**Protokoll Übung 2**

Gruppe: **Schwabakla**

– Robert Dziuba, Tobias Klatt & Inga Schwarze

1. **Absicherung der Klasse AddressBook und Benutzerschnittstelle**

Anmerkung: Der Quellcode befindet sich dieses Mal nur teilweise in dieser Datei. Er kann komplett der zweiten hochgeladenen Datei entnommen werden, die in Eclipse importiert wird.

Aufgabenteil A:

Wir ergänzen unser AddressBook jetzt um die Search-Methode. Wir hatten diese Methode zuvor komplett aus unserem Interface gestrichen, damit wir sie nicht implementieren müssen. Wir fügen sie nun wieder dem Interface hinzu und implementieren sie in unserer Klasse AddressBook. Die Search-Methode sucht im Adressbuch nach Kontakten und zeigt an, was dazu eingetragen wurde.

@Override

**public** ContactDetails[] search(String keyPrefix)

**throws** ParameterStringIsEmptyException, DetailsNotFoundException {

// keyPrefix wird hier aufbereitet: trim und toLowerCase

keyPrefix = (!keyPrefix.isEmpty()) ? keyPrefix.trim().toLowerCase()

: "";

// erstellen wir eine ArrayList, da wir noch nicht wissen wie viele

// Kontakte wir finden werden

List<ContactDetails> matchedDetails = **new** ArrayList<ContactDetails>();

// wir teilen unsere Suchparameter in die Teile die durch ein

// Leerzeichen getrennt wurden

String[] keyPrefixArray = keyPrefix.split(" ");

// und schauen jeden einzelnen Teilprefix an

**for** (String prefix : keyPrefixArray) {

// wir holen uns alle Keys

**for** (String key : namesMap.keySet()) {

// und wenn ein key die Suchparameter enthält

**if** (key.contains(prefix)) {

// splitten sie in ihre einzelnen Bestandteile

// (hans::müller::123456789 -> hans, müller, 123456789)

String[] keyArray = key.split("::");

// und holen uns die ContactDetails

ContactDetails details = **this**.getDetails(keyArray[0],

keyArray[1], keyArray[2], keyArray[3], keyArray[4]);

**if** (!matchedDetails.contains(details)) {

matchedDetails.add(details);

}

}

}

}

// wir geben ein Array mit all unseren Kontakten zurück

**return** matchedDetails

.toArray(**new** ContactDetails[matchedDetails.size()]);

}

Aufgabenteil B:

Um unser Adressbuch nun um geeignete Exceptions zu erweitern, nehmen wir zunächst alle alten Exceptions aus Übung 1 aus dem AddressBook heraus. Die Klasse ContactDetails soll keine Exceptions mehr werfen, das AddressBook soll sich um alle Fehler kümmern.

Die Aufgabe der Klasse AddressBook ist es, Kontakte vom Typ ContactDetails zu speichern, zu ändern, zu suchen und zu löschen. Dabei werden die Kontakte durch einen Schlüssel in der Treemap gespeichert, der sich aus allen ContactDetail Attributen (Namen, Nachnamen,Telefonnummer, E-Mail und Adresse) ergibt. Damit die Originalität des Schlüssels gewährleistet wird, werden einzelnen Werte durch einen doppelten Doppelpunkt getrennt.

(z.B. hans::müller::123456789::rob@web.de::Sonntagstr.12). Dies ermöglicht uns auch direkt in den Schlüsseln zu suchen, da der Schlüssel immer noch die "Klardaten" enthält. Somit müssen die Kontakte nicht mehr doppelt gespeichert werden und es ist möglich Kontakte mit gleichen Vornamen oder Nachnamen zu speichern *(Aufgabenteil 1d gelöst).*

Aus unserer Fehlertabelle (siehe Aufgabe 2a) ergaben sich für uns folgende sinnvolle Exceptions, allesamt vom Typ Exception.

**DetailsNotFoundException:** Unter dem eingegebenen Key können keine passenden Kontaktdaten gefunden werden. Sie wird geworfen, wenn ContactDetails zurück gegeben werden sollen, diese aber nicht in der Map enthalten sind.

**ParameterStringIsEmptyException:** Wird immer dann geworfen, wenn der Aufrufer einer Methode im Parameter null oder ““ übergibt. Es ist Pflicht, einen Vor- oder einen Nachnamen einzugeben. Ist dies nicht der Fall, wird diese Exception geschmissen.

