Adressbuch, 3. Ausbaustufe

Verwendet zusätzlich: Interface, Überschreiben von Methoden, Delegation, einfache Reflection, Selbstständige Recherche in der API-Doku.

Nachdem wir bisher einfach ein wenig vor uns hin programmiert haben, ist es langsam an der Zeit, zu überlegen, wie unser Adressbuch eigentlich langfristig aussehen soll. Natürlich soll es eine graphische Oberfläche bekommen. Aber auch eine Bedienbarkeit über ein Web-Interface wäre hübsch. Irgendeine Form der Persistenz ist auf jeden Fall wünschenswert, d.h. die Adressbuchdaten müssen z.B. in einer Datei gespeichert werden. Binär. Oder als Text. Vielleicht als XML? Oder in einer Datenbank?

Das klingt etwas schwierig... will man das alles wirklich jetzt schon entscheiden? Wäre es nicht viel angenehmer, sich diese Entscheidungen offen zu halten, vielleicht sogar so zu programmieren, dass man später einfach zwischen verschiedenen Varianten wechseln kann?

Wir werden deswegen zunächst einmal ein paar grundlegende Eigenschaften der Komponente des Adressbuchs spezifizieren, die letztlich die Person-Exemplare aufnehmen soll, und zwar, ohne Details der Implementierung festzulegen. Es bietet sich an, diese Spezifikation in Form eines Interfaces vorzunehmen. Da AddressBook später die Hauptklasse der kompletten Anwendung sein soll, bekommt das Interface nicht diesen Namen, sondern einen, der seine Funktionalität besser beschreibt. Hier das Interface, diesmal mal mit vollständiger Javadoc-Kommentierung, welche als Teil der Spezifikation zu betrachten ist:

```
package de.mpaap.addressbook;

public interface AddressBookDataStore {

    /**
     * Fuegt ein Person-Exemplar zum Adressbuch hinzu.
     *
     * @param p
     *          das hinzuzufuegende Person-Exemplar
     */
    void addPerson(Person p);

    /**
     * Liefert einen Array, der alle Person-Exemplare im Adressbuch enthaelt, welche den uebergebenen String als
     * Nachnamen tragen. Anwendungsfall koennte z.B. sein, sich alle Paaps aus dem Adressbuch geben zu lassen,
     * um dann einen davon auszuwaehlen und der Methode remove() als Parameter zum Loeschen zu uebergeben.
     *
     * @param name
     *          der Nachname, anhand dessen die Personen gesucht werden sollen
     * @return ein Array mit den Personen, die den uebergebenen Nachnamen tragen
     */
    Person[] getPersons(String name);

    /**
     * Entfernt das uebergebene Person-Exemplar aus dem Adressbuch. Anwendungsfall koennte z.B. sein, sich mit
     * getPersons(String name) alle Paaps aus dem Adressbuch geben zu lassen, um dann einen davon auszuwaehlen
     * und der Methode remove() als Parameter zum Loeschen zu uebergeben.
     *
     * @param p
     *          das zu entfernende Person-Exemplar
     */
    void remove(Person p);

    /**
     * Liefert die Information, ob ein Person-Exemplar im Adressbuch enthalten ist, das den uebergebenen String
     * als Nachnamen traegt.
     *
     * @param name
     *          der Nachname, anhand dessen die Personen gesucht werden sollen
```

```
     * @return true, wenn eine Person mit dem gegebenen Nachnamen im Adressbuch ist, sonst false
     */
    boolean contains(String name);
}
```

Die Frage der Persistenz lassen wir zunächst einmal beiseite, darum werden wir uns in einer späteren Ausbaustufe kümmern. Wie man sieht, gibt die Methode getPerson() einen Array zurück. Der Grund dafür ist, dass es ja durchaus mehrere Personen mit dem genannten Nachnamen geben kann. Außerdem fällt auf, dass im Interface keine Methode genannt ist, die eine String-Repäsentation liefert, also unserer Methode getEntriesAsString() aus Adressbuch_2 entsprechen würde. Der Grund: Eine solche Methode (mit dem Namen toString()) ist in der Klasse Object bereits vorhanden, und damit auch in *jedem* konkreten Exemplar, welcher Klasse auch immer, da jede Klasse diese Methode von Object *erbt*. Es besteht also kein Anlass, sie im Interface explizit zu erwähnen, es würde aber auch nicht schaden, dies zu tun. Da die Information, die die Methode toString(), so wie sie in der Klasse Object implementiert ist, liefert, wenig aussagekräftig ist, wird man sie für eigene Klassen meist auf eine sinnvolle Weise *überschreiben*.

Nachdem nun die Schnittstelle eines AddressBookDataStore festgelegt ist, können wir "gegen diese Schnittstelle programmieren", d.h. wir können mit dem Typ AddressBookDataStore arbeiten und könnten z.B. eine graphische Oberfläche oder einen Test schreiben, ohne zu wissen, wie die konkreten Implementierungen der Schnittstelle aussehen werden. Tatsächlich werden wir spätere *mehrere* Implementierungen haben, und unser Programm wird so aussehen, dass wir erst beim Aufruf des Programms per Übergabeparameter entscheiden müssen, welche konkrete Adressbuch-Klasse wir verwenden wollen.

Aufgaben:

1. Überschreiben Sie zunächst in der Klasse Person die von Object geerbte Methode toString() so, dass sie die Funktionalität der bisherigen Methode getPersonAsString() übernimmt. Die alte Methode entfällt. Zusätzlich bekommt die Klasse Person eine Methode getLastName(), welche den Nachnamen zurückliefert. Zum Testen können Sie versuchen, „ein Person-Objekt auszugeben", indem Sie es an System.out.println() übergeben, *ohne* explizit darauf toString() aufzurufen (das tut println() nämlich intern automatisch).

2. In manchen objektorientierten Sprachen gibt es die Möglichkeit, auch Klassen, Methoden, Konstruktoren usw. ganz natürlich als Objekte anzusprechen, denen man Nachrichten senden kann. In Java existiert diese Möglichkeit ebenfalls, allerdings wurde sie der Sprache nachträglich hinzugefügt (was man auch merkt). Eine ganz einfache Anwendung dieser Verfahren ist die Erzeugung eines Exemplars einer Klasse über den Klassennamen. Diesen Mechanismus braucht man, wenn die Klasse, von der man ein Exemplar benötigt, erst zur Laufzeit feststeht, ein „klassischer" Konstruktoraufruf also nicht in Frage kommt. In dieser Teilaufgabe soll genau das gemacht werden: Wir wollen erst beim Programmaufruf festlegen, welche Implementierung des Interfaces AddressBookDataStore verwendet werden soll.

Ich beschreibe zunächst kurz die Vorgehensweise, und schlage vor, dass Sie einfach einmal versuchen, das mit Hilfe der API-Dokumentation der betreffenden Klassen und Methoden hinzubekommen.

Man benötigt zunächst ein Exemplar der Klasse Class, welches die Klasse repräsentiert, von der man ein Exemplar erzeugen möchte (die Klasse Class selbst ist die Klasse, welche *Klassen* repräsentiert). Dazu bietet die Klasse Class eine statische Methode forName(String className), welcher man den vollständigen Namen der gewünschten Klasse (also mit Packagenamen) als String übergibt. Auf dem Class-Exemplar kann man dann die Methode newInstance() aufrufen und erhält ein Exemplar eben der Klasse, die durch dieses Class-Exemplar repräsentiert wird.

Nun die Aufgabe: Schreiben Sie eine Testklasse, welche geeignete Tests mit einem AddressBookDataStore durchführt. Der Name der konkreten Klasse, die getestet wird, soll dabei *nicht* im Code auftauchen, sondern der main()-Methode der Testklasse als String übergeben werden. In der main()-Methode soll dann

ein Exemplar dieser Klasse erzeugt und mit diesem die Tests durchgeführt werden. Sie können beim Erstellen der Testklasse davon ausgehen, dass konkrete Implementierungen von AddressBookDataStore die von Object geerbte Methode toString() so überschreiben, dass ihre Funktionalität der Methode getEntriesAsString() der Klasse AddressBook aus Adressbuch_2 entspricht. Programmieren Sie auch einen Testfall, bei dem Sie mehrere Person-Exemplare mit gleichem Nachnamen in das AddressBookDataStore füllen und prüfen, ob getPersons() den Array korrekt zurückgibt. Prüfen Sie außerdem, ob remove() korrekt arbeitet, wenn Sie diese Methode mit einer der Personen aus dem von getPersons() gelieferten Array als Parameter aufrufen.

