Dokumentation Programmentwurf
“Studentenverwaltung”

Im Fach

Advanced Software-Engineering

(Java II)

für das

fünfte Semester

des Studiengangs

Informatik

an der Dualen Hochschule Baden-Württemberg Heidenheim

von

Johannes Bendler und Florian Deufel

April 2022

Inhaltsverzeichnis

1. Soll-Konzeption

1.1. Anforderungen aus der Aufgabenstellung

- Die Erweiterbarkeit der Anwendung soll verbessert werden, hierbei soll es in Zukunft möglich sein, das Studenten-Verwaltungssystem auch in andere Länder zu verkaufen (besonders nach Frankreich, Großbritannien und in die USA). Generell soll das Hinzufügen von neuen Ländern möglichst einfach sein.
- Die Anwendung soll über eine vollständige Testabdeckung verfügen, die Unit Tests sollen sowohl in qualitativer als auch quantitativer Form verbessert werden.
- Die Klasse Main soll die Anwendung steuern und Interaktionsmöglichkeiten für den Benutzer bieten.
- Die Klasse Student soll hingegen die Kernfunktionalitäten der Anwendung implementieren.

1.2. Gewünschte Funktionalitäten

- Die Anwendung soll es erlauben, beim Start eine bestimmte Landeseinstellung festzulegen, welche die Formatierung der Datensätze (Telefon und Adressdaten) konsistent beeinflusst. Zur Laufzeit wird hierbei immer genau eine Art der Formatierung unterstützt (keine Misch-Konfigurationen).
- Studenten sollen über ihre Matrikelnummer gefunden werden. Wird bei der Suche die eingegebene Matrikelnummer nicht gefunden, soll eine Fehlermeldung ausgegeben werden. Anschließend soll das erneute Suchen ermöglicht werden.
- Es sollen einfache Informationen (ID, Vorname und Name) zu den Studenten ausgeben werden können.
- Ebenfalls soll es möglich sein, die Adresse und Telefonnummer des Studenten anzuzeigen. Die Darstellung soll hierbei jeweils über zwei Hilfsklassen (Address und PhoneNumber) so formatiert werden, wie es in der jeweiligen Landeseinstellung üblich ist.
- Das Beenden der Anwendung soll über das Menü möglich sein.
- Die Anwendung soll über die Klasse DataStore eine Datenbank simulieren, welche zur Laufzeit die Daten der Studenten aus einer CSV-Datei in die Anwendung einliest.

2. Evaluierung verschiedener Design Patterns

Für die Weiterentwicklung der Studentenverwaltung sollten passende Entwurfsmuster evaluiert werden. Da der Hauptaspekt der Weiterentwicklung das Verbessern der Erweiterbarkeit war wurde der Fokus stark auf die Creational Design Patterns gelegt, diese verbessern die Flexibilität des Programms und erlauben es, bestehenden Code wieder zu verwenden. Darunter fallen folgende Design Patterns:

- Factory Method
- Abstract Factory
- Builder
- Prototype
- Singleton

Das *Factory Method Pattern* und das *Abstract Factory Pattern* sind sich sehr ähnlich, hierbei stellt das *Abstract Factory Pattern* eine Art Weiterentwicklung des *Factory Method Pattern* dar. Das *Abstract Factory Pattern* ist in seiner Implementierung zwar aufwendiger, erlaubt es aber, mehrere Fabriken für die unterschiedlichen, vom Programm unterstützten Sprachen zu erstellen. Damit ist es flexibler als die einfachere *Factory Method*. Die *Abstract Factory* erlaubt es uns spezielle LanguageFactories zu erstellen, welche wiederum sogenannte Produkte erstellen. Sie sind in unserem Beispiel für die verschiedenen Darstellungen der Adressen (z. B. DE_Address) und der Telefonnummern (z. B. DE_PhoneNumber) zuständig. Dadurch, dass wir verschiedene Produktarten benötigen, ist die *Abstract Factory* für unseren Anwendungsfall besser geeignet. In Zukunft können so nicht nur einfacher neue Sprachen (Produktvarianten), sondern auch neue Produkte (z. B. ein Geburtsdatum) hinzugefügt werden.

