b erhöht worden,
UND der Status des Testkontos in der DB den Wert "soll" besitzen,

UND es wurde ein neues Objekt der Klasse Zahlungsverkehr angelegt,
UND dieses Objekt besitzt die ID von Testkonto als Wert des Attributs **konto**,
UND dieses Objekt besitzt das korrekte Datum als Wert des Attributs **datum**,
UND dieses Objekt besitzt den Text "Einzahlung" als Wert des Attributs **erlaeueterung**,
UND dieses Objekt besitzt den Betrag b als Wert des Attributs **haben**,
UND dieses Objekt besitzt 0 als Wert des Attributs **soll**,
UND dieses Objekt besitzt **FALSE** als Wert des Attributs **ausgedruckt**.

- @Test: geldEinzahlen_05()

WENN ein Testkonto bereits in der DB existiert,
UND das Testkonto den Status "gesperrt" besitzt,
UND für Betrag b und Kontostand des Testkontos gilt: Kontostand + b < 0,
UND für Betrag b und Kontostand und Dispo des Testkontos gilt: Kontostand + dispo + b < 0,
UND die Methode **geldEinzahlen** mit der ID des Testkontos, und dem Betrag b aufgerufen wird,
DANN sollte sie **TRUE** zurückliefern.
UND der Kontostand des Testkontos ist um den Betrag b erhöht worden,
UND der Status des Testkontos in der DB den Wert "gesperrt" besitzen,
UND es wurde ein neues Objekt der Klasse Zahlungsverkehr angelegt,
UND dieses Objekt besitzt die ID von Testkonto als Wert des Attributs **konto**,
UND dieses Objekt besitzt das korrekte Datum als Wert des Attributs **datum**,
UND dieses Objekt besitzt den Text "Einzahlung" als Wert des Attributs **erlaeueterung**,
UND dieses Objekt besitzt den Betrag b als Wert des Attributs **haben**,
UND dieses Objekt besitzt 0 als Wert des Attributs **soll**,
UND dieses Objekt besitzt **FALSE** als Wert des Attributs **ausgedruckt**.

- @Test: geldEinzahlen_06()

WENN ein Testkonto nicht in der DB existiert,
UND die Methode **geldEinzahlen** mit der ID des Testkontos aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern.

- @Test: geldAbheben_00()

WENN ein Testkonto bereits in der DB existiert,
UND dieses Testkonto den Zustand "haben" besitzt,
UND für Betrag b und Kontostand des Testkontos gilt: Kontostand - b >= 0,
UND die Methode **geldAbheben** mit der ID des Testkontos und dem Betrag b aufgerufen wird,
DANN sollte sie **TRUE** zurückliefern.
UND der Kontostand des Testkontos ist um den Betrag b verringert worden,
UND der Status des Testkontos in der DB den Wert "haben" besitzen,
UND es wurde ein neues Objekt der Klasse Zahlungsverkehr angelegt,
UND dieses Objekt besitzt die ID von Testkonto als Wert des Attributs **konto**,
UND dieses Objekt besitzt das korrekte Datum als Wert des Attributs **datum**,
UND dieses Objekt besitzt den Text "Auszahlung" als Wert des Attributs **erlaeueterung**,
UND dieses Objekt besitzt 0 als Wert des Attributs **haben**,
UND dieses Objekt besitzt den Betrag b als Wert des Attributs **soll**,

UND dieses Objekt besitzt **FALSE** als Wert des Attributs **ausgedruckt**.

- @Test: geldAbheben_01()

WENN ein Testkonto bereits in der DB existiert,

UND dieses Testkonto den Zustand "haben" besitzt,

UND für Betrag b und Kontostand des Testkontos gilt: Kontostand - b < 0,

UND für Betrag b und Kontostand und Dispo des Testkontos gilt: Kontostand + dispo - b
>= 0,

UND die Methode **geldAbheben** mit der ID des Testkontos und dem Betrag b aufgerufen wird,

DANN sollte sie **TRUE** zurückliefern.

UND der Kontostand des Testkontos ist um den Betrag b verringert worden,

UND der Status des Testkontos in der DB den Wert "soll" besitzen,

UND es wurde ein neues Objekt der Klasse Zahlungsverkehr angelegt,

UND dieses Objekt besitzt die ID von Testkonto als Wert des Attributs **konto**,

UND dieses Objekt besitzt das korrekte Datum als Wert des Attributs **datum**,

UND dieses Objekt besitzt den Text "Auszahlung" als Wert des Attributs **erlaeueterung**,

UND dieses Objekt besitzt 0 als Wert des Attributs **haben**,

UND dieses Objekt besitzt den Betrag b als Wert des Attributs **soll**,

UND dieses Objekt besitzt **FALSE** als Wert des Attributs **ausgedruckt**.

- @Test: geldAbheben_02()

WENN ein Testkonto bereits in der DB existiert,

UND dieses Testkonto den Zustand "haben" besitzt,

UND für Betrag b und Kontostand des Testkontos gilt: Kontostand - b < 0,

UND für Betrag b und Kontostand und Dispo des Testkontos gilt: Kontostand + dispo - b < 0,

UND die Methode **geldAbheben** mit der ID des Testkontos und dem Betrag b aufgerufen wird,

DANN sollte sie **TRUE** zurückliefern.

UND der Kontostand des Testkontos ist um den Betrag b verringert worden,

UND der Status des Testkontos in der DB den Wert "gesperrt" besitzen,

UND es wurde ein neues Objekt der Klasse Zahlungsverkehr angelegt,

UND dieses Objekt besitzt die ID von Testkonto als Wert des Attributs **konto**,

UND dieses Objekt besitzt das korrekte Datum als Wert des Attributs **datum**,

UND dieses Objekt besitzt den Text "Auszahlung" als Wert des Attributs **erlaeueterung**,

UND dieses Objekt besitzt 0 als Wert des Attributs **haben**,

UND dieses Objekt besitzt den Betrag b als Wert des Attributs **soll**,

UND dieses Objekt besitzt **FALSE** als Wert des Attributs **ausgedruckt**.

- @Test: geldAbheben_03()

WENN ein Testkonto bereits in der DB existiert,

UND dieses Testkonto den Zustand "soll" besitzt,

UND für Betrag b und Kontostand des Testkontos gilt: Kontostand - b < 0,

UND für Betrag b und Kontostand und Dispo des Testkontos gilt: Kontostand + dispo - b
>= 0,

UND die Methode **geldAbheben** mit der ID des Testkontos und dem Betrag b aufgerufen

wird,

DANN sollte sie **TRUE** zurückliefern.

UND der Kontostand des Testkontos ist um den Betrag b verringert worden,

UND der Status des Testkontos in der DB den Wert "soll" besitzen,

UND es wurde ein neues Objekt der Klasse Zahlungsverkehr angelegt,

UND dieses Objekt besitzt die ID von Testkonto als Wert des Attributs **konto**,

UND dieses Objekt besitzt das korrekte Datum als Wert des Attributs **datum**,

UND dieses Objekt besitzt den Text "Auszahlung" als Wert des Attributs **erlaeueterung**,

UND dieses Objekt besitzt 0 als Wert des Attributs **haben**,

UND dieses Objekt besitzt den Betrag b als Wert des Attributs **soll**,

UND dieses Objekt besitzt **FALSE** als Wert des Attributs **ausgedruckt**.

- @Test: **geldAbheben_04()**

WENN ein Testkonto bereits in der DB existiert,

UND dieses Testkonto den Zustand "soll" besitzt,

UND für Betrag b und Kontostand des Testkontos gilt: Kontostand - b < 0,

UND für Betrag b und Kontostand und Dispo des Testkontos gilt: Kontostand + dispo - b < 0,

UND die Methode **geldAbheben** mit der ID des Testkontos und dem Betrag b aufgerufen wird,

DANN sollte sie **TRUE** zurückliefern.

UND der Kontostand des Testkontos ist um den Betrag b verringert worden,

UND der Status des Testkontos in der DB den Wert "gesperrt" besitzen,

UND es wurde ein neues Objekt der Klasse Zahlungsverkehr angelegt,

UND dieses Objekt besitzt die ID von Testkonto als Wert des Attributs **konto**,

UND dieses Objekt besitzt das korrekte Datum als Wert des Attributs **datum**,

UND dieses Objekt besitzt den Text "Auszahlung" als Wert des Attributs **erlaeueterung**,

UND dieses Objekt besitzt 0 als Wert des Attributs **haben**,

UND dieses Objekt besitzt den Betrag b als Wert des Attributs **soll**,

UND dieses Objekt besitzt **FALSE** als Wert des Attributs **ausgedruckt**.

- @Test: **geldAbheben_05()**

WENN ein Testkonto bereits in der DB existiert,

UND dieses Testkonto den Zustand "gesperrt" besitzt,

UND die Methode **geldAbheben** mit der ID des Testkontos und dem Betrag b aufgerufen wird,

aufgerufen wird,

DANN sollte sie **FALSE** zurückliefern, **UND** der Kontostand des Testkontos wurde nicht verändert.

- @Test: **geldAbheben_06()**

WENN ein Testkonto nicht in der DB existiert,

UND die Methode **geldAbheben** mit der ID des Testkontos aufgerufen wird,

DANN sollte sie **FALSE** zurückliefern.

2.3.2. IKontoService

- @Before: angenommen()

Angenommen der EntityManager wird korrekt geholt,
UND die Implementierung der IKontoService Schnittstelle wird als classUnderTest instanziert,
UND der EntityManager em wird per setEntityManager Methode der classUnderTest gesetzt,
UND die Transaktion von em wird gestartet,
UND die Daten der betreffenden Entitäten wurden im Persistence Context gelöscht.
- @After: amEnde()

Am Ende wird die Transaktion zurück gesetzt.
- @Test: ueberweisungErfassen_00()

WENN die Methode ueberweisungErfassen mit einer TestUeberweisung aufgerufen wird,
UND die ID der TestUeberweisung gleich **null** ist,
UND das Attribut **vonkonto** der TestUeberweisung auf ein in der DB existierendes Konto verweist,
UND das Attribut **zukonto** der TestUeberweisung auf ein in der DB existierendes Konto verweist,
DANN sollte sie **TRUE** zurückliefern,
UND die TestUeberweisung sollte im Persistennce Context existieren, **UND** die TestUeberweisung sollte den Status wartet (wt) besitzen.
- @Test: ueberweisungErfassen_01()

WENN die Methode ueberweisungErfassen mit einer TestUeberweisung aufgerufen wird,
UND die ID der TestUeberweisung **ungleich null** ist,
DANN sollte sie **FALSE** zurückliefern,
UND die TestUeberweisung sollte nicht im Persistence Context existieren.
- @Test: ueberweisungErfassen_02()

WENN die Methode ueberweisungErfassen mit einer TestUeberweisung aufgerufen wird,
UND das Attribut **vonkonto** der TestUeberweisung auf ein in der DB nicht existierendes Konto verweist,
DANN sollte sie **FALSE** zurückliefern,
UND die TestUeberweisung sollte nicht im Persistence Context existieren.
- @Test: ueberweisungErfassen_03()

WENN die Methode ueberweisungErfassen mit einer TestUeberweisung aufgerufen wird,
UND das Attribut **zukonto** der TestUeberweisung auf ein in der DB nicht existierendes Konto verweist,
DANN sollte sie **FALSE** zurückliefern,
UND die TestUeberweisung sollte nicht im Persistence Context existieren.
- @Test: getStornierbareUeberweisungen_00()

WENN ein Testkunde bereits in der DB existiert,
UND für diesen Testkunden x ($x > 0$) Überweisungen in der DB existieren, die den Status wartet (wt) haben,

UND die Methode `getStornierbareUeberweisungen` mit der ID des Testkunden aufgerufen wird,

DANN sollte sie die Liste mit **x** Überweisungen zurückliefern.