**DuplicateKeyException:** Ein Key ist schon vorhanden. Doppelte Einträge werden nicht akzeptiert. Wenn ein Kontakt also schon vorhanden ist, kann kein zweiter identischer Kontakt gespeichert werden (logisch!).

**InvalidContactException:** Diese Exception taucht nur in den Methoden addDetails und changeDetails auf. Sie wird geschmissen, wenn keine Kontaktdaten vorhanden sind. Sie ist jeweils nur in der else-Verzweigung vorhanden  „Letzter Ausweg“

**KeyIsNotInUseException:** Ein Kontakt kann unter der Eingabe eines bestimmten Keys nicht gefunden werden. Diese Exception greift beim Ändern und beim Löschen, sofern es keinen Kontakt gibt, der zum eingegebenen Key passt.

**Hier eine kurze Übersicht, welche Methode welche Exception wirft:**

**getDetails(String key)**

wirft DetailsNotFoundException & ParameterStringIsEmptyException

**getDetails(String name, String lastname, String phone, String mail, String address)**

wirft DetailsNotFoundException & ParameterStringIsEmptyException

**keyInUse(String key)**

wirft ParameterStringIsEmptyException

**addDetails(ContactDetails details)**

wirft DuplicateKeyException, InvalidContactException, ParameterStringIsEmptyException

**changeDetails(String oldkey, ContactDetails details)**

wirft DuplicateKeyException, InvalidContactException, KeyIsNotInUseException, ParameterStringException

**ContactDetails[] search(String keyPrefix)**

wirft ParameterStringIsEmptyException, DetailsNotFoundException

**removeDetails(String key)**

wirft KeyIsNotInUseException, ParameterStringIsEmptyException

**getNumberOfEntries()**

wirft keine Exception

**generateKey(String name, String lastname, String phone, String mail, String address)**

wirft ParameterStringIsEmptyException

**generateKey(ContactDetails details)**

wirft ParameterStringIsEmptyException

**getCleanParameter(String key) [nicht im Interface, Hilfsfunktion]**

wirft ParameterStringIsEmptyException

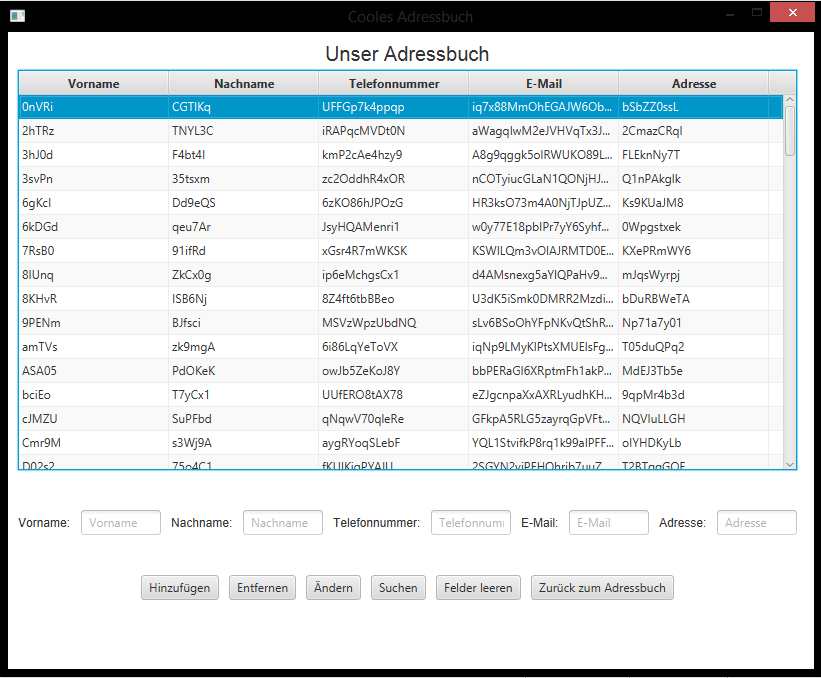
Wir müssen natürlich noch die Signaturen der Methoden im Interface anpassen. Auch dort werfen sie Methoden nun oben besagte Exceptions.

Zur Frage im Aufgabenblatt: Diese Frage stellt sich für uns nicht, da wir mit der generateDetails-Methode eine Möglichkeit gefunden haben, das Problem der zwei Einträge zu umgehen.

Aufgabenteil C:

Wir schreiben eine benutzerfreundliche JavaFX Oberfläche für unser Adressbuch. Wir entschieden uns für eine „Default-Kontaktliste“ (im Screenshot noch ungekürzt), um die Funktionen schneller und leichter testen zu können.