Ansonsten ist bereits das *Builder Pattern* im gegebenen Programmcode implementiert. Wir haben uns entschieden, dieses Pattern im Programmentwurf zu belassen, da es den Aufbau der verschiedenen Formatstrings erleichtert. Das Builder Pattern wird in den PhoneNumber und Address Klassen verwendet, um die verschieden formatierten Rückgabewerte aus den übergebenen Variablen zusammen zu setzen.

Zuletzt wurde von uns noch das Iterator Pattern genutzt. Hierbei handelt es sich anders als bei den anderen zwei Pattern nicht um ein Creational sondern um ein *Behavioral Pattern*. Es erlaubt es, einfacher eine Sammlung von Elementen schrittweise durchzugehen, ohne dass

die unterliegende Struktur bekannt sein muss. Zum Einsatz kommt es in der Main Klasse, in der setLanguage() Methode. Hier wird es verwendet, um das *hashSet* mit den Landeskürzeln aus den Verzeichnissen zu befüllen. An dieser Stelle wäre auch eine Implementierung ohne das Iterator Pattern möglich gewesen, es erlaubt aber eine kompaktere und übersichtlichere Darstellung.

Ansonsten wurde auf die Verwendung von weiteren Design Patterns verzichtet, da diese für den geforderten Anwendungsfall unserer Meinung keinen Mehrwert bieten. Und den Code unnötig verkomplizieren würden.

3. Ist-Zustand der bereitgestellten Anwendung

Im gegebenen Zustand der Studentenverwaltung sind die gewünschten Kernfunktionalitäten bereits gegeben. Dennoch ist die Ausgabe und Menüführung unübersichtlich. Bei Falscheingaben kann es zu Programmabstürzen kommen, da diese nicht korrekt behandelt werden.

3.1. Gefundene Probleme

- Die Eingabe einer unbekannten Matrikelnummer führt zum Programmabsturz.
- Die Auswahl der Menüpunkte zwei bis fünf führt zum Programmabsturz, wenn zuvor kein Student über die den Menüpunkt eins ausgewählt wurde.
- Die Menüführung ist unübersichtlich, da das Hauptmenü direkt nach Ausgabe in der Konsole ausgegeben wird (teilweise ist dann auch der Cursor falsch platziert).
- Die Nummerierung im Hauptmenü ist nicht fortlaufend.
- Im gegebenen Zustand unterstützt die Anwendung lediglich die in Deutschland übliche Formatierung für Adresse und Telefonnummern. Das Hinzufügen weiterer Sprachen / Formatierungen ist nicht ohne weiteres möglich.
- Die vorhandenen Unit-Tests prüfen nur den Normalfall, die Reaktion des Programms auf falsche Eingaben wie Leerstring etc. wird nicht überprüft. Außerdem werden bei weitem nicht alle Methoden und Möglichkeiten getestet.

4. Problemlösung

4.1. Programmabsturz bei unbekannter Matrikelnummer

Um den Programmabsturz bei einer unbekannten Matrikelnummer zu vermeiden, wurde das Anlegen eines neuen Studenten Objekts innerhalb der selectStudent() Methode von einem Try / Catch Block eingefasst. Schlägt das Anlegen des Studenten fehl, z. B. wenn die eingegebene Matrikelnummer nicht gefunden wurde. Wird die printStudentNotFoundInDataStore() Methode ausgeführt, welche wiederum eine Fehlermeldung in der Konsole ausgibt.

```
    //Erstellen einer LanguageFactory für die zuvor ausgewählte Sprache
    try {
        //Erstellen einer neuen Factory abhängig von der ausgewählten Sprache
        //Erstellen eines neuen Studenten, übergabe der id und factory im Konstruktor
        LanguageFactory factory = getFactoryFromLanguage(lang);
        student = new Student(id, factory);

    }catch(Exception e) {
        //Abfangen falls der Student nicht gefunden wurde
        printStudentNotFoundInDataStore(id);
        waitForUser();
        mainMenue();
    }
}
```

```
//Fehlermeldung: Der Student wurde nicht im DataStore gefunden
private static void printStudentNotFoundInDataStore(String id) {
    clearConsole();
    System.out.println("No Student with the ID: <" + id + "> was found in the DataStore.\n"
                    + "Please consider choosing another ID or contact the support.\n");
}
```

Anschließend wird nach der Bestätigung durch Nutzer das Hauptmenü erneut aufgerufen, um dem Benutzer zu erlauben, eine andere Matrikelnummer anzugeben.