- @Test: `getStornierbareUeberweisungen_01()`

WENN ein Testkunde bereits in der DB existiert,
UND für diesen Testkunden keine Überweisungen in der DB existieren, die den Status wartet (wt) haben,
UND die Methode `getStornierbareUeberweisungen` mit der ID des Testkunden aufgerufen wird,
DANN sollte sie eine **leere Liste** zurückliefern.
- @Test: `getStornierbareUeberweisungen_02()`

WENN ein Testkunde nicht in der DB existiert,
UND die Methode `getStornierbareUeberweisungen` mit der ID des Testkunden aufgerufen wird,
DANN sollte sie eine **leere Liste** zurückliefern.
- @Test: `ueberweisungStornieren_00()`

WENN eine TestUeberweisung bereits in der DB existiert,
UND die TestUeberweisung den Status "wartet" besitzt,
UND die Methode `ueberweisungStornieren` mit der Id der TestUeberweisung aufgerufen wird,
DANN sollte sie **TRUE** zurückliefern,
UND der Status der **TestUeberweisung** auf "storniert" gesetzt sein.
- @Test: `ueberweisungStornieren_01()`

WENN eine TestUeberweisung bereits in der DB existiert,
UND die TestUeberweisung den Status "storniert" besitzt,
UND die Methode `ueberweisungStornieren` mit der Id der TestUeberweisung aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern,
UND die TestUeberweisung wurde nicht verändert.
- @Test: `ueberweisungStornieren_02()`

WENN ein TestUeberweisung nicht in der DB existiert,
UND die Methode `ueberweisungStornieren` mit der Id der TestUeberweisung aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern.
- @Test: `getKontoauszug_00()`

WENN ein Testkonto bereits in der DB existiert,
UND für dieses Testkonto x ($x > 0$) Zahlungsverkehre in der DB existieren, die innerhalb des Datumsintervalls liegen,
UND die Methode `getKontoauszug` mit der Id des Testkontos aufgerufen wird,
DANN sollte sie die Liste mit **x** Zahlungsverkehren zurückliefern.
- @Test: `getKontoauszug_01()`

WENN ein Testkonto bereits in der DB existiert,

UND für dieses Testkonto keine Zahlungsverkehre in der DB existieren, die innerhalb des Datumsintervalls liegen,

UND die Methode `getKontoauszug` mit der Id des Testkontos aufgerufen wird,

DANN sollte sie die **leere Liste** zurückliefern.

- @Test: `getKontoauszug_020`

WENN ein Testkonto nicht in der DB existiert,

UND die Methode `getKontoauszug` mit der Id des Testkontos aufgerufen wird,

DANN sollte sie die **leere Liste** zurückliefern.
- @Test: `getKontoauszugDrucken_00()`

WENN ein Testkonto bereits in der DB existiert,

UND für dieses Testkonto x ($x>0$) Zahlungsverkehre in der DB existieren, die als nicht ausgedruckt markiert sind,

UND die Methode `getKontoauszugDrucken` mit der Id des Testkontos aufgerufen wird,

DANN sollte sie die Liste mit diesen x Zahlungsverkehren zurückliefern,

UND diese x Zahlungsverkehre sind dann als ausgedruckt in der DB markiert.
- @Test: `getKontoauszugDrucken_01()`

WENN ein Testkonto bereits in der DB existiert,

UND für dieses Testkonto keine Zahlungsverkehre in der DB existieren, die als nicht ausgedruckt markiert sind,

UND die Methode `getKontoauszugDrucken` mit der Id des Testkontos aufgerufen wird,

DANN sollte sie die **leere Liste** zurückliefern.
- @Test: `getKontoauszugDrucken_02()`

WENN ein Testkonto nicht in der DB existiert,

UND die Methode `getKontoauszugDrucken` mit der Id des Testkontos aufgerufen wird,

DANN sollte sie die **leere Liste** zurückliefern.
- @Test: `getKontoListe_00()`

WENN ein Testkunde bereits in der DB existiert,

UND für diesen Testkunden x ($x>0$) Konten in der DB existieren,

UND die Methode `getKontoListe` mit der ID des Testkunden aufgerufen wird,

DANN sollte sie die Liste seiner x Konten zurückliefern.
- @Test: `getKontoListe_01()`

WENN ein Testkunde bereits in der DB existiert,

UND für diesen Testkunden keine Konten in der DB existieren,

UND die Methode `getKontoListe` mit der ID des Testkunden aufgerufen wird,

DANN sollte sie eine **leere Liste** zurückliefern.
- @Test: `getKontoListe_02()`

WENN ein Testkunde nicht in der DB existiert,

UND die Methode `getKontoListe` mit der ID des Testkunden aufgerufen wird,

DANN sollte sie eine **leere Liste** zurückliefern.
- @Test: `getKontoById_00()`

WENN ein Testkonto bereits in der DB existiert,

UND die Methode `getKontoById` mit der Id des Testkontos aufgerufen wird,

DANN sollte sie das **Testkonto** zurückliefern.

- @Test: getKontoById_01()

WENN ein Testkonto nicht in der DB existiert,
UND die Methode **getKontoById** mit der Id des Testkontos aufgerufen wird,
DANN sollte sie **NULL** zurückliefern.
- @Test: kontoAktualisieren_00()

WENN ein Testkonto bereits in der DB existiert,
UND das Testkonto den Status "haben" besitzt,
UND das Testkonto einen Kontostand < 0 besitzt,
UND für das Testkonto Kontostand + dispo < 0 gilt,
UND die Methode **kontoAktualisieren** mit der ID des Testkontos aufgerufen wird,
DANN sollte danach der Status des Testkontos in der DB den Wert "gesperrt" besitzen.
- @Test: kontoAktualisieren_01()

WENN ein Testkonto bereits in der DB existiert,
UND das Testkonto den Status "haben" besitzt,
UND das Testkonto einen Kontostand < 0 besitzt,
UND für das Testkonto Kontostand + dispo ≥ 0 gilt,
UND die Methode **kontoAktualisieren** mit der ID des Testkontos aufgerufen wird,
DANN sollte danach der Status des Testkontos in der DB den Wert "soll" besitzen.
- @Test: kontoAktualisieren_02()

WENN ein Testkonto bereits in der DB existiert,
UND das Testkonto den Status "haben" besitzt,
UND das Testkonto einen Kontostand ≥ 0 besitzt,
UND die Methode **kontoAktualisieren** mit der ID des Testkontos aufgerufen wird,
DANN sollte danach der Status des Testkontos in der DB den Wert "haben" besitzen.
- @Test: kontoAktualisieren_03()

WENN ein Testkonto bereits in der DB existiert,
UND das Testkonto den Status "soll" besitzt,
UND das Testkonto einen Kontostand < 0 besitzt,
UND für das Testkonto Kontostand + dispo < 0 gilt,
UND die Methode **kontoAktualisieren** mit der ID des Testkontos aufgerufen wird,
DANN sollte danach der Status des Testkontos in der DB den Wert "gesperrt" besitzen.
- @Test: kontoAktualisieren_04()

WENN ein Testkonto bereits in der DB existiert,
UND das Testkonto den Status soll (s) besitzt,
UND das Testkonto einen Kontostand < 0 besitzt,
UND für das Testkonto Kontostand + dispo ≥ 0 gilt,
UND die Methode **kontoAktualisieren** mit der ID des Testkontos aufgerufen wird,
DANN sollte danach der Status des Testkontos in der DB den Wert soll (s) besitzen.
- @Test: kontoAktualisieren_05()

WENN ein Testkonto bereits in der DB existiert,
UND das Testkonto den Status "soll" besitzt,

UND das Testkonto einen Kontostand ≥ 0 besitzt,
UND die Methode **kontoAktualisieren** mit der ID des Testkontos aufgerufen wird,
DANN sollte danach der Status des Testkontos in der DB den Wert "haben" besitzen.

- @Test: kontoAktualisieren_06()

WENN ein Testkonto bereits in der DB existiert,
UND das Testkonto den Status "gesperrt" besitzt,
UND das Testkonto einen Kontostand < 0 besitzt,
UND für das Testkonto Kontostand + dispo < 0 gilt,
UND die Methode **kontoAktualisieren** mit der ID des Testkontos aufgerufen wird,
DANN sollte danach der Status des Testkontos in der DB den Wert "gesperrt" besitzen.
- @Test: kontoAktualisieren_07()

WENN ein Testkonto bereits in der DB existiert,
UND das Testkonto den Status "gesperrt" besitzt,
UND das Testkonto einen Kontostand < 0 besitzt,
UND für das Testkonto Kontostand + dispo ≥ 0 gilt,
UND die Methode **kontoAktualisieren** mit der ID des Testkontos aufgerufen wird,
DANN sollte danach der Status des Testkontos in der DB den Wert "soll" besitzen.
- @Test: kontoAktualisieren_08()

WENN ein Testkonto bereits in der DB existiert,
UND das Testkonto den Status "gesperrt" besitzt,
UND das Testkonto einen Kontostand ≥ 0 besitzt,
UND die Methode **kontoAktualisieren** mit der ID des Testkontos aufgerufen wird,
DANN sollte danach der Status des Testkontos in der DB den Wert "haben" besitzen. *
- @Test: ueberweisungAusfuehren_00()

WENN eine TestUeberweisung bereits in der DB existiert,
UND die TestUeberweisung den Status "wartet" besitzt,
UND das über das Attribut **vonkonto** referenzierte Konto den Status "haben" besitzt,
UND die Methode **ueberweisungAusfuehren** mit der Id der TestUeberweisung aufgerufen wird,
DANN sollte sie **TRUE** zurückliefern,
UND der Status der **TestUeberweisung** sollte auf "ueberwiesen" gesetzt sein,
UND der Kontostand des zu belastenden Kontos sollte um den Betrag der TestUeberweisung reduziert sein,
UND der Kontostand des gutgeschriebenen Kontos sollte um den Betrag der TestUeberweisung erhöht sein,
UND es wurde ein neues Objekt der Klasse Zahlungsverkehr für das zu belastende Konto angelegt,
UND dieses Objekt besitzt die ID von **vonkonto** als Wert des Attributs **konto**,
UND dieses Objekt besitzt das korrekte Datum als Wert des Attributs **datum**,
UND dieses Objekt besitzt den Text von **verwendungszweck** als Wert des Attributs **erlaeuetzung**,
UND dieses Objekt besitzt 0 als Wert des Attributs **haben**,
UND dieses Objekt besitzt den Betrag der TestUeberweisung als Wert des Attributs **soll**,

UND dieses Objekt besitzt FALSE als Wert des Attributs **ausgedruckt**, UND es wurde ein neues Objekt der Klasse Zahlungsverkehr für das zu gutgeschriebene Konto angelegt,
 UND dieses Objekt besitzt die ID von **zukonto** als Wert des Attributs **konto**,
 UND dieses Objekt besitzt das korrekte Datum als Wert des Attributs **datum**,
 UND dieses Objekt besitzt den Text von **verwendungszweck** als Wert des Attributs **erlauerterung**,
 UND dieses Objekt besitzt den Betrag der TestUeberweisung als Wert des Attributs **haben**,
 UND dieses Objekt besitzt 0 als Wert des Attributs **soll**,
 UND dieses Objekt besitzt FALSE als Wert des Attributs **ausgedruckt**.