Noch einmal ausdrücklich: Sie sollen wirklich die Testklasse programmieren, ohne überhaupt eine konkrete Klasse zu haben, die das Interface AddressBookDataStore implementiert! Wenn Sie Ihre Testklasse testen wollen, dann können Sie sich zu diesem Zweck eine „Dummy-Implementierung" bauen, d.h. eine Klasse AddressBookDummy implements AddressBookDataStore, welche die Funktionalität einer echten Implementierung simuliert, indem ihre Methoden immer bestimmte Werte zurückliefern.

Achtung: Von den Mechanismen, die Reflection zur Verfügung stellt, werden für die Aufgabe wirklich *nur* die beiden genannten Methoden von Class benötigt! Insbesondere brauchen Sie nach Erzeugung des Adressbuch-Exemplars mit newInstance() *keine* Reflection, um mit dem Adressbuch zu arbeiten! Wenn Sie dabei auf ein Problem stoßen, lösen Sie es bitte mit „konventionellen" Mitteln!

3. Es soll eine Klasse ArrayListAddressBookDataStore erstellt werden, die das Interface AddressBookDataStore implementiert. Diese soll intern ein Exemplar der Klasse ArrayList verwenden, um die Personen abzulegen, d.h. jedes Exemplar der neuen Klasse *kapselt* eine ArrayList und nutzt deren Fähigkeiten, indem sie bestimmte Aufgaben an sie *delegiert*. Zur einfacheren Bearbeitung habe ich die Aufgabe in mehrere Teile untergliedert:

3.1. Erstellen Sie zunächst die Klasse ArrayListAddressBookDataStore mit der internen ArrayList und sämtlichen durch das Interface vorgegebenen Methoden, aber noch ohne deren Implementierung. Wenn Sie sich schon mit parametrischer Polymorphie beschäftigt haben, verwenden Sie bitte eine mit dem Typ Person parametrisierte ArrayList.

3.2. Überschreiben Sie in der Klasse ArrayListAddressBookDataStore die von Object geerbte Methode toString() so, dass sie einen String liefert, der die in Ihrem Adressbuch enthaltenen Personen enthält. Diese Methode ersetzt damit die Methode getEntriesAsString() der Klasse AddressBook aus Adressbuch_2, wobei Füllstand/Kapazität nicht mehr in den String aufgenommen werden, weil die Verdoppelungs-Funktion ja jetzt von der internen ArrayList übernommen wird.

3.3. Schauen Sie sich in der API-Doku die Methoden der Klasse ArrayList an und überlegen Sie, welche Methoden Sie für Ihre Implementierungen der AddressBookDataStore-Methoden verwenden können, indem Sie aus diesen die Methoden der ArrayList aufrufen. Implementieren Sie zunächst die Methode addPerson(Person p), indem sie das übergebene Person-Exemplar an die entsprechende Methode der internen ArrayList „durchreichen".

3.4. Implementieren Sie die Methode getPersons(String name), indem Sie über die ArrayList iterieren und die „passenden" Personen heraussuchen. Verwenden Sie dazu einen Iterator oder die for-each-Form der for-Schleife. Verwenden Sie zur Zwischenlagerung der gefunden Person-Exemplare eine weitere ArrayList. Den als Rückgabetyp der Methode verlangten Person-Array erhalten Sie über die – zugegebenermaßen nicht ganz einfach anzuwendende – Methode toArray(T[] a) der ArrayList, siehe API-Doku.

3.5. Schauen Sie sich in der API-Doku an, wie die Methode contains(Object o) der ArrayList arbeitet. Überlegen Sie, ob Sie diese Methode bei Ihrer Implementierung verwenden können. Welches Problem stellt sich dabei?

3.6. Implementieren Sie nun die Methode contains(String name). Es werden mit einiger Sicherheit interessante Probleme und Fragen auftauchen, die natürlich in der Newsgroup besprochen werden können und sollen.

3.7. Implementieren Sie die Methode remove(Person p). Die Klasse ArrayList bietet *zwei* remove-Methoden. Schauen Sie sich an, wie diese Methoden arbeiten. Können Sie eine davon bei der Implementierung der Methode remove(Person p) verwenden?

4. Testen Sie die in (3) erstellte Klasse mit dem Testprogramm aus (2).

5. Wenn Sie ein SDK installiert haben, dann versuchen Sie doch einmal, das Werkzeug „javadoc" (findet sich im bin-Verzeichnis des SDK) mit der Quelldatei des Interfaces AddressBookDataStore auszuführen und schauen Sie sich die erzeugten HTML-Dateien an.