4.2. Programmabsturz bei der Auswahl der Menüpunkte zwei bis fünf

Um Programmabstürze bei nicht ausgewählten Studenten zu vermeiden, wurde eine zusätzliche Abfrage vor den Menüpunkten eingeführt. Diese überprüft, ob bereits ein Student angelegt wurde. Erkennt die Abfrage, dass noch kein Student ausgewählt ist, fordert sie den Nutzer auf, einen Studenten auszuwählen. Somit können die Menüpunkte erst ausgeführt werden, wenn ein Student ausgewählt wurde.

```
//Abfangen der Nutzereingaben / Menüpunkte 2-5, sofern noch kein Student ausgewählt wurde
if(1 < action && action < 6) {
    if(student == null) {
        printNoStudentSelectedError();      //Ausgabe der Fehlermeldung
        selectStudent();                    //Direkter Aufruf zum Auswahl einer Studenten ID
    }
}
```

4.3. Unübersichtliche Menüführung

Zur Verbesserung der Übersichtlichkeit und Menüführung wurde die printMenu() Methode überarbeitet. Es wurde darauf geachtet, einen gewissen Wiedererkennungswert für Eingabemöglichkeiten zu schaffen. In diesem Fall befindet sich in der letzten Zeile eine einfache Printausgabe mit “> ”, welche es dem Nutzer erlaubt, in dieselbe Zeile zu schreiben.

```
//Methode gibt das Hauptmenü aus
private static void printMenu() {
    System.out.println("Please select an option...");
    System.out.println("--------------------------------");
    System.out.println("[1] - Search for student by id");
    System.out.println("[2] - Display info");
    System.out.println("[3] - Display address");
    System.out.println("[4] - Display phone number");
    System.out.println("[5] - Display int'l phone number");
    System.out.println("[6] - Exit program");
    System.out.println("--------------------------------");
    System.out.print("> ");
}
```

Des Weiteren wurden noch die Methoden clearConsole() und waitForUser() eingeführt. Die Methode clearConsole() wird dazu genutzt, die Konsole zu leeren. Sie wird häufig nach Eingaben oder vor neuen Ausgaben ausgeführt.

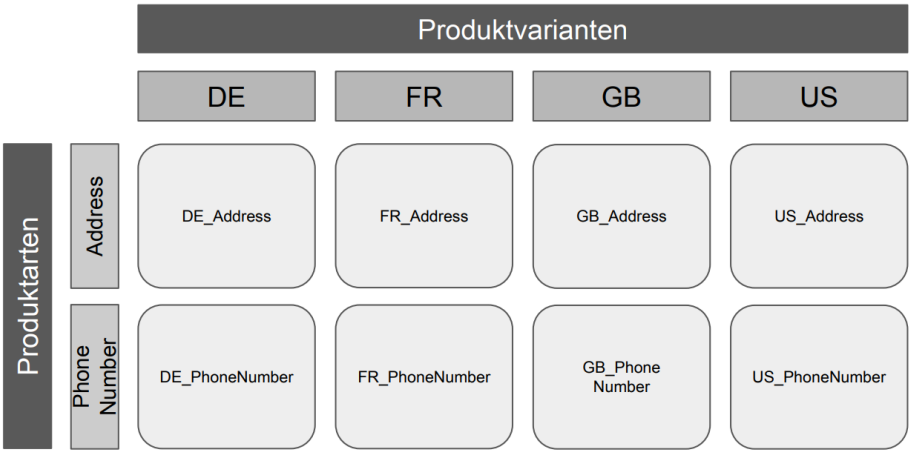
```
//Methode "leert" die Konsole (funktioniert auch in Eclipse)
private static void clearConsole() {
    for(int i = 0; i < 100; i++) {
        System.out.println("\n");
    }
}
```