- @Test: ueberweisungAusfuehren_01()

WENN eine TestUeberweisung bereits in der DB existiert,
 UND die TestUeberweisung den Status "wartet" besitzt,
 UND das über das Attribut **vonkonto** referenzierte Konto den Status "soll" besitzt,
 UND die Methode **ueberweisungAusfuehren** mit der Id der TestUeberweisung aufgerufen wird,
 DANN sollte sie TRUE zurückliefern,
 UND der Status der **TestUeberweisung** sollte auf "ueberwiesen" gesetzt sein,
 UND der Kontostand des zu belastenden Kontos sollte um den Betrag der TestUeberweisung reduziert sein,
 UND der Kontostand des gutgeschriebenen Kontos sollte um den Betrag der TestUeberweisung erhöht sein, UND es wurde ein neues Objekt der Klasse Zahlungsverkehr für das zu belastende Konto angelegt,
 UND dieses Objekt besitzt die ID von **vonkonto** als Wert des Attributs **konto**,
 UND dieses Objekt besitzt das korrekte Datum als Wert des Attributs **datum**,
 UND dieses Objekt besitzt den Text von **verwendungszweck** als Wert des Attributs **erlauerterung**,
 UND dieses Objekt besitzt 0 als Wert des Attributs **haben**,
 UND dieses Objekt besitzt den Betrag der TestUeberweisung als Wert des Attributs **soll**,
 UND dieses Objekt besitzt FALSE als Wert des Attributs **ausgedruckt**, UND es wurde ein neues Objekt der Klasse Zahlungsverkehr für das zu gutgeschriebene Konto angelegt,
 UND dieses Objekt besitzt die ID von **zukonto** als Wert des Attributs **konto**,
 UND dieses Objekt besitzt das korrekte Datum als Wert des Attributs **datum**,
 UND dieses Objekt besitzt den Text von **verwendungszweck** als Wert des Attributs **erlauerterung**,
 UND dieses Objekt besitzt den Betrag der TestUeberweisung als Wert des Attributs **haben**,
 UND dieses Objekt besitzt 0 als Wert des Attributs **soll**,
 UND dieses Objekt besitzt FALSE als Wert des Attributs **ausgedruckt**.
- @Test: ueberweisungAusfuehren_02()

WENN eine TestUeberweisung bereits in der DB existiert,
 UND die TestUeberweisung den Status "wartet" besitzt,
 UND das über das Attribut **vonkonto** referenzierte Konto den Status "gesperrt" besitzt,
 UND die Methode **ueberweisungAusfuehren** mit der Id der TestUeberweisung aufgerufen wird,
 DANN sollte sie FALSE zurückliefern,

UND der Status der **TestUeberweisung** sollte auf "nicht ueberweisbar" gesetzt sein,
UND der Kontostand des zu belastenden Kontos sollte nicht verändert worden sein,
UND der Kontostand des gutgeschriebenen Kontos sollte nicht verändert worden sein.

- @Test: kontoFreischalten_00()

WENN ein TestKonto bereits in der DB existiert,
UND das TestKonto den Status "init" besitzt,
UND die Methode **kontoFreischalten** mit der Id der TestKontos aufgerufen wird,
DANN sollte sie **TRUE** zurückliefern,
UND der Status des TestKonto sollte auf "haben" gesetzt sein.
- @Test: kontoFreischalten_01()

WENN ein TestKonto bereits in der DB existiert,
UND das TestKonto nicht den Status "init" besitzt,
UND die Methode **kontoFreischalten** mit der Id der TestKontos aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern,
UND der Status des TestKonto sollte nicht verändert worden sein.
- @Test: kontoFreischalten_02()

WENN ein TestKonto nicht in der DB existiert,
UND die Methode **kontoFreischalten** mit der Id der TestKontos aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern.
- @Test: getNichtFreigeschalteteKonten_00()

WENN x ($x > 0$) Konten mit dem Status "init" in der DB existieren,
UND die Methode **getNichtFreigeschalteteKonten** aufgerufen wird,
DANN sollte sie die Liste der **x** Konten zurückliefern.
- @Test: getNichtFreigeschalteteKonten_01()

WENN keine Konten mit dem Status "init" in der DB existieren,
UND die Methode **getNichtFreigeschalteteKonten** aufgerufen wird,
DANN sollte sie eine **leere Liste** zurückliefern.

2.3.3. ISonderKontoService

- @Before: angenommen()

Angenommen der EntityManager wird korrekt geholt,
UND die Implementierung der ISonderKontoService Schnittstelle wird als classUnderTest instanziert,
UND der EntityManager em wird per setEntityManager Methode der classUnderTest gesetzt,
UND die Transaktion von em wird gestartet,
UND die Daten der betreffenden Entitäten wurden im Persistence Context gelöscht.
- @After: amEnde()

Am Ende wird die Transaktion zurück gesetzt.
- @Test: getKontoByID_00()

WENN ein Testkonto bereits in der DB existiert,

UND die Methode `getKontoByID` mit der Id des Testkontos aufgerufen wird,
DANN sollte sie das `Testkonto` zurückliefern.

- @Test: `getKontoByID_01()`

WENN ein Testkonto nicht in der DB existiert,
UND die Methode `getKontoByID` mit der Id des Testkontos aufgerufen wird,
DANN sollte sie `NULL` zurückliefern.
- @Test: `getUeberzogeneKontos_00()`

WENN $x > 0$ Konten mit Status "soll" oder Status "gesperrt" in der DB existieren,
UND die Methode `getUeberzogeneKontos` aufgerufen wird,
DANN sollte sie die Liste dieser x Konten zurückliefern.
- @Test: `getUeberzogeneKontos_01()`

WENN keine Konten mit Status "soll" oder Status "gesperrt" in der DB existieren,
UND die Methode `getUeberzogeneKontos` aufgerufen wird,
DANN sollte sie eine `leere Liste` zurückliefern.
- @Test: `neuesDispoSetzen_00()`

WENN eine TestKonto bereits in der DB existiert,
UND das TestKonto den Status "haben" besitzt,
UND die Methode `neuesDispoSetzen` mit der Id des TestKontos und dem neuen Dispo d aufgerufen wird,
DANN sollte sie `TRUE` zurückliefern,
UND der Status des `TestKontos` sollte auf "haben" gesetzt sein,
UND der Dispo des TestKontos sollte auf den Wert d gesetzt sein.
- @Test: `neuesDispoSetzen_01()`

WENN eine TestKonto bereits in der DB existiert,
UND das TestKonto den Status "soll" besitzt,
UND für TestKonto gilt: kontostand + d < 0,
UND die Methode `neuesDispoSetzen` mit der Id des TestKontos und dem neuen Dispo d aufgerufen wird,
DANN sollte sie `TRUE` zurückliefern,
UND der Status des `TestKontos` sollte auf "gesperrt" gesetzt sein,
UND der Dispo des TestKontos sollte auf den Wert d gesetzt sein.
- @Test: `neuesDispoSetzen_02()`

WENN eine TestKonto bereits in der DB existiert,
UND das TestKonto den Status "gesperrt" besitzt,
UND für TestKonto gilt: kontostand + d > 0,
UND die Methode `neuesDispoSetzen` mit der Id des TestKontos und dem neuen Dispo d aufgerufen wird,
DANN sollte sie `TRUE` zurückliefern,
UND der Status des `TestKontos` sollte auf "soll" gesetzt sein,
UND der Dispo des TestKontos sollte auf den Wert d gesetzt sein.
- @Test: `neuesDispoSetzen_03()`

WENN eine TestKonto nicht in der DB existiert,

UND die Methode **neuesDispoSetzen** mit der Id des TestKontos und dem neuen Dispo d aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern.

- @Test: kontoAktualisieren_00()

WENN ein Testkonto bereits in der DB existiert,
UND das Testkonto den Status "haben" besitzt,
UND das Testkonto einen Kontostand < 0 besitzt,
UND für das Testkonto Kontostand + dispo < 0 gilt,
UND die Methode **kontoAktualisieren** mit der ID des Testkontos aufgerufen wird,
DANN sollte danach der Status des Testkontos in der DB den Wert "gesperrt" besitzen.
- @Test: kontoAktualisieren_01()

WENN ein Testkonto bereits in der DB existiert,
UND das Testkonto den Status "haben" besitzt,
UND das Testkonto einen Kontostand < 0 besitzt,
UND für das Testkonto Kontostand + dispo >= 0 gilt,
UND die Methode **kontoAktualisieren** mit der ID des Testkontos aufgerufen wird,
DANN sollte danach der Status des Testkontos in der DB den Wert "soll" besitzen.
- @Test: kontoAktualisieren_02()

WENN ein Testkonto bereits in der DB existiert,
UND das Testkonto den Status "haben" besitzt,
UND das Testkonto einen Kontostand >= 0 besitzt,
UND die Methode **kontoAktualisieren** mit der ID des Testkontos aufgerufen wird,
DANN sollte danach der Status des Testkontos in der DB den Wert "haben" besitzen.
- @Test: kontoAktualisieren_03()

WENN ein Testkonto bereits in der DB existiert,
UND das Testkonto den Status "soll" besitzt,
UND das Testkonto einen Kontostand < 0 besitzt,
UND für das Testkonto Kontostand + dispo < 0 gilt,
UND die Methode **kontoAktualisieren** mit der ID des Testkontos aufgerufen wird,
DANN sollte danach der Status des Testkontos in der DB den Wert "gesperrt" besitzen.
- @Test: kontoAktualisieren_04()

WENN ein Testkonto bereits in der DB existiert,
UND das Testkonto den Status soll (s) besitzt,
UND das Testkonto einen Kontostand < 0 besitzt,
UND für das Testkonto Kontostand + dispo >= 0 gilt,
UND die Methode **kontoAktualisieren** mit der ID des Testkontos aufgerufen wird,
DANN sollte danach der Status des Testkontos in der DB den Wert soll (s) besitzen.
- @Test: kontoAktualisieren_05()

WENN ein Testkonto bereits in der DB existiert,
UND das Testkonto den Status "soll" besitzt,
UND das Testkonto einen Kontostand >= 0 besitzt,
UND die Methode **kontoAktualisieren** mit der ID des Testkontos aufgerufen wird,
DANN sollte danach der Status des Testkontos in der DB den Wert "haben" besitzen.

- @Test: kontoAktualisieren_06()

WENN ein Testkonto bereits in der DB existiert,
UND das Testkonto den Status "gesperrt" besitzt,
UND das Testkonto einen Kontostand < 0 besitzt,
UND für das Testkonto Kontostand + dispo < 0 gilt,
UND die Methode **kontoAktualisieren** mit der ID des Testkontos aufgerufen wird,
DANN sollte danach der Status des Testkontos in der DB den Wert "gesperrt" besitzen.
- @Test: kontoAktualisieren_07()

WENN ein Testkonto bereits in der DB existiert,
UND das Testkonto den Status "gesperrt" besitzt,
UND das Testkonto einen Kontostand < 0 besitzt,
UND für das Testkonto Kontostand + dispo >= 0 gilt,
UND die Methode **kontoAktualisieren** mit der ID des Testkontos aufgerufen wird,
DANN sollte danach der Status des Testkontos in der DB den Wert "soll" besitzen.
- @Test: kontoAktualisieren_08()

WENN ein Testkonto bereits in der DB existiert,
UND das Testkonto den Status "gesperrt" besitzt,
UND das Testkonto einen Kontostand >= 0 besitzt,
UND die Methode **kontoAktualisieren** mit der ID des Testkontos aufgerufen wird,
DANN sollte danach der Status des Testkontos in der DB den Wert "haben" besitzen.

2.4. Testspezifikation zur Komponente ManagerDaten

2.4.1. ICRUDManager

- @Before: angenommen()

Angenommen der EntityManager wird korrekt geholt,
UND die Implementierung der ICRUDManager Schnittstelle wird als classUnderTest instanziiert,
UND der EntityManager em wird per setEntityManager Methode der classUnderTest gesetzt,
UND die Transaktion von em wird gestartet,
UND die Daten der betreffenden Entitäten wurden im Persistence Context gelöscht.
- @After: amEnde()

Am Ende wird die Transaktion zurück gesetzt.
- @Test: getManagerByID_00()

WENN ein TestManager bereits in der DB existiert,
UND die Methode **getManagerByID** mit der Id des TestManagers aufgerufen wird,
DANN sollte sie den **TestManager** zurückliefern.
- @Test: getManagerByID_01()

WENN ein TestManager nicht in der DB existiert,

UND die Methode `getManagerByID` mit der Id des TestManagers aufgerufen wird,
DANN sollte sie `NULL` zurückliefern.

- @Test: `getManagerListe_00()`
WENN x ($x > 0$) Manager in der DB existieren,
UND die Methode `getManagerListe` aufgerufen wird,
DANN sollte sie eine Liste mit x Managern zurückliefern.
- @Test: `getManagerListe_01()`
WENN keine Manager in der DB existieren,
UND die Methode `getManagerListe` aufgerufen wird,
DANN sollte sie eine leere Liste zurückliefern.
- @Test: `insertManager_00()`
WENN die Methode `insertManager` mit einem TestManager aufgerufen wird,
UND die ID des TestManagers gleich `null` ist,
DANN sollte sie `TRUE` zurückliefern,
UND der TestManager sollte im Persistence Context existieren.
- @Test: `insertManager_01()`
WENN die Methode `insertManager` mit einem TestManager aufgerufen wird,
UND die ID des TestManagers `ungleich null` ist,
DANN sollte sie `FALSE` zurückliefern,
UND der Persistence Context wurde nicht verändert.
- @Test: `editManager_00()`
WENN ein TestManager in der DB existiert,
UND die Methode `editManager` mit einem veränderten TestManager (aber gleicher ID) aufgerufen wird,
DANN sollte sie `TRUE` zurückliefern,
UND der TestManager sollte im Persistence Context verändert sein.
- @Test: `editManager_01()`
WENN ein TestManager nicht in der DB existiert,
UND die Methode `editManager` mit dem TestManager aufgerufen wird,
DANN sollte sie `FALSE` zurückliefern,
UND der TestManager sollte nicht im Persistence Context existieren.
- @Test: `deleteManager_00()`
WENN ein TestManager in der DB existiert,
UND die Methode `deleteManager` mit der ID des TestManagers aufgerufen wird,
DANN sollte sie `TRUE` zurückliefern,
UND der TestManager sollte nicht mehr im Persistence Context existieren.
- @Test: `deleteManager_01()`
WENN ein TestManager nicht in der DB existiert,
UND die Methode `deleteManager` mit der ID des TestManagers aufgerufen wird,
DANN sollte sie `FALSE` zurückliefern.

2.4.2. IAntragManager

- @Before: angenommen()

Angenommen der EntityManager wird korrekt geholt,
UND die Implementierung der IAntragManager Schnittstelle wird als classUnderTest instanziert,
UND der EntityManager em wird per setEntityManager Methode der classUnderTest gesetzt,
UND die Transaktion von em wird gestartet,
UND die Daten der betreffenden Entitäten wurden im Persistence Context gelöscht.
- @After: amEnde()

Am Ende wird die Transaktion zurück gesetzt.
- @Test: getAntragListe_00()

WENN ein TestManager bereits in der Datenbank existiert,
UND x ($x > 0$) Anträge in der DB existieren, die an diesen TestManager gerichtet sind,
UND die Methode **getAntragListe** mit der ID des TestManagers aufgerufen wird,
DANN sollte sie die Liste mit diesen **x** Anträgen zurückliefern.
- @Test: getAntragListe_01()

WENN ein TestManager bereits in der Datenbank existiert,
UND keine Anträge in der DB existieren, die an diesen TestManager gerichtet sind,
UND die Methode **getAntragListe** mit der ID des TestManagers aufgerufen wird,
DANN sollte sie eine leere Liste zurückliefern.
- @Test: getAntragListe_02()

WENN ein TestManager nicht in der Datenbank existiert,
UND die Methode **getAntragListe** mit der ID des TestManager aufgerufen wird,
DANN sollte sie eine **leere Liste** zurückliefern.
- @Test: getListeEigenerAntraege_00()

WENN ein TestManager bereits in der Datenbank existiert,
UND x ($x > 0$) Anträge in der DB existieren, die dieser TestManager erstellt hat,
UND die Methode **getListeEigenerAntraege** mit der ID des TestManagers aufgerufen wird,
DANN sollte sie die Liste mit diesen **x** Anträgen zurückliefern.
- @Test: getListeEigenerAntraege_01()

WENN ein TestManager bereits in der Datenbank existiert,
UND keine Anträge in der DB existieren, die dieser TestManager erstellt hat,
UND die Methode **getListeEigenerAntraege** mit der ID des TestManagers aufgerufen wird,
DANN sollte sie eine leere Liste zurückliefern.
- @Test: getListeEigenerAntraege_02()

WENN ein TestManager nicht in der Datenbank existiert,
UND die Methode **getListeEigenerAntraege** mit der ID des TestManager aufgerufen wird,
DANN sollte sie eine **leere Liste** zurückliefern.

- @Test: getSachbearbeiterListe_00()

WENN $x (x>0)$ Sachbearbeiter in der DB existieren,
UND die Methode `getSachbearbeiterListe` aufgerufen wird,
DANN sollte sie die Liste mit diesen x Sachbearbeitern zurückliefern.
- @Test: getSachbearbeiterListe_01()

WENN keine Sachbearbeiter in der DB existieren,
UND die Methode `getSachbearbeiterListe` aufgerufen wird,
DANN sollte sie eine **leere Liste** zurückliefern.
- @Test: setAntragGenehmigt_00()

WENN ein TestAntrag bereits in der Datenbank existiert,
UND der Typ des TestAntrags "Kontoeröffnungs-Antrag" ist,
UND der Status des TestAntrags "neu" ist,
UND die Methode `setAntragGenehmigt` mit der Id des TestAntrags aufgerufen wird,
DANN sollte sie **TRUE** zurückliefern,
UND der TestAntrag sollte im Persistence Context den Status "genehmigt" besitzen.
- @Test: setAntragGenehmigt_01()

WENN ein TestAntrag nicht in der Datenbank existiert,
UND die Methode `setAntragGenehmigt` mit der Id des TestAntrags aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern,
UND der Persistence Context wurde nicht verändert.
- @Test: setAntragGenehmigt_02()

WENN ein TestAntrag bereits in der Datenbank existiert,
UND der Typ des TestAntrags "Kontoeröffnungs-Antrag" ist,
UND der Status des TestAntrags "genehmigt" ist,
UND die Methode `setAntragGenehmigt` mit der Id des TestAntrags aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern,
UND der Persistence Context wurde nicht verändert.
- @Test: setAntragGenehmigt_03()

WENN ein TestAntrag bereits in der Datenbank existiert,
UND der Typ des TestAntrags "Sachbearbeiter-Anlegen-Antrag" ist,
UND der Status des TestAntrags "neu" ist,
UND die Methode `setAntragGenehmigt` mit der Id des TestAntrags aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern,
UND der Persistence Context wurde nicht verändert.
- @Test: setAntragGenehmigt_04()

WENN ein TestAntrag bereits in der Datenbank existiert,
UND der Typ des TestAntrags "Kontolimit-Antrag" ist,
UND der Status des TestAntrags "neu" ist,
UND die Methode `setAntragGenehmigt` mit der Id des TestAntrags aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern,
UND der Persistence Context wurde nicht verändert.
- @Test: setAntragGenehmigt_05()

WENN ein TestAntrag bereits in der Datenbank existiert,
UND der Typ des TestAntrags "Kontoschließungs-Antrag" ist,
UND der Status des TestAntrags "neu" ist,
UND die Methode `setAntragGenehmigt` mit der Id des TestAntrags aufgerufen wird,
DANN sollte sie `FALSE` zurückliefern,
UND der Persistence Context wurde nicht verändert.

- @Test: setKSAntragGenehmigt_00()

WENN ein TestAntrag bereits in der Datenbank existiert,
UND der Typ des TestAntrags "Kontoschließungs-Antrag" ist,
UND der Status des TestAntrags "neu" ist,
UND ein Testadmin bereits in der Datenbank existiert,
UND die Methode `setKSAntragGenehmigt` mit der Id des TestAntrags und der Id des Testadmins aufgerufen wird,
DANN sollte sie `TRUE` zurückliefern,
UND der TestAntrag sollte im Persistennc Context den Status "genehmigt" besitzen,
UND der TestAntrag sollte im Persistennc Context mit dem Attribut `admin` auf den Testadmin verweisen.
- @Test: setKSAntragGenehmigt_01()

WENN ein TestAntrag bereits in der Datenbank existiert,
UND der Typ des TestAntrags "Sachbearbeiter-Anlegen-Antrag" ist,
UND der Status des TestAntrags "neu" ist,
UND ein Testadmin bereits in der Datenbank existiert,
UND die Methode `setKSAntragGenehmigt` mit der Id des TestAntrags und der Id des Testadmins aufgerufen wird,
DANN sollte sie `FALSE` zurückliefern,
UND der Persistence Context wurde nicht verändert.
- @Test: setKSAntragGenehmigt_02()

WENN ein TestAntrag bereits in der Datenbank existiert,
UND der Typ des TestAntrags "Kontoeröffnungs-Antrag" ist,
UND der Status des TestAntrags "neu" ist,
UND ein Testadmin bereits in der Datenbank existiert,
UND die Methode `setKSAntragGenehmigt` mit der Id des TestAntrags und der Id des Testadmins aufgerufen wird,
DANN sollte sie `FALSE` zurückliefern,
UND der Persistence Context wurde nicht verändert.
- @Test: setKSAntragGenehmigt_03()

WENN ein TestAntrag bereits in der Datenbank existiert,
UND der Typ des TestAntrags "Kontolimit-Antrag" ist,
UND der Status des TestAntrags "neu" ist,
UND ein Testadmin bereits in der Datenbank existiert,
UND die Methode `setKSAntragGenehmigt` mit der Id des TestAntrags und der Id des Testadmins aufgerufen wird,
DANN sollte sie `FALSE` zurückliefern,

UND der Persistence Context wurde nicht verändert.