Der Befehl für addDetails läuft über den Hinzufügen-Button. Der Befehl für removeDetails läuft über den Entfernen-Button. Der Befehl für changeDetails läuft über den Ändern-Button. Um die Oberfläche noch benutzerfreundlicher zu machen, haben wir auch noch die Suchen-Funktion integriert. Unser Adressbuch verfügt außerdem über die Möglichkeit, die Felder zu leeren und immer die Möglichkeit zu haben, in die Adressbuch-Ansicht zurück zu gehen (z.B. nachdem ein Kontakt geändert wurde). Hier ein Screenshot:

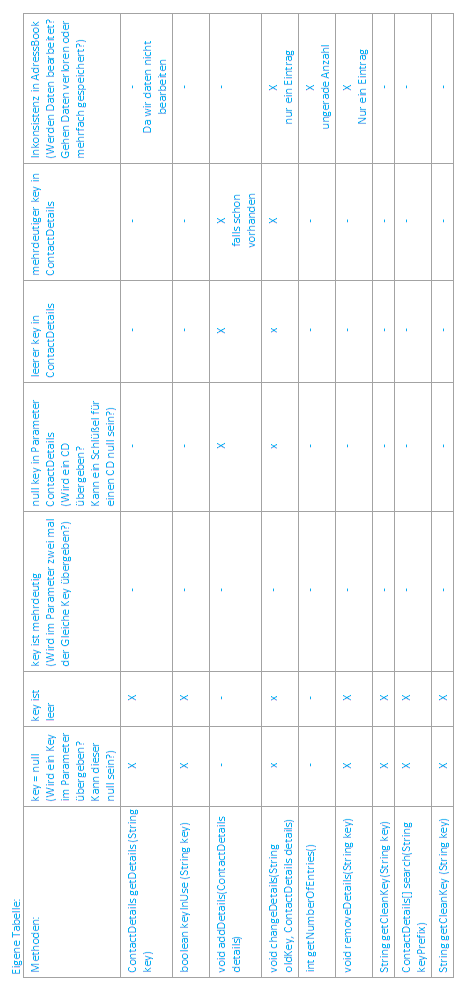


Aufgabenteil D: siehe Aufgabenteil B. Das Problem wird durch die Methode genererateKey() gelöst. Statt 2 keys, nutzen wir ab jetzt nur noch einen key, der aus ALLEN Daten (Vorname, Nachname, Telefonnummer, E-Mail und Adresse) besteht.

1. **Testen**

Aufgabenteil A:

Hier unsere selbst ausgefüllte Tabelle, die alle möglichen Probleme und Fehler erkennt und dann im Programm behandelt.



Aufgabenteil B:

Wir testen unsere FX-Oberfläche und versuchen, alle auftauchenden Fehlerquellen zu beheben.

BUG gefunden: Wir stellen mit erschrecken fest, dass unsere changeDetails-Methode fehlerhaft arbeitet. Beim Ändern eines Kontakts mit unvollständigen Daten wird der Kontakt gelöscht aber nicht mehr hinzugefügt und der Kontakt geht verloren. FIXEN!

Aufgabenteil C:

Damit alles übersichtlich bleibt, lagern wir unsere Main-Methode in die Klasse AddressBookMain aus.

Aus der 'Problem-Tabelle' erstellen wir nun unsere Tests. Hierbei fällt schnell auf, dass jedes Ergebnis umständlich mit if Anweisungen kontrolliert werden muss.

Des Weiteren wird der Code schnell unübersichtlich durch die ganzen try-catch Blöcke. Es ist klar, dass bei größeren Projekten eine Main-Methoden-Kontrolle den Arbeitsaufwand sprengen würde. Der Junit-Test verspricht da Abhilfe.

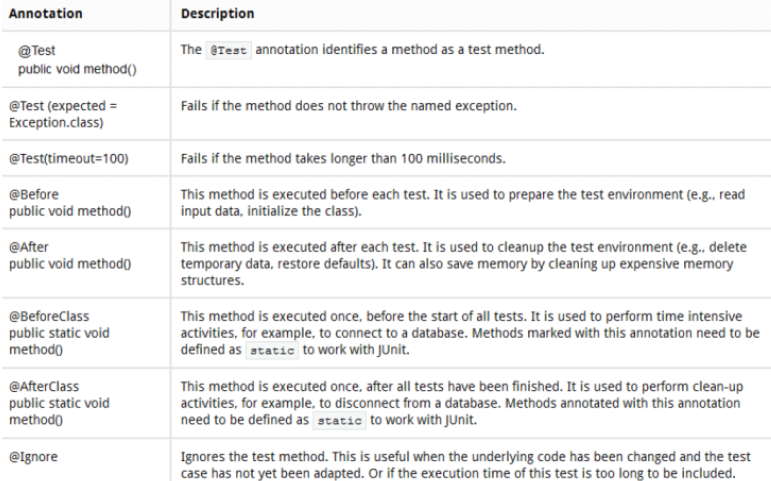
Aufgabenteil D:

Eine JUnit-Klasse ist eine Klasse, die eine andere testet. Die Namenkonvention schreibt als Namen für eine JUnit-TextCase-Klasse vor: KLASSETest und als Paket .tests

Unter dem Reiter New generieren wir in Eclipse eine neue JUnit TestCase. Ein TestCase enthält Methoden mit der Annotation @Test. Es folgen void-Methoden, deren Ergebnis von JUnit überprüft wird.

Außer Methoden zum Testen beinhaltet ein TestCase auch ein Fixture (eine Art Inventar). Darunter fallen Attribute, die für jeden Test bereitstehen sollen. Anweisungen, die vor jedem oder nach jedem Test ausgeführt werden. Aber auch Anweisungen, die vor oder nach jedem TestCase ausgeführt werden sollen. Zu diesem Zweck gibt es noch weitere Annotations.

@Before, @BefordeClass, @After, @AfterClass



Wir schreiben für jede Klasse ein TestCase, um alle Methoden gründlich zu überprüfen.

Wenn mehrere TestCase Klassen existieren, ist es sinnvoll, eine TestSuite zu erstellen. Es handelt sich dabei um eine leere Klasse, die alle übergebenen TestCases ausführt und unser gesamtes Projekt testen kann. Wir übergeben der TestSuite dann noch die zu testenden TestCases:

**package** tests;

**import** org.junit.runner.RunWith;

**import** org.junit.runners.Suite;

**import** org.junit.runners.Suite.SuiteClasses;

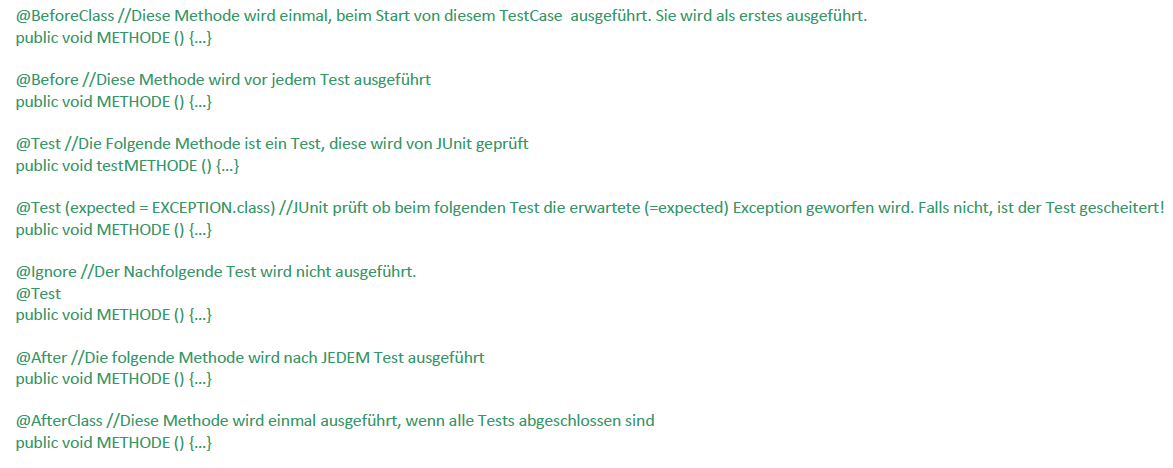
@RunWith(Suite.**class**)

@SuiteClasses({ ContactDetailsTest.**class**,

AddressBookTest.**class**})

**public** **class** AllTests {

}



Wir schreiben die TestCase-Klassen AddressBookTest, die unser AddressBook testet und ContactDetailsTest, die stichprobenartig unsere Klasse ContactDetails testet. ContactDetailsTest erstellen wir nur um der TestSuite zwei TestCases geben zu können (Aufgabe war es, nur das AddressBook zu testen).

Des Weiteren schreiben wir eine J-Unit TestSuite, die alle Tests mit einem Klick ausführt. Unser Ziel ist es, dass alle Balken beim Ausführen grün sind, d.h., dass keine Fehler ausgegeben werden.

Screenshot-Beleg der TestSuite:

