

Programmentwurf Kontoverwaltung

Namen: Braje, Patrick und Fischer, Simon
Matrikelnummern: 6956092 und 1155364

Abgabedatum: 04.05.2022

Kapitel 1: Einführung

Übersicht über die Applikation

Was macht die Applikation?

- Banken anlegen, anzeigen & löschen
- Kunden anlegen, anzeigen & löschen
- Konten verwalten
 - Konten anlegen, anzeigen & löschen
 - Pin ändern
 - Kontowechsel zu anderer Bank
- Überweisungen durchführen
- Geld einzahlen und abheben
- Kontoauszug anzeigen

Wie funktioniert die Anwendung?

- alle Eingaben werden in der Konsole/Terminal getätigt
- ist ein Befehl unvollständig, wird eine entsprechende Hilfe angezeigt
- auf jede Eingabe erfolgt eine Rückmeldung (SUCCESS / ERROR)

Folgende Befehle sind verfügbar:

KUNDE

GET

<KUNDENID>

CREATE

<VORNAME> <NACHNAME>

DELETE

<KUNDENID>

BANK

GET

<BANKID>

CREATE

<BANKID>

DELETE

<BANKID>

GIROKONTO

GET

<KONTOID>

CREATE

<BANKID> <KUNDENID> <PIN>

DELETE

<KONTOID>

CHANGEPIN

<OLD-KONTOID> <OLD-PIN> <NEW-PIN>

CHANGEBANK

<OLD-KONTOID> <NEW-BANKID>

GETHISTORY

<KONTOID>

TRANSAKTION

TRANSFER

<SENDERKONTOID> <EMPFÄNGERKONTOID> <BETRAG>

CASH_IN

<AUTOMATID> <KONTOID> <BETRAG>

CASH_OUT

<AUTOMATID> <KONTOID> <BETRAG> <PIN>

GELDAUSGABEAUTOMAT

GET

<AUTOMATID>

CREATE

<AUTOMATID> <BETRAG>

DELETE

<AUTOMATID>

Beispiel-Nutzung des Programms

- BANK CREATE testbank1
 - SUCCESS: bank created
- KUNDE CREATE Peter Maier
 - SUCCESS: kunde Cfe6wNj8H created
- GIROKONTO CREATE testbank1 Cfe6wNj8H 7777
 - SUCCESS: girokonto FnvMKQq1k created
- GELDAUSGABEAUTOMAT CREATE automat-bahnhof 10000.00
 - SUCCESS: geldausgabeautomat automat-bahnhof created
- TRANSAKTION CASH_IN automat-bahnhof FnvMKQq1k 50.00
 - SUCCESS: transfer complete
- GELDAUSGABEAUTOMAT GET automat-bahnhof
 - SUCCESS: Automat [AutomatID=automat-bahnhof, Betrag=Betrag [10050,0]]
- ...

Tipp: Es wird beim Absenden eines Befehls eine Hilfe angezeigt, sofern der Befehl nicht vollständig ist.

Welches Problem löst sie/welchen Zweck hat sie?

Durch die Anwendung ist es einfach möglich, verschiedene Kunden, Konten und Banken zu verwalten. Zusätzlich können Überweisungen, Einzahlungen und Auszahlungen durchgeführt werden.

Wie startet man die Applikation?

Die Applikation ist direkt startbereit und benötigt keine weitere Konfiguration. Die App wurde bereits kompiliert und befindet sich im Ordner "compiled".

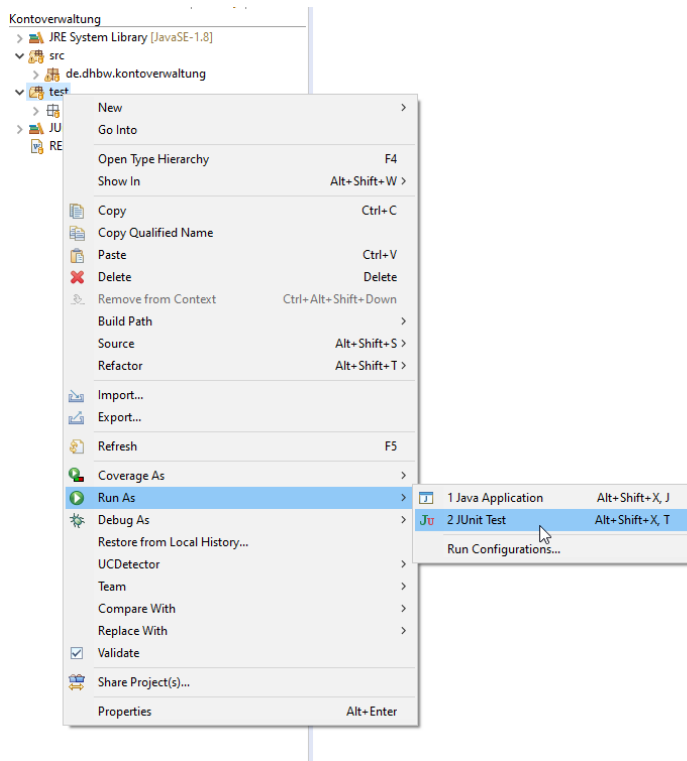
- cd compiled
- java -jar KontoverwaltungApp.jar

Danach können Befehle über die Befehlszeile eingegeben werden.

Wie testet man die Applikation?

Die Tests sind mit JUnit 5 realisiert. Am einfachsten testet man die Applikation während der Entwicklung in Eclipse.

Importiere das Projekt in Eclipse und klicke anschließend mit der rechten Maustaste auf den "test" Ordner im Paket-Explorer. Wähle anschließend "Run As" und dann "JUnit Test".



Kapitel 2: Clean Architecture

Was ist Clean Architecture?

Bei der Clean Architecture geht es darum, Abhängigkeiten zwischen den einzelnen Teilen der Anwendung aktiv zu gestalten. Dabei wird der Programmcode in mehrere Schichten aufgeteilt, wobei die inneren Schichten die äußeren Schichten nicht kennen. Somit lassen sich äußere Schichten leichter austauschen, ohne tiefgreifende Änderungen im Code umsetzen zu müssen.