- @Test: setKSAntragGenehmigt_04()

WENN ein TestAntrag nicht in der Datenbank existiert,
UND ein Testadmin bereits in der Datenbank existiert,
UND die Methode **setKSAntragGenehmigt** mit der Id des TestAntrags und der Id des Testadmins aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern,
UND der Persistence Context wurde nicht verändert.
- @Test: setKSAntragGenehmigt_05()

WENN ein TestAntrag bereits in der Datenbank existiert,
UND der Typ des TestAntrags "Kontoschließungs-Antrag" ist,
UND der Status des TestAntrags "genehmigt" ist,
UND ein Testadmin bereits in der Datenbank existiert,
UND die Methode **setKSAntragGenehmigt** mit der Id des TestAntrags und der Id des Testadmins aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern,
UND der Persistence Context wurde nicht verändert.
- @Test: setAntragBearbeitet_00()

WENN ein Testantrag bereits in der DB existiert,
UND der Testantrag den Typ "Kontolimit-Antrag" besitzt,
UND die Methode **setAntragBearbeitet** mit der Id des Testantrags aufgerufen wird,
DANN sollte sie den Status des Testantrags auf "bearbeitet" setzen,
UND **TRUE** zurückliefern
- @Test: setAntragBearbeitet_01()

WENN ein Testantrag bereits in der DB existiert,
UND der Testantrag den Typ "Sachbearbeiter-Löschen-Antrag" besitzt,
UND der Testantrag den Status "genehmigt" besitzt,
UND die Methode **setAntragBearbeitet** mit der Id des Testantrags aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern,
UND der Testantrag wurde nicht verändert.
- @Test: setAntragBearbeitet_02()

WENN ein Testantrag bereits in der DB existiert,
UND der Testantrag den Typ "Sachbearbeiter-Bearbeiten-Antrag" besitzt,
UND der Testantrag den Status "genehmigt" besitzt,
UND die Methode **setAntragBearbeitet** mit der Id des Testantrags aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern,
UND der Testantrag wurde nicht verändert.
- @Test: setAntragBearbeitet_03()

WENN ein Testantrag bereits in der DB existiert,
UND der Testantrag den Typ "Kontoschließungs-Antrag" besitzt,
UND der Testantrag den Status "genehmigt" besitzt,
UND die Methode **setAntragBearbeitet** mit der Id des Testantrags aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern,

UND der Testantrag wurde nicht verändert.

- @Test: setAntragBearbeitet_04()

WENN ein Testantrag bereits in der DB existiert,
UND der Testantrag den Typ "Kontoeröffnungs-Antrag" besitzt,
UND die Methode **setAntragBearbeitet** mit der Id des Testantrags aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern,
UND der Testantrag wurde nicht verändert.
- @Test: setAntragBearbeitet_05()

WENN ein Testantrag bereits in der DB existiert,
UND der Testantrag den Typ "Sachbearbeiter-Anlegen-Antrag" besitzt,
UND die Methode **setAntragBearbeitet** mit der Id des Testantrags aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern,
UND der Testantrag wurde nicht verändert.
- @Test: setAntragBearbeitet_06()

WENN ein Testantrag bereits in der DB existiert,
UND der Testantrag den Typ "Kontolimit-Antrag" besitzt,
UND der Testantrag den Status "bearbeitet" besitzt,
UND die Methode **setAntragBearbeitet** mit der Id des Testantrags aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern,
UND der Testantrag wurde nicht verändert.
- @Test: setAntragBearbeitet_07()

WENN ein Testantrag nicht in der DB existiert,
UND die Methode **setAntragBearbeitet** mit der Id des Testantrags aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern.
- @Test: setAntragAbgelehnt_00()

WENN ein TestAntrag bereits in der Datenbank existiert,
UND der Typ des TestAntrags "Kontoeröffnungs-Antrag" ist,
UND der Status des TestAntrags "neu" ist,
UND die Methode **setAntragAbgelehnt** mit der Id des TestAntrags aufgerufen wird,
DANN sollte sie **TRUE** zurückliefern,
UND der TestAntrag sollte im Persistenncce Context den Status "abgelehnt" besitzen.
- @Test: setAntragAbgelehnt_01()

WENN ein TestAntrag bereits in der Datenbank existiert,
UND der Typ des TestAntrags "Kontoschließungs-Antrag" ist,
UND der Status des TestAntrags "neu" ist,
UND die Methode **setAntragAbgelehnt** mit der Id des TestAntrags aufgerufen wird,
DANN sollte sie **TRUE** zurückliefern,
UND der TestAntrag sollte im Persistenncce Context den Status "abgelehnt" besitzen.
- @Test: setAntragAbgelehnt_02()

WENN ein TestAntrag bereits in der Datenbank existiert,
UND der Typ des TestAntrags "Kontolimit-Antrag" ist,
UND der Status des TestAntrags "neu" ist,

UND die Methode **setAntragAbgelehnt** mit der Id des TestAntrags aufgerufen wird,
DANN sollte sie **TRUE** zurückliefern,
UND der TestAntrag sollte im Persistennc Context den Status "abgelehnt" besitzen.

- @Test: setAntragAbgelehnt_03()

WENN ein TestAntrag nicht in der Datenbank existiert,
UND die Methode **setAntragAbgelehnt** mit der Id des TestAntrags aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern,
UND der Persistence Context wurde nicht verändert.
- @Test: setAntragAbgelehnt_04()

WENN ein TestAntrag bereits in der Datenbank existiert,
UND der Typ des TestAntrags "Kontoeröffnungs-Antrag" ist,
UND der Status des TestAntrags "abgelehnt" ist,
UND die Methode **setAntragAbgelehnt** mit der Id des TestAntrags aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern,
UND der Persistence Context wurde nicht verändert.
- @Test: setAntragAbgelehnt_05()

WENN ein TestAntrag bereits in der Datenbank existiert,
UND der Typ des TestAntrags "Sachbearbeiter-Löschen-Antrag" ist,
UND die Methode **setAntragAblehnt** mit der Id des TestAntrags aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern,
UND der Persistence Context wurde nicht verändert.
- @Test: setAntragStorniert_00()

WENN ein TestAntrag bereits in der Datenbank existiert,
UND ein TestManager bereits in der DB existiert,
UND der Status des TestAntrags "neu" ist,
UND der Typ des TestAntrags "'Sachbearbeiter-Anlegen-Antrag" ist,
UND der Testantrag vom TestManager erstellt wurde,
UND die Methode **setAntragStorniert** mit der Id des TestAntrags und der ID des TestManagers aufgerufen wird,
DANN sollte sie **TRUE** zurückliefern,
UND der TestAntrag sollte im Persistennc Context den Status "storniert" besitzen.
- @Test: setAntragStorniert_01()

WENN ein TestAntrag bereits in der Datenbank existiert,
UND ein TestManager bereits in der DB existiert, **UND** der Status des TestAntrags "neu" ist,
UND der Typ des TestAntrags "'Sachbearbeiter-Löschen-Antrag" ist,
UND der Testantrag vom TestManager erstellt wurde,
UND die Methode **setAntragStorniert** mit der Id des TestAntrags und der ID des TestManagers aufgerufen wird,
DANN sollte sie **TRUE** zurückliefern,
UND der TestAntrag sollte im Persistennc Context den Status "storniert" besitzen.
- @Test: setAntragStorniert_02()

WENN ein TestAntrag bereits in der Datenbank existiert,
UND ein TestManager bereits in der DB existiert,

UND der Status des TestAntrags "neu" ist,
UND der Typ des TestAntrags "'Sachbearbeiter-Bearbeiten-Antrag" ist,
UND der Testantrag vom TestManager erstellt wurde,
UND die Methode `setAntragStorniert` mit der Id des TestAntrags und der ID des TestManagers aufgerufen wird,
DANN sollte sie `TRUE` zurückliefern,
UND der TestAntrag sollte im Persistence Context den Status "storniert" besitzen.

- @Test: `setAntragStorniert_03()`
WENN ein TestAntrag nicht in der Datenbank existiert,
UND die Methode `setAntragStorniert` mit der Id des TestAntrags aufgerufen wird,
DANN sollte sie `FALSE` zurückliefern,
UND der Persistence Context wurde nicht verändert.
- @Test: `setAntragStorniert_04()`
WENN ein TestAntrag bereits in der Datenbank existiert,
UND ein TestManager bereits in der DB existiert,
UND der Typ des TestAntrags "'Sachbearbeiter-Bearbeiten-Antrag" ist,
UND der Status des TestAntrags "storniert" ist,
UND der Testantrag vom TestManager erstellt wurde,
UND die Methode `setAntragStorniert` mit der Id des TestAntrags und der ID des TestManagers aufgerufen wird,
DANN sollte sie `FALSE` zurückliefern,
UND der Persistence Context wurde nicht verändert.
- @Test: `setAntragStorniert_05()`
WENN ein TestAntrag bereits in der Datenbank existiert,
UND ein TestManager bereits in der DB existiert,
UND der Typ des TestAntrags "Kontolimit-Antrag" ist,
UND der Status des TestAntrags "neu" ist,
UND die Methode `setAntragStorniert` mit der Id des TestAntrags und der ID des TestManagers aufgerufen wird,
DANN sollte sie `FALSE` zurückliefern,
UND der Persistence Context wurde nicht verändert.

2.4.3. IAntragStellenManager

- @Before: `angenommen()`
Angenommen der EntityManager wird korrekt geholt,
UND die Implementierung der IAntragStellenManager Schnittstelle wird als classUnderTest instanziert,
UND der EntityManager em wird per `setEntityManager` Methode der classUnderTest gesetzt,
UND die Transaktion von em wird gestartet,
UND die Daten der betreffenden Entitäten wurden im Persistence Context gelöscht.
- @After: `amEnde()`

Am Ende wird die Transaktion zurück gesetzt.

- @Test: antragStellen_000
WENN die Methode **antragStellen** mit einem TestAntrag aufgerufen wird,
UND die ID des TestAntrags gleich **null** ist,
UND der Typ des Antrags "ks" ist,
DANN sollte sie **TRUE** zurückliefern,
UND der TestAntrag sollte im Persistennc Context existieren,
UND der Status des Testantrags sollte "neu" sein.
- @Test: antragStellen_010
WENN die Methode **antragStellen** mit einem TestAntrag aufgerufen wird,
UND die ID des TestAntrags gleich **null** ist,
UND der Typ des Antrags "kl" ist,
DANN sollte sie **TRUE** zurückliefern,
UND der TestAntrag sollte im Persistennc Context existieren,
UND der Status des Testantrags sollte "neu" sein.
- @Test: antragStellen_020
WENN die Methode **antragStellen** mit einem TestAntrag aufgerufen wird,
UND die ID des TestAntrags gleich **null** ist,
UND der Typ des Antrags "ko" ist,
DANN sollte sie **TRUE** zurückliefern,
UND der TestAntrag sollte im Persistennc Context existieren,
UND der Status des Testantrags sollte "neu" sein.
- @Test: antragStellen_030
WENN die Methode **antragStellen** mit einem TestAntrag aufgerufen wird,
UND die ID des TestAntrags gleich **null** ist,
UND der Typ des Antrags "sd" ist,
DANN sollte sie **FALSE** zurückliefern,
UND der Persistence Context wurde nicht verändert.
- @Test: antragStellen_040
WENN die Methode **antragStellen** mit einem TestAntrag aufgerufen wird,
UND die ID des TestAntrags **ungleich null** ist,
DANN sollte sie **FALSE** zurückliefern,
UND der Persistence Context wurde nicht verändert.
- @Test: getManagerListe_000
WENN x ($x > 0$) Manager in der DB existieren,
UND die Methode **getManagerListe** aufgerufen wird,
DANN sollte sie die Liste mit diesen x Manager zurückliefern.
- @Test: getManagerListe_010
WENN keine Manager in der DB existieren,
UND die Methode **getManagerListe** aufgerufen wird,
DANN sollte sie eine **leere Liste** zurückliefern.