Die Methode waitForUser() erlaubt es der Anwendung auf die Bestätigung des Nutzers zu warten (z. B. nach einer Ausgabe). Anschließend kann clearConsole() ausgeführt werden und das Hauptmenü erneut aufgerufen werden.

```java
//Methode wartet darauf, dass der Nutzer Enter drückt
private static void waitForUser() throws IOException {
    System.out.println("--------------------------------");
    System.out.println("Press <Enter> to continue");
    cin.readLine();
    clearConsole();
}
```

4.4. Verbesserung der Erweiterbarkeit und Unterstützung für unterschiedliche Formatierungen

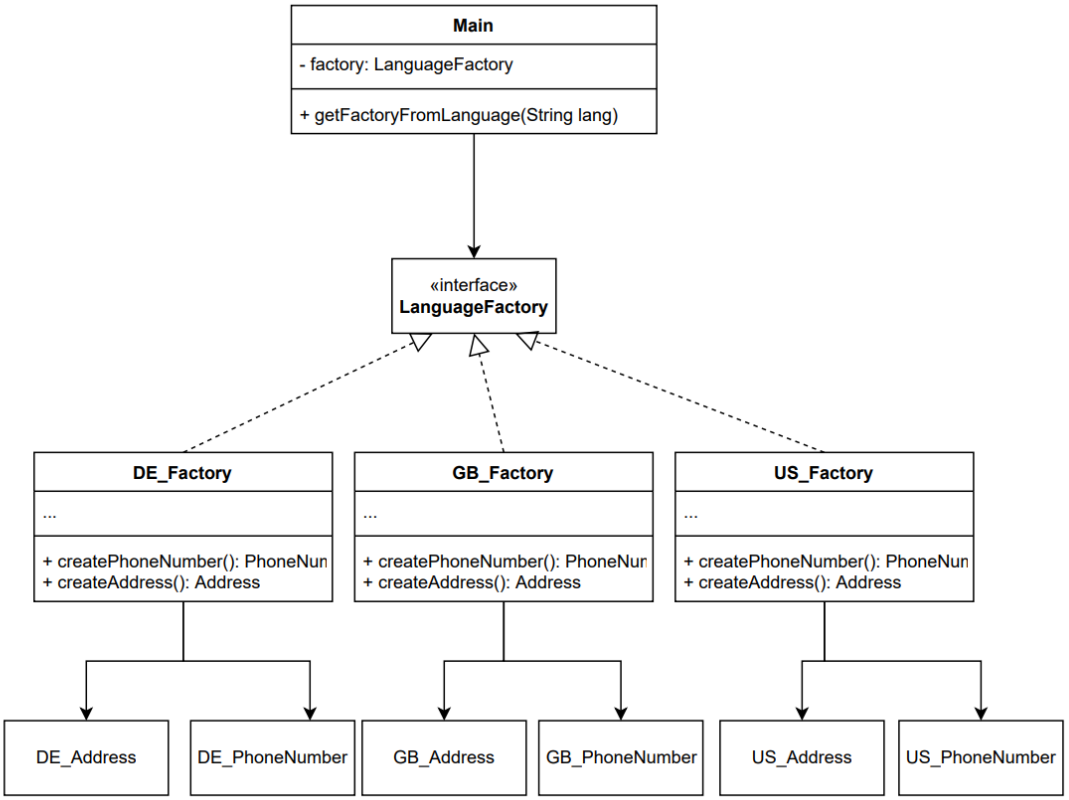
Im gegebenen Zustand unterstützt die Studentenverwaltung lediglich die in Deutschland übliche Darstellung für Adressen und Telefonnummern. Um neue Darstellungen hinzufügen zu können, hätten für jede neue Darstellungsform aufwendige Änderungen am Code vorgenommen werden müssen. Um zukünftige Erweiterungen zu erleichtern, entschieden wir uns, für die Implementierung des *Abstract Factory Patterns*. Dieses erlaubt es uns, den Programmcode flexibler und erweiterbarer zu gestalten und das Hinzufügen von weiteren Sprachen zu erleichtern. Es werden spezifische Factories für die unterstützen Sprachdarstellungen erstellt und die Adressen und Telefonnummern als Produktarten der Factories definiert.