Gängig ist die Verwendung von vier Schichten. Die innerste Schicht dabei, der Domain Code, enthält die Objekte und die Logik der Anwendung. Die nächste Schicht beinhaltet den Application Code, der die eigentlichen Use Cases und Anwendungen implementiert. Die dritte Schicht sorgt für die Vermittlung der Daten zwischen den inneren Schichten und den externen Plugins, weshalb sie auch Adapters genannt wird. Die äußerste Schicht, die Plugins, besteht nicht aus Anwendungslogik, sondern nur aus Frameworks und externen Schnittstellen und Bibliotheken.

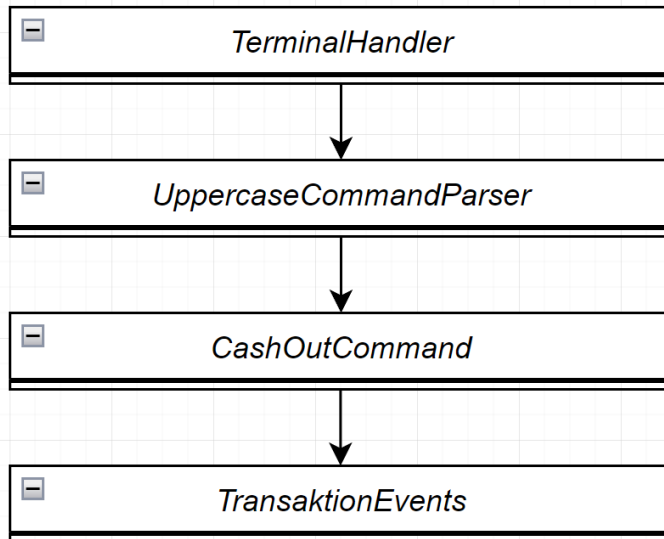
Analyse der Dependency Rule

Im folgenden Abschnitt folgt jeweils eine Klasse, die die Dependency Rule einhält und eine Klasse, die die Dependency Rule verletzt.

Positiv-Beispiel:

Klasse TransaktionEvents

Beispiel: TransaktionEvents



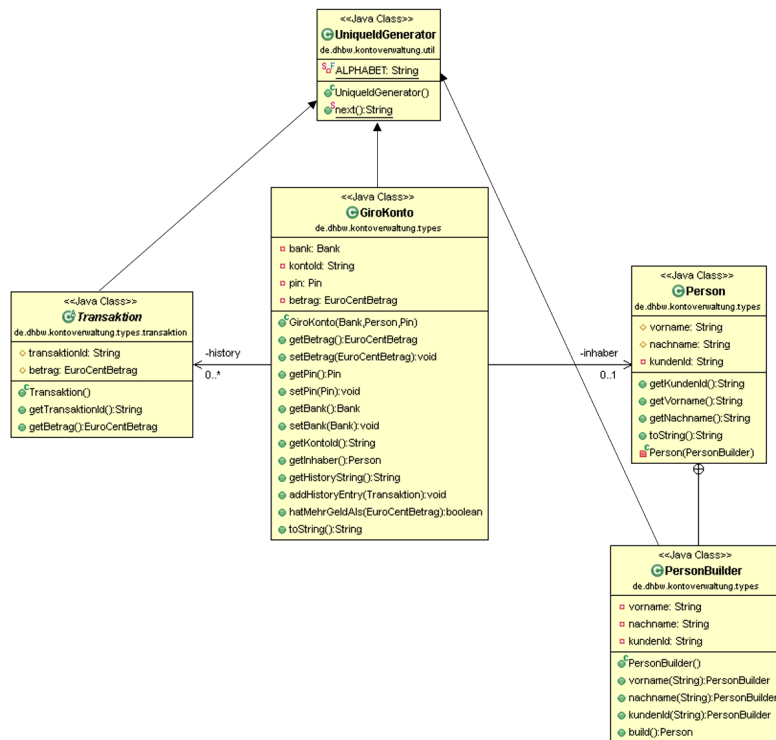
Analyse der Abhängigkeiten in beide Richtungen:

TerminalHandler benötigt *UppercaseCommandParser*, welcher *CashOutCommand* benötigt. Diese Klasse benötigt *TransaktionEvents*.

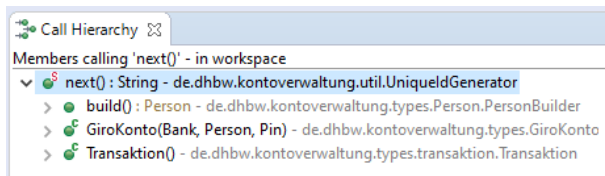
In der anderen Richtung gibt es keine Abhängigkeiten.

Negativ-Beispiel:

Klasse UniqueldGenerator



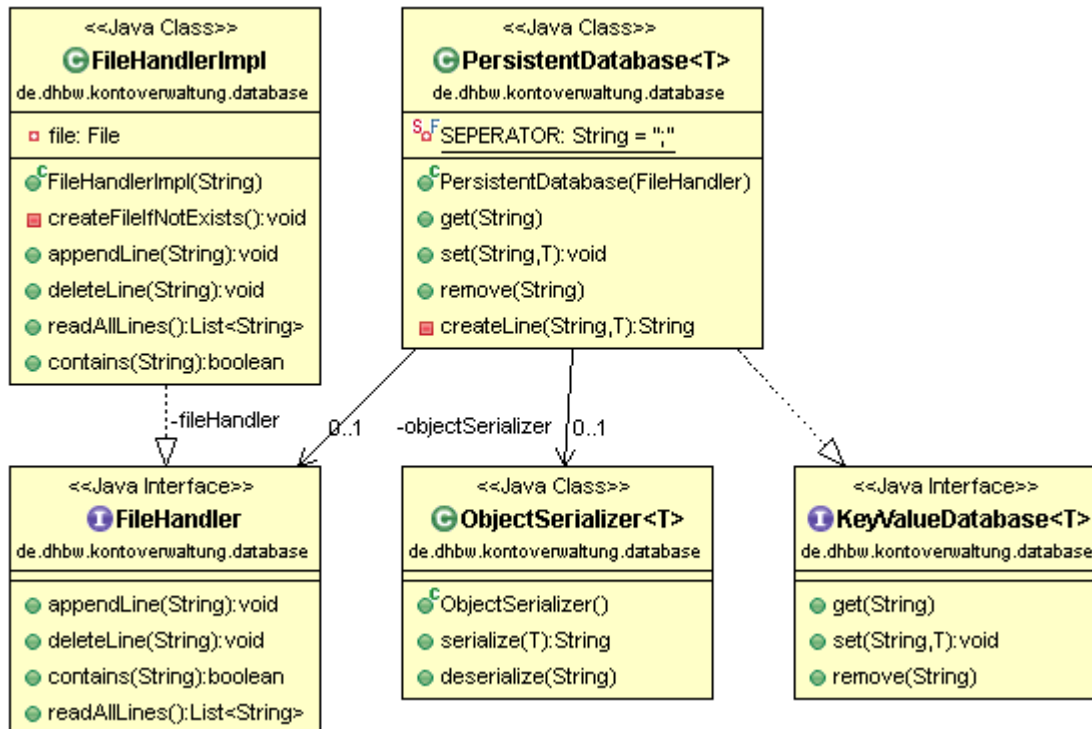
Analyse der Abhängigkeiten in beide Richtungen:



Die Klassen *PersonBuilder*, *GiroKonto* und *Transaktion* hängen von *UniqueldGenerator* ab. Die Klasse *UniqueldGenerator* hat dagegen keine Abhängigkeiten.