2.5. Testspezifikation zur Komponente SachbearbeiterDaten

2.5.1. ICRUDSach

- @Before: angenommen()

Angenommen der EntityManager wird korrekt geholt,
UND die Implementierung der ICRUDSach Schnittstelle wird als classUnderTest instanziert,
UND der EntityManager em wird per setEntityManager Methode der classUnderTest gesetzt,
UND die Transaktion von em wird gestartet,
UND die Daten der betreffenden Entitäten wurden im Persistence Context gelöscht.
- @After: amEnde()

Am Ende wird die Transaktion zurück gesetzt.
- @Test: getSachByID_00()

WENN ein Testsachbearbeiter bereits in der DB existiert,
UND die Methode **getSachByID** mit der Id des Testsachbearbeiters aufgerufen wird,
DANN sollte sie den **Testsachbearbeiter** zurückliefern.
- @Test: getSachByID_01()

WENN ein Testsachbearbeiter nicht in der DB existiert,
UND die Methode **getSachByID** mit der Id des Testsachbearbeiters aufgerufen wird,
DANN sollte sie **NULL** zurückliefern.
- @Test: getSachListe_00()

WENN x ($x > 0$) Sachbearbeiter in der DB existieren,
UND die Methode **getSachListe** aufgerufen wird,
DANN sollte sie eine Liste mit x Sachbearbeitern zurückliefern.
- @Test: getSachListe_01()

WENN keine Sachbearbeiter in der DB existieren,
UND die Methode **getSachListe** aufgerufen wird,
DANN sollte sie eine **leere Liste** zurückliefern.
- @Test: insertSach_00()

WENN die Methode **insertSach** mit einem Testsachbearbeiter aufgerufen wird,
UND die ID des Testsachbearbeiters gleich **null** ist,
DANN sollte sie **TRUE** zurückliefern,
UND der Testsachbearbeiter sollte im Persistence Context existieren.
- @Test: insertSach_01()

WENN die Methode **insertSach** mit einem Testsachbearbeiter aufgerufen wird,
UND die ID des Testsachbearbeiter **ungleich null** ist,
DANN sollte sie **FALSE** zurückliefern,
UND der Persistence Context wurde nicht verändert.

- @Test: editSach_00()

WENN ein Testsachbearbeiter in der DB existiert,
UND die Methode **editSach** mit einem veränderten Testsachbearbeiter (aber gleicher ID) aufgerufen wird,
DANN sollte sie **TRUE** zurückliefern,
UND der Testsachbearbeiter sollte im Persistence Context verändert sein.
- @Test: editSach_01()

WENN ein Testsachbearbeiter nicht in der DB existiert,
UND die Methode **editSach** mit dem Testsachbearbeiter aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern,
UND der Testsachbearbeiter sollte nicht im Persistence Context existieren.
- @Test: deleteSach_00()

WENN ein Testsachbearbeiter in der DB existiert,
UND die Methode **deleteSach** mit der ID des Testsachbearbeiters aufgerufen wird,
DANN sollte sie **TRUE** zurückliefern,
UND der Testsachbearbeiter sollte nicht mehr im Persistence Context existieren.
- @Test: deleteSach_01()

WENN ein Testsachbearbeiter nicht in der DB existiert,
UND die Methode **deleteSach** mit der ID des Testsachbearbeiters aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern.
- @Test: getSachA1_00()

WENN x ($x > 0$) Sachbearbeiter in der DB existieren, die zur Abteilung **A1** gehören,
UND die Methode **getSachA1** aufgerufen wird,
DANN sollte sie eine Liste mit x Sachbearbeitern zurückliefern.
- @Test: getSachA1_01()

WENN keine Sachbearbeiter in der DB existieren, die zur Abteilung **A1** gehören,
UND die Methode **getSachA1** aufgerufen wird,
DANN sollte sie eine **leere Liste** zurückliefern.
- @Test: getSachA2_00()

WENN x ($x > 0$) Sachbearbeiter in der DB existieren, die zur Abteilung **A2** gehören,
UND die Methode **getSachA2** aufgerufen wird,
DANN sollte sie eine Liste mit x Sachbearbeitern zurückliefern.
- @Test: getSachA2_01()

WENN keine Sachbearbeiter in der DB existieren, die zur Abteilung **A2** gehören,
UND die Methode **getSachA2** aufgerufen wird,
DANN sollte sie eine **leere Liste** zurückliefern.

2.5.2. IUeberweisungsVorlage

- @Before: angenommen()

Angenommen der EntityManager wird korrekt geholt,
UND die Implementierung der IUeberweisungsVorlage Schnittstelle wird als

classUnderTest instanziert,
UND der EntityManager em wird per setEntityManager Methode der classUnderTest gesetzt,
UND die Transaktion von em wird gestartet,
UND die Daten der betreffenden Entitäten wurden im Persistence Context gelöscht.

- @After: amEnde()
Am Ende wird die Transaktion zurück gesetzt.
- @Test: getUeberweisungsVorlageByID_000
WENN eine TestUeberweisungsVorlage bereits in der DB existiert,
UND die Methode `getUeberweisungsVorlageByID` mit der Id der TestUeberweisungsVorlage aufgerufen wird,
DANN sollte sie die `TestUeberweisungsVorlage` zurückliefern.
- @Test: getUeberweisungsVorlageByID_010
WENN eine TestUeberweisungsVorlage nicht in der DB existiert,
UND die Methode `getUeberweisungsVorlageByID` mit der Id der TestUeberweisungsVorlage aufgerufen wird,
DANN sollte sie `NULL` zurückliefern.
- @Test: getUeberweisungsVorlagenDesKunden_000
WENN ein TestKunde bereits in der DB existiert,
UND x ($x > 0$) ÜberweisungsVorlagen in der DB existieren, deren Attribut `kunde` auf den TestKunden verweisen,
UND die Methode `getUeberweisungsVorlagenDesKunden` mit der ID des TestKunden aufgerufen wird,
DANN sollte sie die Liste mit den `x` ÜberweisungsVorlagen zurückliefern.
- @Test: getUeberweisungsVorlagenDesKunden_010
WENN ein TestKunde bereits in der DB existiert,
UND keine ÜberweisungsVorlagen in der DB existieren, deren Attribut `kunde` auf den TestKunden verweisen,
UND die Methode `getUeberweisungsVorlagenDesKunden` mit der ID des TestKunden aufgerufen wird,
DANN sollte sie eine `leere Liste` zurückliefern.
- @Test: getUeberweisungsVorlagenDesKunden_020
WENN ein TestKunde nicht in der DB existiert,
UND die Methode `getUeberweisungsVorlagenDesKunden` mit der ID des TestKunden aufgerufen wird,
DANN sollte sie eine `leere Liste` zurückliefern.
- @Test: getUeberweisungsVorlagenDesKontos_00()
WENN ein TestKonto bereits in der DB existiert,
UND x ($x > 0$) ÜberweisungsVorlagen in der DB existieren, deren Attribut `vonkontos` auf das TestKonto verweisen,
UND die Methode `getUeberweisungsVorlagenDesKontos` mit der ID des TestKontos aufgerufen wird,

DANN sollte sie die Liste mit den **x** ÜberweisungsVorlagen zurückliefern.

- @Test: getUeberweisungsVorlagenDesKontos_01()

WENN ein TestKonto bereits in der DB existiert,
UND keine ÜberweisungsVorlagen in der DB existieren, deren Attribut **vonkont** auf das TestKonto verweisen,
UND die Methode `getUeberweisungsVorlagenDesKontos` mit der ID des TestKontos aufgerufen wird,
DANN sollte sie eine **leere Liste** zurückliefern.
- @Test: getUeberweisungsVorlagenDesKontos_02()

WENN ein TestKonto nicht in der DB existiert,
UND die Methode `getUeberweisungsVorlagenDesKontos` mit der ID des TestKontos aufgerufen wird,
DANN sollte sie eine **leere Liste** zurückliefern.
- @Test: insertUeberweisungsVorlage_00()

WENN die Methode `insertUeberweisungsVorlage` mit einer TestUeberweisungsVorlage aufgerufen wird,
UND die ID der TestUeberweisungsVorlage gleich **null** ist,
DANN sollte sie **TRUE** zurückliefern,
UND die TestUeberweisungsVorlage sollte im Persistence Context existieren.
- @Test: insertUeberweisungsVorlage_01()

WENN die Methode `insertUeberweisungsVorlage` mit einer TestUeberweisungsVorlage aufgerufen wird,
UND die ID der TestUeberweisungsVorlage **ungleich null** ist,
DANN sollte sie **FALSE** zurückliefern,
UND der Persistence Context wurde nicht verändert.
- @Test: deleteUeberweisungsVorlage_00()

WENN eine TestUeberweisungsVorlage in der DB existiert,
UND die Methode `deleteUeberweisungsVorlage` mit der ID der TestUeberweisungsVorlage aufgerufen wird,
DANN sollte sie **TRUE** zurückliefern,
UND die TestUeberweisungsVorlage sollte nicht mehr im Persistence Context existieren.
- @Test: deleteUeberweisungsVorlage_01()

WENN eine TestUeberweisungsVorlage nicht in der DB existiert,
UND die Methode `deleteUeberweisungsVorlage` mit der ID der TestUeberweisungsVorlage aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern.
- @Test: editUeberweisungsVorlage_00()

WENN eine TestUeberweisungsVorlage in der DB existiert,
UND die Methode `editUeberweisungsVorlage` mit einer veränderten TestUeberweisungsVorlage (aber gleicher ID) aufgerufen wird,
DANN sollte sie **TRUE** zurückliefern,
UND die TestUeberweisungsVorlage sollte im Persistence Context verändert sein.

- @Test: editUeberweisungsVorlage_01()

WENN eine TestUeberweisungsVorlage nicht in der DB existiert,
UND die Methode `editUeberweisungsVorlage` mit der TestUeberweisungsVorlage aufgerufen wird,
DANN sollte sie `FALSE` zurückliefern,
UND die TestUeberweisungsVorlage sollte nicht im Persistence Context existieren.
- @Test: getUeberweisungenZuKonto_00()

WENN ein Testkonto bereits in der Datenbank existiert,
UND x ($x > 0$) Überweisungen in der DB existieren, die über das Attribut `zukonto` auf das Testkonto verweisen,
UND die Methode `getUeberweisungenZuKonto` mit der ID des TestKontos aufgerufen wird,
DANN sollte sie die Liste mit den x Überweisungen zurückliefern.
- @Test: getUeberweisungenZuKonto_01()

WENN ein Testkonto bereits in der Datenbank existiert,
UND keine Überweisungen in der DB existieren, die über das Attribut `zukonto` auf das Testkonto verweisen,
UND die Methode `getUeberweisungenZuKonto` mit der ID des TestKontos aufgerufen wird,
DANN sollte sie eine **leere Liste** zurückliefern.
- @Test: getUeberweisungenZuKonto_02()

WENN ein Testkonto nicht in der Datenbank existiert,
UND die Methode `getUeberweisungenZuKonto` mit der ID des TestKontos aufgerufen wird,
DANN sollte sie eine **leere Liste** zurückliefern.
- @Test: getUeberweisungenVonKonto_00()

WENN ein Testkonto bereits in der Datenbank existiert,
UND x ($x > 0$) Überweisungen in der DB existieren, die über das Attribut `vonkonto` auf das Testkonto verweisen,
UND die Methode `getUeberweisungenVonKonto` mit der ID des TestKontos aufgerufen wird,
DANN sollte sie die Liste mit den x Überweisungen zurückliefern.
- @Test: getUeberweisungenVonKonto_01()

WENN ein Testkonto bereits in der Datenbank existiert,
UND keine Überweisungen in der DB existieren, die über das Attribut `vonkonto` auf das Testkonto verweisen,
UND die Methode `getUeberweisungenVonKonto` mit der ID des TestKontos aufgerufen wird,
DANN sollte sie eine **leere Liste** zurückliefern.
- @Test: getUeberweisungenVonKonto_02()

WENN ein Testkonto nicht in der Datenbank existiert,
UND die Methode `getUeberweisungenVonKonto` mit der ID des TestKontos aufgerufen wird,
DANN sollte sie eine **leere Liste** zurückliefern.
- @Test: getLoeschbareUeberweisungen_00()

WENN x ($x > 0$) Überweisungen mit Status "ueberwiesen" in der DB existieren,
UND y ($y > 0$) Überweisungen mit Status "storniert" in der DB existieren,
UND z ($z > 0$) Überweisungen mit Status "nicht ueberweisbar" in der DB existieren,

UND die Methode `getLoeschbareUeberweisungen` aufgerufen wird,
DANN sollte sie die Liste mit den `x+y+z` Überweisungen zurückliefern.

- @Test: `getLoeschbareUeberweisungen_01()`

WENN keine Überweisungen mit Status "ueberwiesen" in der DB existieren,
UND keine Überweisungen mit Status "storniert" in der DB existieren,
UND keine Überweisungen mit Status "nicht ueberweisbar" in der DB existieren,
UND die Methode `getLoeschbareUeberweisungen` aufgerufen wird,
DANN sollte sie eine **leere Liste** zurückliefern.
- @Test: `loescheUeberweisung_00()`

WENN eine TestUeberweisung bereits in der DB existiert,
UND diese TestUeberweisung den Status "ueberwiesen" besitzt
UND die Methode `loescheUeberweisung` mit der Id der TestUeberweisung aufgerufen wird,
DANN sollte sie **TRUE** zurückliefern,
UND die TestUeberweisung sollte nicht mehr im Persistence Context existieren.
- @Test: `loescheUeberweisung_01()`

WENN eine TestUeberweisung bereits in der DB existiert,
UND diese TestUeberweisung den Status "storniert" besitzt
UND die Methode `loescheUeberweisung` mit der Id der TestUeberweisung aufgerufen wird,
DANN sollte sie **TRUE** zurückliefern,
UND die TestUeberweisung sollte nicht mehr im Persistence Context existieren.
- @Test: `loescheUeberweisung_02()`

WENN eine TestUeberweisung bereits in der DB existiert,
UND diese TestUeberweisung den Status "nicht ueberweisbar" besitzt
UND die Methode `loescheUeberweisung` mit der Id der TestUeberweisung aufgerufen wird,
DANN sollte sie **TRUE** zurückliefern,
UND die TestUeberweisung sollte nicht mehr im Persistence Context existieren.
- @Test: `loescheUeberweisung_03()`

WENN eine TestUeberweisung bereits in der DB existiert,
UND diese TestUeberweisung den Status "wartet" besitzt
UND die Methode `loescheUeberweisung` mit der Id der TestUeberweisung aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern,
UND die TestUeberweisung sollte noch in der DB existieren.
- @Test: `loescheUeberweisung_04()`

WENN eine TestUeberweisung nicht in der DB existiert,
UND die Methode `loescheUeberweisung` mit der Id der TestUeberweisung aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern.
- @Test: `getWartendeUeberweisungen_00()`

WENN $x (x>0)$ Überweisungen mit Status "wartet" in der DB existieren,
UND die Methode `getWartendeUeberweisungen` aufgerufen wird,
DANN sollte sie die Liste mit den x Überweisungen zurückliefern.

- @Test: `getWartendeUeberweisungen_01()`
WENN keine Überweisungen mit Status "wartet" in der DB existieren,
UND die Methode `getWartendeUeberweisungen` aufgerufen wird,
DANN sollte sie eine `leere Liste` zurückliefern.

2.5.3. IAntragSach

- @Before: `angenommen()`
Angenommen der EntityManager wird korrekt geholt,
UND die Implementierung der IAntragSach Schnittstelle wird als classUnderTest instanziert,
UND der EntityManager em wird per `setEntityManager` Methode der classUnderTest gesetzt,
UND die Transaktion von em wird gestartet,
UND die Daten der betreffenden Entitäten wurden im Persistence Context gelöscht.
- @After: `amEnde()`
Am Ende wird die Transaktion zurück gesetzt.
- @Test: `getAntragListe_00()`
WENN ein TestSachbearbeiter bereits in der Datenbank existiert,
UND $x (x>0)$ Anträge in der DB existieren, die dieser TestSachbearbeiter erstellt hat,
UND die Methode `getAntragListe` mit der ID des TestSachbearbeiters aufgerufen wird,
DANN sollte sie die Liste mit diesen x Anträgen zurückliefern.
- @Test: `getAntragListe_01()`
WENN ein TestSachbearbeiter bereits in der Datenbank existiert,
UND keine Anträge in der DB existieren, die dieser TestSachbearbeiter erstellt hat,
UND die Methode `getAntragListe` mit der ID des TestSachbearbeiters aufgerufen wird,
DANN sollte sie eine leere Liste zurückliefern.
- @Test: `getAntragListe_02()`
WENN ein TestSachbearbeiter nicht in der Datenbank existiert,
UND die Methode `getAntragListe` mit der ID des TestSachbearbeiters aufgerufen wird,
DANN sollte sie eine `leere Liste` zurückliefern.
- @Test: `setAntragBearbeitet_00()`
WENN ein Testantrag bereits in der DB existiert,
UND ein TestSachbearbeiter bereits in der DB existiert,
UND der Testantrag den Typ "Kontoeröffnungs-Antrag" besitzt,
UND der Testantrag den Status "genehmigt" besitzt,
UND der Testantrag vom TestSachbearbeiter erstellt wurde,
UND die Methode `setAntragBearbeitet` mit der ID des Testantrags und der ID des TestSachbearbeiters aufgerufen wird,
DANN sollte der Status des `Testantrags` auf "bearbeitet" gesetzt sein,

UND TRUE zurückliefern

- @Test: setAntragBearbeitet_01()

WENN ein Testantrag bereits in der DB existiert,
UND ein TestSachbearbeiter bereits in der DB existiert,
UND der Testantrag den Typ "Kontolimit-Antrag" besitzt,
UND der Testantrag den Status "genehmigt" besitzt,
UND der Testantrag vom TestSachbearbeiter erstellt wurde,
UND die Methode **setAntragBearbeitet** mit der Id des Testantrags und der ID des TestSachbearbeiters aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern,
UND der Testantrag wurde nicht verändert.
- @Test: setAntragBearbeitet_02()

WENN ein Testantrag bereits in der DB existiert,
UND ein TestSachbearbeiter bereits in der DB existiert,
UND der Testantrag den Typ "Sachbearbeiter-Bearbeiten-Antrag" besitzt,
UND der Testantrag den Status "genehmigt" besitzt,
UND die Methode **setAntragBearbeitet** mit der Id des Testantrags und der ID des TestSachbearbeiters aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern,
UND der Testantrag wurde nicht verändert.
- @Test: setAntragBearbeitet_03()

WENN ein Testantrag bereits in der DB existiert,
UND ein TestSachbearbeiter bereits in der DB existiert,
UND der Testantrag den Typ "Kontoschließungs-Antrag" besitzt,
UND der Testantrag den Status "genehmigt" besitzt,
UND der Testantrag vom TestSachbearbeiter erstellt wurde,
UND die Methode **setAntragBearbeitet** mit der Id des Testantrags und der ID des TestSachbearbeiters aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern,
UND der Testantrag wurde nicht verändert.
- @Test: setAntragBearbeitet_04()

WENN ein Testantrag bereits in der DB existiert,
UND ein TestSachbearbeiter bereits in der DB existiert,
UND der Testantrag den Typ "Sachbearbeiter-Löschen-Antrag" besitzt,
UND die Methode **setAntragBearbeitet** mit der Id des Testantrags und der ID des TestSachbearbeiters aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern,
UND der Testantrag wurde nicht verändert.
- @Test: setAntragBearbeitet_05()

WENN ein Testantrag bereits in der DB existiert,
UND ein TestSachbearbeiter bereits in der DB existiert,
UND der Testantrag den Typ "Sachbearbeiter-Anlegen-Antrag" besitzt,
UND die Methode **setAntragBearbeitet** mit der Id des Testantrags und der ID des

TestSachbearbeiters aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern,
UND der Testantrag wurde nicht verändert.

- @Test: setAntragBearbeitet_06()

WENN ein Testantrag bereits in der DB existiert,
UND ein TestSachbearbeiter bereits in der DB existiert,
UND der Testantrag den Typ "Kontoeröffnungs-Antrag" besitzt,
UND der Testantrag den Status "neu" besitzt,
UND der Testantrag vom TestSachbearbeiter erstellt wurde,
UND die Methode **setAntragBearbeitet** mit der Id des Testantrags und der ID des TestSachbearbeiters aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern,
UND der Testantrag wurde nicht verändert.
- @Test: setAntragBearbeitet_07()

WENN ein Testantrag nicht in der DB existiert,
UND ein TestSachbearbeiter bereits in der DB existiert,
UND die Methode **setAntragBearbeitet** mit der Id des Testantrags und der ID des TestSachbearbeiters aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern.
- @Test: antragStornieren_00()

WENN ein TestAntrag bereits in der Datenbank existiert,
UND ein TestSachbearbeiter bereits in der DB existiert,
UND der Status des TestAntrags "neu" ist,
UND der Typ des TestAntrags "Kontoeröffnungs-Antrag" ist,
UND der Testantrag vom TestSachbearbeiter erstellt wurde,
UND die Methode **antragStornieren** mit der Id des TestAntrags und der ID des TestSachbearbeiters aufgerufen wird,
DANN sollte sie **TRUE** zurückliefern,
UND der TestAntrag sollte im Persistenncce Context den Status "storniert" besitzen.
- @Test: antragStornieren_01()

WENN ein TestAntrag bereits in der Datenbank existiert,
UND ein TestSachbearbeiter bereits in der DB existiert,
UND der Status des TestAntrags "genehmigt" ist,
UND der Typ des TestAntrags "Kontoeröffnungs-Antrag" ist,
UND der Testantrag vom TestSachbearbeiter erstellt wurde,
UND die Methode **antragStornieren** mit der Id des TestAntrags und der ID des TestSachbearbeiters aufgerufen wird,
DANN sollte sie **TRUE** zurückliefern,
UND der TestAntrag sollte im Persistenncce Context den Status "storniert" besitzen.
- @Test: antragStornieren_02()

WENN ein TestAntrag bereits in der Datenbank existiert,
UND ein TestSachbearbeiter bereits in der DB existiert,
UND der Status des TestAntrags "neu" ist,

UND der Typ des TestAntrags "Kontoschließungs-Antrag" ist,
UND der Testantrag vom TestSachbearbeiter erstellt wurde,
UND die Methode **antragStornieren** mit der Id des TestAntrags und der ID des TestSachbearbeiters aufgerufen wird,
DANN sollte sie **TRUE** zurückliefern,
UND der TestAntrag sollte im Persistennc Context den Status "storniert" besitzen.

- @Test: antragStornieren_03()

WENN ein TestAntrag bereits in der Datenbank existiert,
UND ein TestSachbearbeiter bereits in der DB existiert,
UND der Status des TestAntrags "genehmigt" ist,
UND der Typ des TestAntrags "Kontoschließungs-Antrag" ist,
UND der Testantrag vom TestSachbearbeiter erstellt wurde,
UND die Methode **antragStornieren** mit der Id des TestAntrags und der ID des TestSachbearbeiters aufgerufen wird,
DANN sollte sie **TRUE** zurückliefern,
UND der TestAntrag sollte im Persistennc Context den Status "storniert" besitzen.
- @Test: antragStornieren_04()

WENN ein TestAntrag bereits in der Datenbank existiert,
UND ein TestSachbearbeiter bereits in der DB existiert,
UND der Status des TestAntrags "neu" ist,
UND der Typ des TestAntrags "Kontolimit-Antrag" ist,
UND der Testantrag vom TestSachbearbeiter erstellt wurde,
UND die Methode **antragStornieren** mit der Id des TestAntrags und der ID des TestSachbearbeiters aufgerufen wird,
DANN sollte sie **TRUE** zurückliefern,
UND der TestAntrag sollte im Persistennc Context den Status "storniert" besitzen.
- @Test: antragStornieren_05()

WENN ein TestAntrag nicht in der Datenbank existiert,
UND ein TestSachbearbeiter bereits in der DB existiert,
UND die Methode **antragStornieren** mit der Id des TestAntrags und der ID des TestSachbearbeiters aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern,
UND der Persistence Context wurde nicht verändert.
- @Test: antragStornieren_06()

WENN ein TestAntrag bereits in der Datenbank existiert,
UND ein TestSachbearbeiter bereits in der DB existiert,
UND der Typ des TestAntrags "Kontoeröffnungs-Antrag" ist,
UND der Status des TestAntrags "abgelehnt" ist,
UND der Testantrag vom TestSachbearbeiter erstellt wurde,
UND die Methode **antragStornieren** mit der Id des TestAntrags und der ID des TestSachbearbeiters aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern,
UND der Persistence Context wurde nicht verändert.

- @Test: antragStornieren_07()

WENN ein TestAntrag bereits in der Datenbank existiert,
UND ein TestSachbearbeiter bereits in der DB existiert,
UND der Typ des TestAntrags "Sachbearbeiter-Anlegen-Antrag" ist,
UND die Methode **antragStornieren** mit der Id des TestAntrags und der ID des TestSachbearbeiters aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern,
UND der Persistence Context wurde nicht verändert.
- @Test: getNeueAntraege_00()

WENN ein TestSachbearbeiter bereits in der Datenbank existiert,
UND x (x>0) Anträge mit Status "neu" in der DB existieren, die dieser TestSachbearbeiter erstellt hat,
UND die Methode **getNeueAntraege** mit der ID des TestSachbearbeiters aufgerufen wird,
DANN sollte sie die Liste mit diesen x Anträgen zurückliefern.
- @Test: getNeueAntraege_01()

WENN ein TestSachbearbeiter bereits in der Datenbank existiert,
UND keine Anträge mit Status "neu" in der DB existieren, die dieser TestSachbearbeiter erstellt hat,
UND die Methode **getNeueAntraege** mit der ID des TestSachbearbeiters aufgerufen wird,
DANN sollte sie eine leere Liste zurückliefern.
- @Test: getNeueAntraege_02()

WENN ein TestSachbearbeiter nicht in der Datenbank existiert,
UND die Methode **getNeueAntraege** mit der ID des TestSachbearbeiters aufgerufen wird,
DANN sollte sie eine **leere Liste** zurückliefern.
- @Test: getAllAntraege_00()

WENN x (x>0) Anträge in der DB existieren,
UND die Methode **getAllAntraege** aufgerufen wird,
DANN sollte sie die Liste mit diesen x Anträgen zurückliefern.
- @Test: getAllAntraege_01()

WENN keine Anträge in der DB existieren,
UND die Methode **getAllAntraege** aufgerufen wird,
DANN sollte sie eine leere Liste zurückliefern.
- @Test: deleteAntrag_00()

WENN ein TestSachbearbeiter bereits in der Datenbank existiert,
UND ein TestAntrag mit Status "abgelehnt" und Typ "Kontolimit-Antrag" bereits der DB existiert,
UND der TestSachbearbeiter den TestAntrag erstellt hat,
UND die Methode **deleteAntrag** mit der ID des Testantrags und der ID des TestSachbearbeiters aufgerufen wird,
DANN sollte sie **TRUE** zurückliefern,
UND der TestAntrag sollte aus der DB entfernt sein.
- @Test: deleteAntrag_01()

WENN ein TestSachbearbeiter bereits in der Datenbank existiert,
UND ein TestAntrag mit Status "abgelehnt" und Typ "Kontoschließungs-Antrag" bereits der DB existiert,
UND der TestSachbearbeiter den TestAntrag erstellt hat,
UND die Methode `deleteAntrag` mit der ID des Testantrags und der ID des TestSachbearbeiters aufgerufen wird,
DANN sollte sie **TRUE** zurückliefern,
UND der TestAntrag sollte aus der DB entfernt sein.

- @Test: deleteAntrag_02()

WENN ein TestSachbearbeiter bereits in der Datenbank existiert,
UND ein TestAntrag mit Status "abgelehnt" und Typ "Kontoeröffnungs-Antrag" bereits der DB existiert,
UND der TestSachbearbeiter den TestAntrag erstellt hat,
UND die Methode `deleteAntrag` mit der ID des Testantrags und der ID des TestSachbearbeiters aufgerufen wird,
DANN sollte sie **TRUE** zurückliefern,
UND der TestAntrag sollte aus der DB entfernt sein.

- @Test: deleteAntrag_03()

WENN ein TestSachbearbeiter bereits in der Datenbank existiert,
UND ein TestAntrag mit Status "genehmigt" und Typ "Kontoeröffnungs-Antrag" bereits der DB existiert,
UND der TestSachbearbeiter den TestAntrag erstellt hat,
UND die Methode `deleteAntrag` mit der ID des Testantrags und der ID des TestSachbearbeiters aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern,
UND der TestAntrag sollte nicht aus der DB entfernt sein.

- @Test: deleteAntrag_04()

WENN ein TestSachbearbeiter bereits in der Datenbank existiert,
UND ein TestAntrag mit Status "abgelehnt" und Typ "Kontoschließungs-Antrag" bereits der DB existiert,
UND der TestSachbearbeiter den TestAntrag nicht erstellt hat,
UND die Methode `deleteAntrag` mit der ID des Testantrags und der ID des TestSachbearbeiters aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern,
UND der TestAntrag sollte nicht aus der DB entfernt sein.

- @Test: deleteAntrag_05()

WENN ein TestSachbearbeiter bereits in der Datenbank existiert,
UND ein TestAntrag mit Status "abgelehnt" und Typ "Sachbearbeiter-Anlegen" bereits der DB existiert,
UND die Methode `deleteAntrag` mit der ID des Testantrags und der ID des TestSachbearbeiters aufgerufen wird,
DANN sollte sie **FALSE** zurückliefern,
UND der TestAntrag sollte nicht aus der DB entfernt sein.

- @Test: getAbgelehnteAntraege_00()

WENN ein TestSachbearbeiter bereits in der Datenbank existiert,
UND x ($x > 0$) Anträge mit Status "abgelehnt" in der DB existieren, die dieser TestSachbearbeiter erstellt hat,
UND die Methode `getAbgelehnteAntraege` mit der ID des TestSachbearbeiters aufgerufen wird,
DANN sollte sie die Liste mit diesen x Anträgen zurückliefern.
- @Test: getAbgelehnteAntraege_01()

WENN ein TestSachbearbeiter bereits in der Datenbank existiert,
UND keine Anträge mit Status "abgelehnt" in der DB existieren, die dieser TestSachbearbeiter erstellt hat,
UND die Methode `getAbgelehnteAntraege` mit der ID des TestSachbearbeiters aufgerufen wird,
DANN sollte sie eine leere Liste zurückliefern.
- @Test: getAbgelehnteAntraege_02()

WENN ein TestSachbearbeiter nicht in der Datenbank existiert,
UND die Methode `getAbgelehnteAntraege` mit der ID des TestSachbearbeiters aufgerufen wird,
DANN sollte sie eine **leere Liste** zurückliefern.
- @Test: getBearbeiteteAntraege_00()

WENN ein TestSachbearbeiter bereits in der Datenbank existiert,
UND x ($x > 0$) Anträge mit Status "bearbeitet" in der DB existieren, die dieser TestSachbearbeiter erstellt hat,
UND die Methode `getBearbeiteteAntraege` mit der ID des TestSachbearbeiters aufgerufen wird,
DANN sollte sie die Liste mit diesen x Anträgen zurückliefern.
- @Test: getBearbeiteteAntraege_01()

WENN ein TestSachbearbeiter bereits in der Datenbank existiert,
UND keine Anträge mit Status "bearbeitet" in der DB existieren, die dieser TestSachbearbeiter erstellt hat,
UND die Methode `getBearbeiteteAntraege` mit der ID des TestSachbearbeiters aufgerufen wird,
DANN sollte sie eine leere Liste zurückliefern.
- @Test: getBearbeiteteAntraege_02()

WENN ein TestSachbearbeiter nicht in der Datenbank existiert,
UND die Methode `getBearbeiteteAntraege` mit der ID des TestSachbearbeiters aufgerufen wird,
DANN sollte sie eine **leere Liste** zurückliefern.
- @Test: getStornierteAntraege_00()

WENN ein TestSachbearbeiter bereits in der Datenbank existiert,
UND x ($x > 0$) Anträge mit Status "storniert" in der DB existieren, die dieser TestSachbearbeiter erstellt hat,
UND die Methode `getStornierteAntraege` mit der ID des TestSachbearbeiters aufgerufen

wird,

DANN sollte sie die Liste mit diesen **x** Anträgen zurückliefern.

- @Test: getStornierteAntraege_010
 - WENN** ein TestSachbearbeiter bereits in der Datenbank existiert,
 - UND** keine Anträge mit Status "storniert" in der DB existieren, die dieser TestSachbearbeiter erstellt hat,
 - UND** die Methode **getStornierteAntraege** mit der ID des TestSachbearbeiters aufgerufen wird,
 - DANN** sollte sie eine leere Liste zurückliefern.
- @Test: getStornierteAntraege_020
 - WENN** ein TestSachbearbeiter nicht in der Datenbank existiert,
 - UND** die Methode **getStornierteAntraege** mit der ID des TestSachbearbeiters aufgerufen wird,
 - DANN** sollte sie eine **leere Liste** zurückliefern.
- @Test: getGenehmigteAntraege_000
 - WENN** ein TestSachbearbeiter bereits in der Datenbank existiert,
 - UND** **x** ($x > 0$) Anträge mit Status "genehmigt" in der DB existieren, die dieser TestSachbearbeiter erstellt hat,
 - UND** die Methode **getGenehmigteAntraege** mit der ID des TestSachbearbeiters aufgerufen wird,
 - DANN** sollte sie die Liste mit diesen **x** Anträgen zurückliefern.
- @Test: getGenehmigteAntraege_010
 - WENN** ein TestSachbearbeiter bereits in der Datenbank existiert,
 - UND** keine Anträge mit Status "genehmigt" in der DB existieren, die dieser TestSachbearbeiter erstellt hat,
 - UND** die Methode **getGenehmigteAntraege** mit der ID des TestSachbearbeiters aufgerufen wird,
 - DANN** sollte sie eine leere Liste zurückliefern.
- @Test: getGenehmigteAntraege_020
 - WENN** ein TestSachbearbeiter nicht in der Datenbank existiert,
 - UND** die Methode **getGenehmigteAntraege** mit der ID des TestSachbearbeiters aufgerufen wird,
 - DANN** sollte sie eine **leere Liste** zurückliefern.