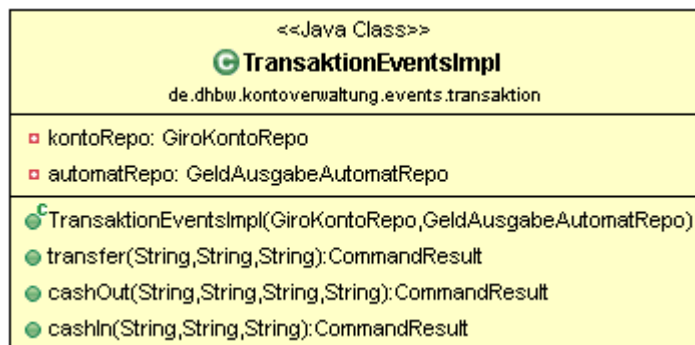
Analyse der Schichten

Schicht: Datenzugriffsschicht



Die Klasse *PersistentDatabase* implementiert das Interface *KeyValueDatabase* und stellt die Speicherung, das Löschen und das Laden von Objekten zur Verfügung. Die Klassen *ObjectSerializer* und *FileHandler* bzw. *FileHandlerImpl* unterstützen dabei aus technischer Sicht. Logik oder die Darstellung der Daten sind nicht in den Klassen enthalten.

Schicht: Business-Logik-Schicht



Die Klasse *TransaktionEventsImpl* implementiert die Logik, die beim Aufruf einer Methode, die im Zusammenhang mit einer Transaktion steht, ausgeführt wird. Der Aufruf erfolgt nach der Eingabe eines entsprechenden Befehls durch den Nutzer. Die Klasse hat keinen direkten Zugang zu den Daten und keinen Einfluss auf die Darstellung der Daten.

Kapitel 3: SOLID

Analyse Single-Responsibility-Principle (SRP)

Positiv-Beispiel


<<Java Class>>	
GetGeldAusgabeAutomatCommand	
de.dhbw.kontoverwaltung.terminal.process.geldausgabeautomat	
S	COMMAND_HELP_CUT: int = 2
S	ARG_AUTOMAT: int = 2
S	EXPECTED_LENGTH: int = 3
□	ausgabeAutomatEvents: GeldAusgabeAutomatEvents
C	GetGeldAusgabeAutomatCommand(GeldAusgabeAutomatEvents)
●	execute(SplittedCommand): CommandResult


Die Klasse *GetGeldAusgabeAutomatCommand* setzt nur den Aufruf der Methode *getGeldAusgabeAutomat* vom Interface *ausgabeAutomatEvents* um.


Negativ-Beispiel


<<Java Class>>	
GeldAusgabeAutomatEventsImpl	
de.dhbw.kontoverwaltung.events.geldausgabeautomat	
□	geldAusgabeAutomatRepo: GeldAusgabeAutomatRepo
C	GeldAusgabeAutomatEventsImpl(GeldAusgabeAutomatRepo)
●	getGeldAusgabeAutomat(String): CommandResult
●	createGeldAusgabeAutomat(String,String): CommandResult
●	deleteGeldAusgabeAutomat(String): CommandResult

Die Klasse *GeldAusgabeAutomatEventsImpl* implementiert die Logik aller Funktionen, die mit dem Geldausgabeautomat in Verbindung stehen. Die einzelnen Funktionen *getGeldAusgabeAutomat*, *createGeldAusgabeAutomat* und *deleteGeldAusgabeAutomat* könnten jeweils in eigene Klassen ausgelagert werden:

 <i>GeldAusgabeAutomatEvents (Interface)</i>
Methode: <code>getGeldAusgabeAutomat(String geldAusgabeAutomatId): CommandResult</code> Methode: <code>createGeldAusgabeAutomat(String geldAusgabeAutomatId, String betragFilled): CommandResult</code> Methode: <code>deleteGeldAusgabeAutomat(String geldAusgabeAutomatId): CommandResult</code>

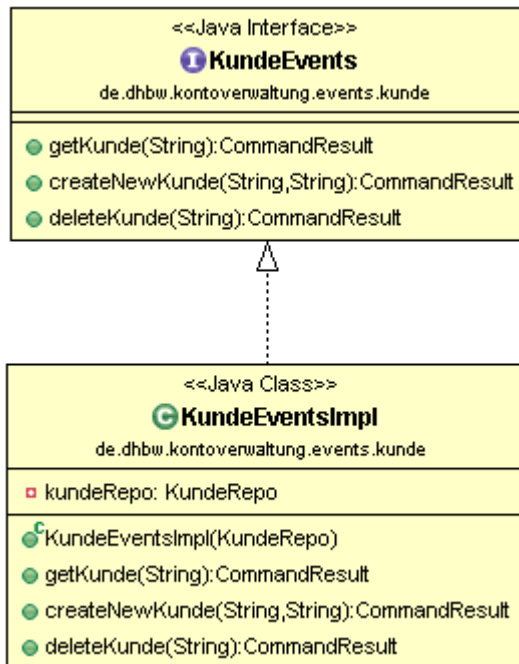
 <i>getGeldAusgabeAutomatImpl</i>
GeldAusgabeAutomatRepo geldAusgabeAutomatRepo
Konstruktur: <code>getGeldAusgabeAutomatImpl(GeldAusgabeAutomatRepo geldAusgabeAutomatRepo)</code> Methode: <code>execute(String geldAusgabeAutomatId)</code>

 <i>createGeldAusgabeAutomatImpl</i>
GeldAusgabeAutomatRepo geldAusgabeAutomatRepo
Konstruktur: <code>createGeldAusgabeAutomatImpl(GeldAusgabeAutomatRepo geldAusgabeAutomatRepo)</code> Methode: <code>execute(String geldAusgabeAutomatId, String betragFilled)</code>

 <i>deleteGeldAusgabeAutomatImpl</i>
GeldAusgabeAutomatRepo geldAusgabeAutomatRepo
Konstruktur: <code>deleteGeldAusgabeAutomatImpl(GeldAusgabeAutomatRepo geldAusgabeAutomatRepo)</code> Methode: <code>execute(String geldAusgabeAutomatId)</code>

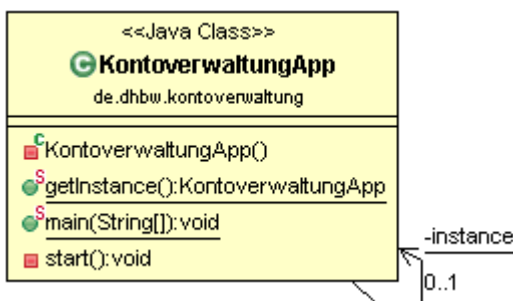
Analyse Open-Closed-Principle (OCP)

Positiv-Beispiel



Die vorhandenen Events werden in einem Interface definiert, das wiederum von einer eigenen Klasse implementiert wird. Eine Erweiterung der Funktionen ist jederzeit über das Interface möglich. Eine fehlende Implementierung kann schnell erkannt und nachträglich hinzugefügt werden. Zudem entkoppelt das Interface den Aufruf der Funktion aus dem Bereich des "Befehlsparasing".

Negativ-Beispiel

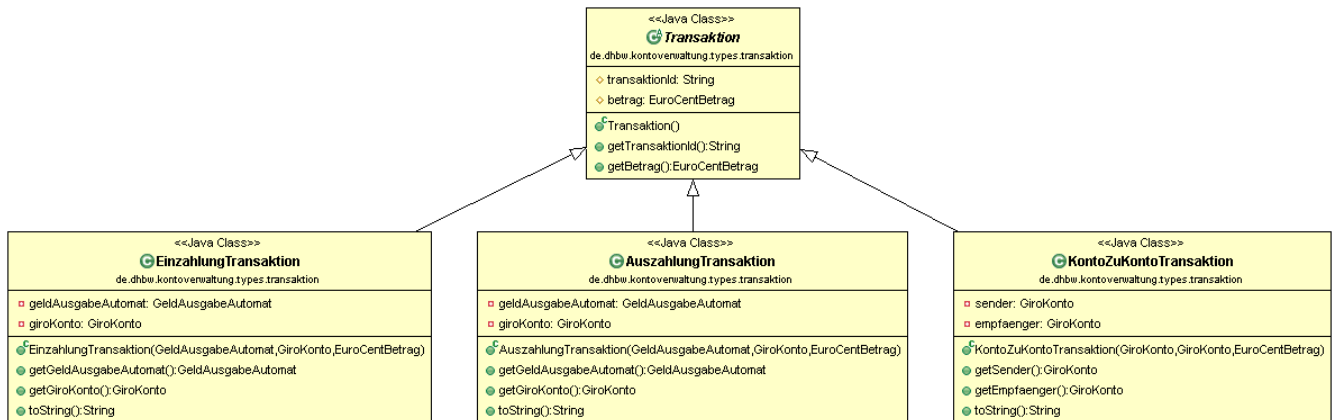


Die Main-Klasse der Anwendung generiert unter anderem die erforderlichen Repos und Events-Implementations-Klassen. Eine Erweiterung der Klasse ist aufgrund der starren Konstruktoren der aufgerufenen Klassen nur schwer möglich. Eine Lösung für das Problem wäre, falls in dem Fall möglich, einen weiteren Thread mit der zusätzlichen Aufgabe zu starten, der unabhängig vom Rest des Programms ausgeführt werden kann. Auch die Ausführung einer Methode vor der Erzeugung der restlichen Klassen wäre möglich.

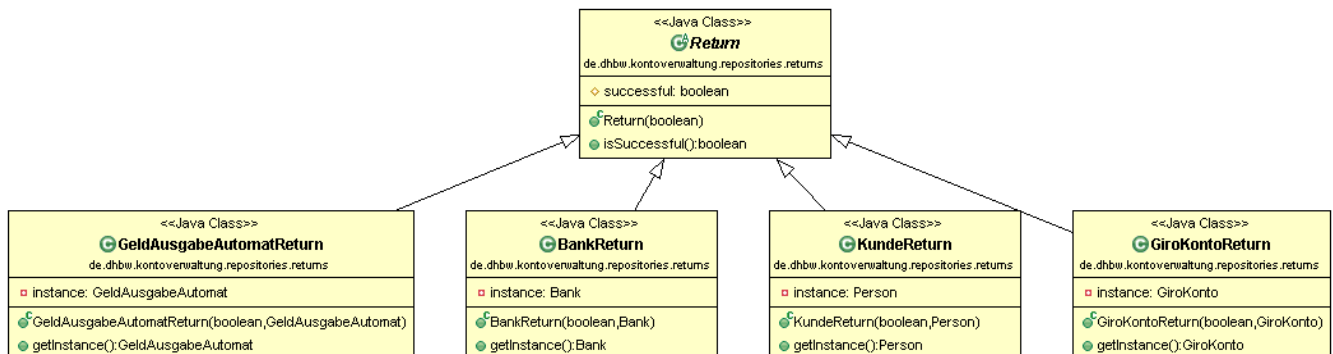
Analyse Liskov-Substitution-Principle (LSP)

Die Unterklassen verwenden jeweils alle Attribute und Methoden der Oberklasse. Zusätzlich definieren die Unterklassen zusätzliche Attribute und/oder Methoden. Ein Objekt der Unterklasse kann anstelle eines Objekts ihrer Oberklasse eingesetzt werden (Substitutionsprinzip), umgekehrt ist dies nicht möglich.

1. Positiv-Beispiel



2. Positiv-Beispiel

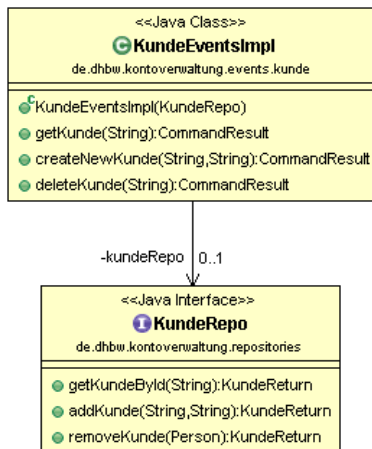


Kapitel 4: Weitere Prinzipien

Analyse GRASP: Geringe Kopplung

Positiv-Beispiel

Klasse: *KundeEventsImpl*



Aufgabenbeschreibung:

Die Klasse *KundeEventsImpl* implementiert die Geschäftslogik für die Kunden-Operationen. Sie benötigt daher auch Zugriff auf das Repository der Kunden.

Umsetzung:

Durch Dependency Injection konnte eine geringe Kopplung realisiert werden. Das benötigte Repository *KundeRepo* wird beim erzeugen der Klasse im Konstruktor mitgegeben.

Negativ-Beispiel

Klasse: *CommandResult*

Members calling constructors of 'CommandResult' - in workspace

- ▼  *CommandResult*(boolean, String) - de.dhbw.kontoverwaltung.terminal.command.results.CommandResult
 - >  *error*(String) : *CommandResult* - de.dhbw.kontoverwaltung.terminal.command.results.CommandResult
 - >  *ObjectToStringCommandResult*(Object) - de.dhbw.kontoverwaltung.terminal.command.results.ObjectToStringCommandResult
 - >  *success*(String) : *CommandResult* - de.dhbw.kontoverwaltung.terminal.command.results.CommandResult
- ▼  *CommandResult*(boolean) - de.dhbw.kontoverwaltung.terminal.command.results.CommandResult
 - >  *CommandResult*(boolean, String) - de.dhbw.kontoverwaltung.terminal.command.results.CommandResult

Aufgabenbeschreibung:

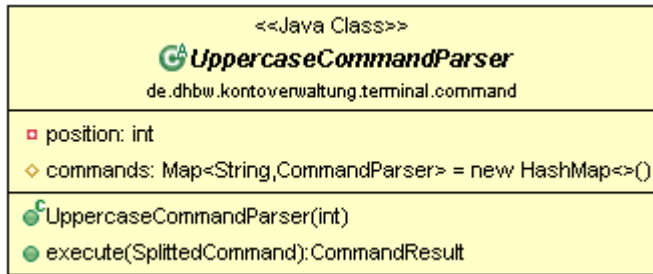
Die Klasse *CommandResult* stellt das Ergebnis (positiv und negativ) für einen ausgeführten Command dar. Durch die statischen Methoden (welche von vielen Stellen im Programm aufgerufen werden) lässt sich ein *CommandResult* Objekt erzeugen.

Begründung:

Im Programm werden in der aktuellen Realisierung an vielen Stellen *CommandResult* Objekte benötigt. Daher ist eine geringe Kopplung hier aktuell nur mit sehr großem Aufwand (Umstrukturierung des Programms) möglich. Auf diesen Aufwand haben wir verzichtet.

Analyse GRASP: Hohe Kohäsion

Klasse: *UppercaseCommandParser*



Begründung:

Die Kohäsion an dieser Stelle ist hoch, da sich die Programmeinheit lediglich verantwortlich für genau eine wohldefinierte Aufgabe ist.

In diesem Fall ist die Klasse *UppercaseCommandParser* für das Auslesen eines Strings an einer bestimmten Stelle eines Commands verantwortlich. Weitere Verantwortlichkeiten liegen nicht vor.

Don't Repeat Yourself (DRY)

Commit:

<https://github.com/Kontoverwaltung/Kontoverwaltung/commit/c2c6a7c0574c30c0de0df055f56a54c46d4c0398>

Begründung:

Code, der doppelte Einträge enthält, ist schwieriger zu warten. Wenn eine Wartung durchgeführt wird besteht die Gefahr, dass eine Kopie des Codes aktualisiert wird, ohne dass weitere Instanzen desselben Codes geändert werden. Dies würde möglicherweise in Bugs resultieren. Daher haben wir uns dazu entschieden, die identisch definierten Objekte und Strings einmalig global auszulagern.

Vorher:

```
@Test
void testSuccessData() {
    CommandResult target = CommandResult.success("test ok");
    assertThat(target.isSuccessful(), is(true));
    assertThat(target.getAdditionalInfo(), is("test ok"));
}

@Test
void testSuccessToString() {
    CommandResult target = CommandResult.success("test ok");
    assertThat(target.toString(), is("SUCCESS: test ok"));
}
```

Nachher:

```
private static final String TEST_OK = "test ok";
private CommandResult targetOk = CommandResult.success(TEST_OK);

@Test
void testSuccessData() {
    assertThat(targetOk.isSuccessful(), is(true));
    assertThat(targetOk.getAdditionalInfo(), is(TEST_OK));
}

@Test
void testSuccessToString() {
    assertThat(targetOk.toString(), is("SUCCESS: " + TEST_OK));
}
```

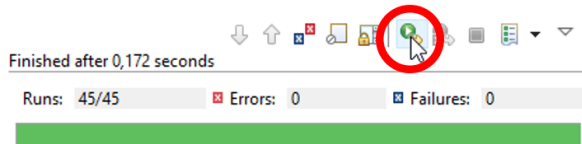
Kapitel 5: Unit Tests

10 Unit Tests

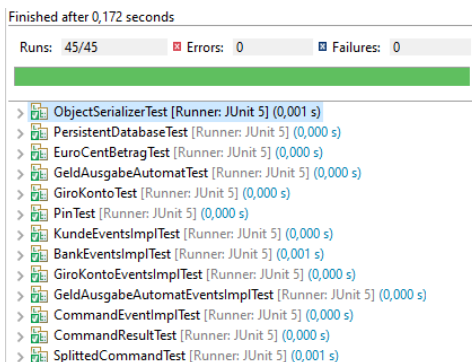
Unit Test	Beschreibung
KundeEventsImplTest# testCreateKunde	Prüft den korrekten Aufruf des Kunden-Events beim Erzeugen eines neuen Kunden (Verwendung eines MockObjekts)
PersistentDatabaseTest# testOneElement	Hinzufügen eines Objekts zur Datenbank, anschließend auslesen und Werte auf Gleichheit prüfen, danach entfernen des Objekts aus der Datenbank und Überprüfung auf null beim Auslesen
PersistentDatabaseTest# testMultipleElements	Hinzufügen mehrerer Objekte zur Datenbank, anschließend auslesen und Werte auf Gleichheit prüfen, danach entfernen der Objekte aus der Datenbank und Überprüfung auf null beim Auslesen
ObjectSerializerTest# test	Serialisieren und anschließend deserialisieren von einem Objekt und Werte auf Gleichheit prüfen
CommandResultTest# testPossibilities	Der Test prüft die korrekte Konkatenation der Befehl-Vervollständigungs-Hilfe.
CommandResultTest# testSuccessToString	Prüfung der korrekten Zusammensetzung der Zusatzinfo im Erfolgsfall.
CommandResultTest# testSuccessData	Prüft die Befehls-Rückgabe im Erfolgsfall. Geprüft wird ob das Flag "isSuccessful" korrekt auf "true" gesetzt wird. Zusätzlich wird geprüft, ob die Zusatzinfo korrekt gesetzt wurde.
CommandResultTest# testErrorData	Prüft die Befehls-Rückgabe im Fehlerfall. Geprüft wird ob das Flag "isSuccessful" korrekt auf "false" gesetzt wird. Zusätzlich wird geprüft, ob die Fehlerinfo korrekt gesetzt wurde.
PinTest#testWrongPin	Setzen einer Pin, Eingabe einer falschen Pin und Überprüfung auf false
PinTest#testCorrectPin	Setzen einer Pin, Eingabe der korrekten Pin und Überprüfung auf true

ATRIP: Automatic

- nur ein Knopfdruck zum Starten der Tests
 - schnelles & ständiges Wiederholen der Tests einfach möglich



- Tests laufen vollautomatisch (falls nötig in korrekter Reihenfolge) ab
- Tests überprüfen sich selbst auf bestanden/fehlgeschlagen



ATRIP: Thorough

Positiv

BankEventsImplTest

Begründung:

Kernlogik der Software wird komplett abgekapselt vom Rest getestet. Durch gemockte Daten wurden Abhängigkeiten reduziert. Auch das Zusammenspiel der unterschiedlichen Methoden wird durch die Angabe der Test-Reihenfolge berücksichtigt.

Der Test deckt alle Funktionen (create, get, delete) der zu testenden Klasse ab.

Code-Beispiel:

```
private static final String BANK_NAME = "testbank";

@Test
@Order(2)
void testCreateBank() {
    CommandResult result = target.createBank(BANK_NAME);
    assertThat(result.isSuccessful(), is(true));
    assertThat(result.getAdditionalInfo(), is("bank created"));
}

@Test
@Order(3)
void testBankFound() {
    CommandResult result = target.getBank(BANK_NAME);
    assertThat(result.isSuccessful(), is(true));
    assertThat(result.getAdditionalInfo(), is("Bank [Name=testbank]"));
}
```

Negativ

SplittedCommandTest

Begründung:

Beim Prüfen der Testabdeckung ist aufgefallen, dass einige Methoden der SplittedCommand nicht vom Test getestet werden. Ebenfalls werden Randfälle (z. B. negative Zahlen) nicht getestet, somit ist das Verhalten im Fehlerfall unklar bzw. nicht getestet.

Code-Beispiel:

```
// nicht vollständig getestete Methoden:
argSize();
getStringUppercaseAt(int pos);
getStringAt(int pos);
```

ATRIP: Professional

Positiv:

(Der folgende Code wurde vereinfacht, vollständig verfügbar in der Klasse *GeldAusgabeAutomatEventsImplTest*)

```
@Test
void testAutomatNotFound() {
    CommandResult result = target.getGeldAusgabeAutomat(AUTOMAT_ID);
    assertThat(result.isSuccessful(), is(false));
    assertThat(result.getAdditionalInfo(), is("failed to load geldausgabeautomat"));
}
```

Analyse & Begründung:

Der abgedruckte Test prüft keinen Erfolgsfall, sondern einen Fehlerfall.

Im Test wird ein Automat mit einer bestimmten ID angefordert. Dieser Automat ist allerdings nicht vorhanden, daher wird eine Fehlermeldung ausgegeben.

Es ist professionell, auch Fehlerfälle zu testen und deren korrekte Funktion zu prüfen.

Negativ:







```
private Pin target = new Pin("22tUzX");

@Test
void testWrongPin() {
    assertThat(target.isCorrectPin("bGs22Ax"), is(false));
}
```

Analyse & Begründung:

Der Pin-Test testet eine Methode, die einzig aus einem Aufruf der String-Methode equals besteht. Der Test ist daher wenig aussagekräftig und nicht wichtig für die Sicherstellung der Codequalität der Kernlogik des Programms.

Code Coverage

Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
✓  Kontoverwaltung	 64,1 %	3.280	1.835	5.115
>  src	 48,6 %	1.632	1.724	3.356
>  test	 93,7 %	1.648	111	1.759

48,6% im Programmcode (≈ 50%)

Eine Testabdeckung zu 100% ist nicht sinnvoll, weil zum Beispiel einfache Methoden wie Getter und Setter nicht getestet werden sollten. Auch ein Teil der Eingabeverarbeitung wird nicht getestet, da er zu großen Teilen aus Code von Java und keiner komplexen Logik besteht.

Fakes und Mocks

Klasse: *BankEventsImplTest*

```
<<Java Class>>
G BankEventsImplTest
de.dhbw.kontoverwaltung.events.bank

S F BANK_NAME: String = "testbank"
S bank: Bank = null
target: BankEventsImpl = new BankEventsImpl(new BankRepo() {
    @Override
    public BankReturn removeBank(Bank bankRequest) {
        assertThat(bankRequest.getName(), is(BANK_NAME));
        bank = null;
        return new BankReturn(true, bank);
    }

    @Override
    public BankReturn getBankByName(String bankName) {
        if (bank != null && bankName.equals(BANK_NAME)) {
            return new BankReturn(true, bank);
        } else {
            return new BankReturn(false, null);
        }
    }

    @Override
    public BankReturn addBank(String bankName) {
        assertThat(bankName, is(BANK_NAME));
        bank = new Bank(bankName);
        return new BankReturn(true, bank);
    }
})

C BankEventsImplTest()
testBankNotFound1():void
testCreateBank():void
testBankFound():void
testBankDelete():void
testBankNotFound2():void
bankNotFound():void
```

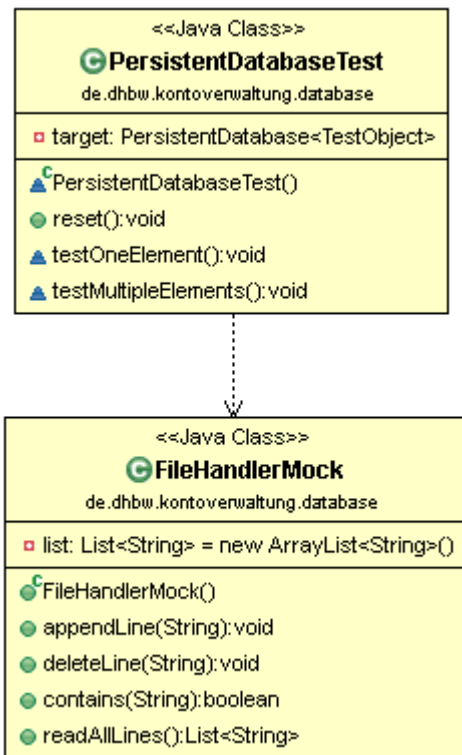
Analyse:

Die Tests in der Klasse *BankEventsImplTest* prüfen die korrekten Aufrufe der Bank-Events. Für die Ausführung der zu testenden Methoden ist zwingend ein Datenbankzugriff notwendig.

Begründung des Einsatzes:

Um keinen realen Datenbankzugriff durchzuführen (und damit Dateien auf die Festplatte zu schreiben) wurde ein Mock-Objekt eingesetzt. Somit existiert keine Abhängigkeit zu einer Datenbank.

Klasse: *PersistentDatabaseTest*



Analyse:

Die Tests in der Klasse prüfen die Funktion der Implementierung der Textdatei-Datenbank. Ohne Einsatz eines Mock-Objektes würde bei jedem Ausführen der Tests eine Datei auf die Festplatte geschrieben werden.

Begründung des Einsatzes:

Um beim Ausführen der Tests keine realen Textdateien auf die Festplatte zu schreiben, wurde der Mock *FileHandlerMock* erstellt. Dieser implementiert alle Methoden des *FileHandler* Interfaces, schreibt dabei aber keine Dateien auf die Festplatte, sondern lediglich in den Arbeitsspeicher.

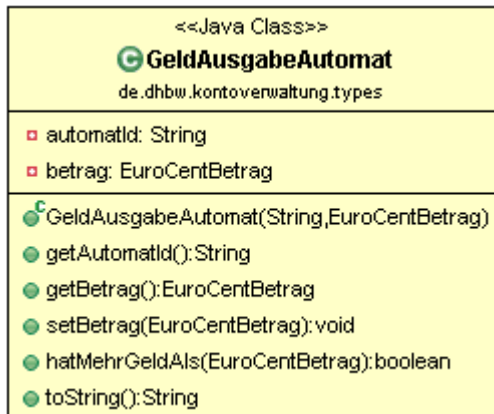
Kapitel 6: Domain Driven Design

Ubiquitous Language

Bezeichnung	Bedeutung	Begründung
GeldAusgabeAutomat	Geldautomat, an dem Kunden Geld ein- und auszahlen können	Der Begriff ist weit verbreitet und wird auch von Bankmitarbeitern verwendet und verstanden
EuroCentBetrag	Geldbetrag, bestehend aus einem Euro- und einem Centanteil	Es wird klar gemacht, dass es sich um einen Geldbetrag handelt, der in Euro angegeben wird
GiroKonto	Girokonto eines Kunden	Eindeutige Bezeichnung des Bankkontos, klare Abgrenzung zu anderen Konten
KontoZuKontoTransaktion	Transaktion zwischen zwei Konten, z.B. Überweisung	Die Art der Transaktion wird eindeutig benannt und gegenüber anderen Transaktionen abgegrenzt , z.B. Auszahlung am Geldautomaten

Entities

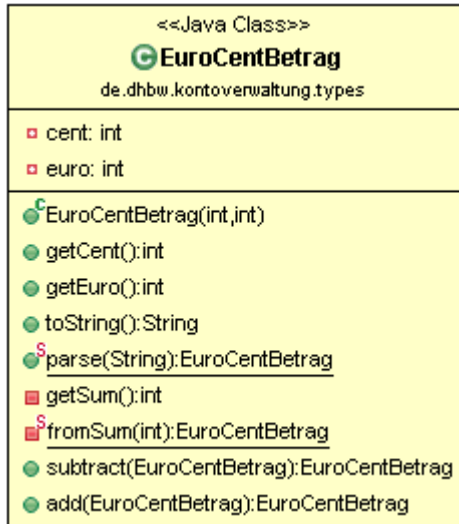
Klasse: *GeldAusgabeAutomat*



Jeder Automat wird über eine eindeutige ID identifiziert. Aufgrund der eindeutigen ID lassen sich die verschiedenen Automaten eindeutig ansprechen. Eine ungültige Generierung des Objekts oder Versetzung in einen ungültigen Zustand ist nicht möglich.

Value Objects

Klasse: *EuroCentBetrag*



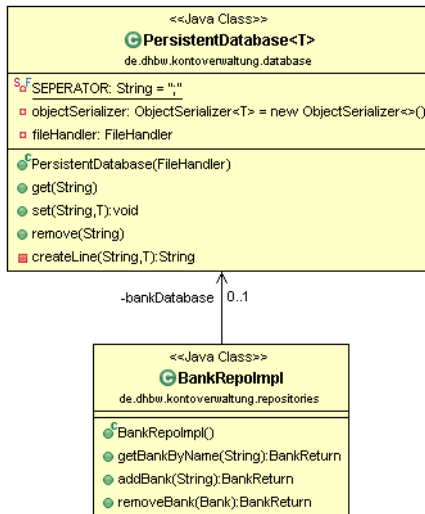
Betrag wird einmalig erstellt, bei einer Addition oder Subtraktion wird ein neues Betragsobjekt zurückgegeben.

Begründung:

Der Betrag wurde unveränderlich (immutable) aufgebaut, um den Programmierer zu zwingen, bei Betragsänderungen das neue Objekt aktiv erneut in die Datenbank zu schreiben. Somit werden unerwünschte Seiteneffekte vermieden. Zusätzlich ist das Objekt gekapselt und behält seinen Wert dauerhaft.

Repositories

Klasse: BankRepoImpl



Das eingesetzte Repository vermittelt zwischen der Domäne und dem Datenmodell. Durch diese Abstraktion ist eine Änderung der Art der Speicherung (z.B. von CSV auf SQL) einfach realisierbar. Der konkrete technische Zugriff wird durch das Repository vor dem Rest der Anwendung verborgen.

Aggregates

Im Projekt wurden keine Aggregates verwendet. Stattdessen wurde auf eine starke Trennung der einzelnen Module geachtet. Ein Vorteil dieses Vorgehens ist die hohe Entkopplung der Klassen. Viele Objekte, z.B. ein Betrag, werden an mehreren Stellen des Programms benötigt (Kontoauszug, Transaktion,...). Eine "Zuordnung" zu einem Aggregate wäre in diesem Fall nicht sinnvoll. Diese Problematik trifft auch auf weite Teile der restlichen Anwendung zu, unter anderem das Girokonto.

Kapitel 7: Refactoring

Code Smells

1. Dead Code:

<https://github.com/Kontoverwaltung/Kontoverwaltung/commit/8b8739415b75e0c553607d1a6dc95f4919cc936d>

Die Methode wird im gesamten Projekt nicht aufgerufen.

Genommener Lösungsweg:

Die nicht verwendete Methode wurde aus dem Code entfernt. Somit wurde ebenfalls die Größe des Codes reduziert und Wartungen sind einfacher möglich.

Vorher:

```
public static CommandResult notFound(String name) {  
    return error(name + " not found");  
}
```

Nachher:

```
// toter Code wurde entfernt
```

2. Duplicated Code:

<https://github.com/Kontoverwaltung/Kontoverwaltung/commit/c2c6a7c0574c30c0de0df055f56a54c46d4c0398>

Vorher:

```
@Test
void testErrorData() {
    CommandResult target = CommandResult.error("test error");
    assertThat(target.isSuccessful(), is(false));
    assertThat(target.getAdditionalInfo(), is("test error"));
}

@Test
void testErrorToString() {
    CommandResult target = CommandResult.error("test error");
    assertThat(target.toString(), is("ERROR: test error"));
}
```

Nachher:

```
private static final String TEST_ERROR = "test error";
private CommandResult targetError = CommandResult.error(TEST_ERROR);

@Test
void testErrorData() {
    assertThat(targetError.isSuccessful(), is(false));
    assertThat(targetError.getAdditionalInfo(), is(TEST_ERROR));
}

@Test
void testErrorToString() {
    assertThat(targetError.toString(), is("ERROR: " + TEST_ERROR));
}
```

2 Refactorings

1. Umbenennung von den Packages

<https://github.com/Kontoverwaltung/Kontoverwaltung/commit/970fa2bd3bc6d4a12112d8406a5369c6f451c4a0>


Umbenennung Packages Automat → Geldausgabeautomat und Konto → Girokonto

Für die Übersichtlichkeit und die Einheitlichkeit der Bezeichnungen (Ubiquitous Language)


2. "Magic Numbers" entfernt

<https://github.com/Kontoverwaltung/Kontoverwaltung/commit/f19b2e2a2f502f0a6df372b0d17adcb70b46dace>

Vorher:

	<i>ChangeBankKontoCommand</i>
KontoEvents kontoEvents	
Konstruktur: ChangeBankKontoCommand(KontoEvents kontoEvents) Methode: execute(SplittedCommand command)	

Nachher:

	<i>ChangeBankKontoCommand</i>
int COMMAND_HELP_CUT int ARG_KONTO_ID int ARG_BANK int EXPECTED_LENGTH KontoEvents kontoEvents	
Konstruktur: ChangeBankKontoCommand(KontoEvents kontoEvents) Methode: execute(SplittedCommand command)	

Kapitel 8: Entwurfsmuster

Entwurfsmuster: Builder

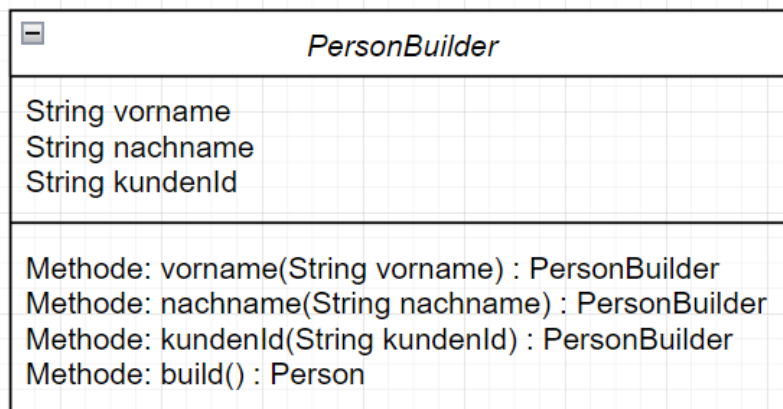
<https://github.com/Kontoverwaltung/Kontoverwaltung/commit/b4a807eee216d2124030cfbfdce0d43efa4b1f57>

Durch den Einsatz des Builder-Patterns kann nun besser kontrolliert werden, wie eine Person erzeugt wird. Sowohl der Vorname als auch der Nachname werden zusätzlich geprüft und müssen gesetzt sein. Wird keine KundenID angegeben, wird eine neue erzeugt.

```
public Person build() {  
    if (this.vorname == null) {  
        throw new NullPointerException();  
    }  
    if (this.nachname == null) {  
        throw new NullPointerException();  
    }  
    if (this.kundenId == null) {  
        this.kundenId = UniqueIdGenerator.next();  
    }  
    return new Person(this);  
}
```

```
Person person = new PersonBuilder().vorname(vorname).nachname(nachname).build();
```

UML:



Entwurfsmuster: Singleton


<https://github.com/Kontoverwaltung/Kontoverwaltung/commit/e8ad3491726edbab70b78ec66c8a53ab25c5a0c7>

Die Start-Klasse ("KontoverwaltungApp") unserer Anwendung eignet sich für die Nutzung des Singleton Entwurfsmusters, da nie mehr als eine Instanz davon im laufenden Programm existieren sollte. Die Methode "getInstance" erzeugt beim ersten Aufrufen eine Instanz, bei allen weiteren Aufrufen gibt sie diese Instanz zurück.

```
private static KontoverwaltungApp instance = null;

private KontoverwaltungApp() {
}

public static KontoverwaltungApp getInstance() {
    if (instance == null) {
        instance = new KontoverwaltungApp();
    }
    return instance;
}
```

	<i>KontoverwaltungApp</i>
KontoverwaltungApp instance	
Methode: getInstance() : KontoverwaltungApp	
Methode: start()	