Aufteilung Produktarten und Produktvarianten

Für die jeweiligen Sprachen (DE, FR, GB, US) werden LanguageFactories definiert, die jeweils Adressen und Telefonnummern als Produkte liefern. Die Produkte sind für jede

Sprache ähnlich, müssen aber auf die individuellen Darstellungen angepasst werden. Diese Architektur erlaubt es sowohl, die unterstützen Darstellungen als auch Produktarten zu erweitern. So könnte z. B. auch die individuelle Darstellung eines Geburtsdatums relativ leicht in das Programm aufgenommen werden.



Klassendiagramm für die Abstract Factory (ohne FR)

Auswahl der Darstellungen

Die Auswahl der Sprach-Formatierung erfolgt einmalig nach dem Programmstart. Hierfür wird die Methode setLanguage() ausgeführt. Diese durchsucht das Verzeichnis nach Dateien, die den String “Lang” enthalten, anschließend wird das vorausgehende Länderkürzel in ein *hashSet* eingetragen um Duplikate zu vermeiden. Alle gefundenen Sprachen werden anschließend in der Konsole ausgegeben.

```java
//Methode erlaubt das Setzen der Darstellungsprache,
//hierbei werden dem Nutzer automatisch die unterstützen Spachen ausgegeben
private static void setLanguage() throws IOException {

    //Speichert das Arbeitsverzeichnis der Anwendung in der Variablen directory
    String directory = "\\"+System.getProperty("user.dir")+"\\src\\de\\dhbw\\t2inf3001\\pe";
    LinkedHashSet<String> hashSet = new LinkedHashSet<String>();        //HashSet vermeidet Dopplungen
    Iterator itr;

    File[] files = new File(directory).listFiles();     //Speichern der Dateinamen im Verzeichnis

    //Durchsucht das Verzeichnis nach Unterverzeichnisen mit "Lang"
    //Und speichert das abgeschnittene Sprachkürzel im hashSet (hierdurch ergeben sich die unterstützen Sprachen)
    for (File file : files) {
        if (file.isDirectory() && file.getName().contains("Lang")) {
            hashSet.add(file.getName().split("Lang")[1]);
        }
    }
```

Der Nutzer hat anschließend die Möglichkeit, eines der Länderkürzel auszuwählen, welches dann in der Klassenvariablen lang gespeichert wird.

Das Erstellen der entsprechenden Factory erfolgt bei der Auswahl eines Studenten. Hierbei wird die getFactoryFromLanguage() Methode aufgerufen, welche abhängig vom Länderkürzel ein Objekt der passenden Factory Klasse erstellt.

```java
//Methode erstellt passende Factory abhängig von der Spracheinstellung,
//bei der Erweiterung der unterstützen Darstellungen kann diese Methode leicht angepasst werden
//Gewählter Sprachenkürzel ist Übergabeparameter
private static LanguageFactory getFactoryFromLanguage(String lang) {
    LanguageFactory factory = null;

    switch(lang) {      //Erstellen der passenden Factory abhängig vom Sprachkürzel
    case "DE":
        factory = new DE_Factory();
        break;

    case "FR":
        factory = new FR_Factory();
        break;

    case "GB":
        factory = new GB_Factory();
        break;

    case "US":
        factory = new US_Factory();
        break;
```

Dieses wird von der Methode anschließend zurückgegeben und innerhalb der selectStudent() Methode an den den Konstruktor von Student übergeben.

```java
//Erstellen einer LanguageFactory für die zuvor ausgewählte Sprache
try {
    //Erstellen einer neuen Factory abhängig von der ausgewählten Sprache
    //Erstellen eines neuen Studenten, übergabe der id und factory im Konstruktor
    LanguageFactory factory = getFactoryFromLanguage(lang);
    student = new Student(id, factory);

}catch(Exception e) {
```

Damit arbeitet das Programm in dem definierten Darstellungsmodus.
Die Getter-Methoden in der Studenten-Klasse liefern die Informationen in der gewählten Ländereinstellung.

Schritte zum Hinzufügen einer neuen Darstellung
Durch die Abstract Factory lassen sich neue Darstellungen in fünf einfachen Schritten durchführen:

1. Erstellen neuen Verzeichnisses unter src mit der Bezeichnung Lang+Länderkürzel
2. Kopieren der Java-Files für Adresse, Factory und PhoneNumber von einer bestehenden Darstellung
3. Anpassen der Dateinamen auf neues Länderkürzel
4. Anpassen der Länderkürzel in den Dateien und Anpassen der format() Methode auf die landestypische Darstellung
5. Erweitern des Switch / Case in der getFactoryFromLanguage() Methode unter der Main Klasse um das neue Länderkürzel

4.5. Erweiterte Unit-Tests und bessere Testabdeckung

Die Unit Tests sind ebenfalls zweiteilig aufgebaut. Zum einen gibt es generelle Tests, welche unabhängig von der Wahl der Sprache sind. Dazu zählt vor allem die Klasse *StudentTestGeneral*() in der alle Eigenschaften eines Studenten überprüft werden, die unabhängig von der Sprachwahl sind, wie die Überprüfung des Vor- und Nachnamen, sowie die angezeigte Info, bestehend aus Matrikelnummer und Namen.

Zum Testen der Studenten kann die Datenbank (.csv-Datei) nicht genutzt werden, da ihr Inhalt nicht verlässlich gleichbleibend ist und auch nicht garantiert werden kann, dass überhaupt Einträge vorhanden sind. Zum Ausführen von Unit Tests ist es aber unbedingt notwendig, dass die Daten vorhersagbar und kontrollierbar sind.

Deshalb wurde innerhalb der Studenten-Klasse ein zusätzlicher Konstruktor angelegt, der dem Zweck dienen soll für die Unit Tests Studenten-Objekte zu erstellen. Dafür werden im Konstruktor alle Daten, die eigentlich aus der Datenbank gelesen werden, als String

übergeben. Der Konstruktor generiert dann zur jeweiligen Sprache ein Studenten-Objekt mit dem die Tests ausgeführt werden können.

Für jede Sprache werden sowohl Tests für die Adress-Darstellung sowie die Darstellung aller Sprachspezifischen Eigenschaften ausgeführt:

Bei der Adressierung wird grundlegend nur der String der Formatierung getestet:

- *testFormatBasic()*: Das Standardformat der jeweiligen Sprache
- *testFormatUmlaut()*: Testen, ob das jeweilige Adress-Objekt auch beim Nutzen von Umlauten noch verarbeitet und dargestellt wird. Dieser Test wird nur bei Sprachen mit Umlauten (z.B. Deutsch) ausgeführt.
- *TestFormatItemMissing():* Bei Lesen aus der Datenbanke, bzw. beim Erstellen eines neuen Adress-Objektes ist eines der Argumente ein Leerstring. Geprüft wird, ob die Adresse trotzdem noch fehlerfrei dargestellt wird.
- *TestFormatEmpty()*: Hier wird ein Adress-Objekt nur mit Leerstrings initialisiert und getestet wie die Formatierung darauf reagiert:

```java
package de.dhbw.t2inf3001.pe.TestDE;

import static org.junit.Assert.*;

public class AddressTestDE {
    @Test
    public void testFormatBasic() {
        Address address = new DE_Address("Finkenweg", "1", "Berlin", "12345", "DE");
        String expected = "Finkenweg 1\n12345 Berlin";
        assertEquals(expected, address.format());
    }

    @Test
    public void testFormatUmlaut() {
        Address address = new DE_Address("Königsstraße", "10", "Köln", "50669", "DE");
        String expected = "Königsstraße 10\n50669 Köln";
        assertEquals(expected, address.format());
    }

    @Test
    public void testFormatItemMissing() {
        Address address = new DE_Address("Budapester Straße", "38", "", "35239", "DE");
        String expected = "Budapester Straße 38\n35239 ";
        assertEquals(expected, address.format());
    }

    @Test
    public void testFormatEmpty() {
        Address address = new DE_Address("", "", "", "", "");
        String expected = " \n ";
        assertEquals(expected, address.format());
    }
}
```

Die zweite Sprachspezifische Testdatei umfasst ebenfalls Tests für die Adressierung, diesmal aber als Abfrage eines Studenten-Objekts, sowie die Überprüfung der Darstellung der Telefonnummern. Dazu wird zu Beginn ein Studenten-Objekt erstellt mit Namen, Adresse und Telefonnummer. Bei allen Unit Tests wird auf dieses Studenten-Objekt zugegriffen.

Getestet wird mit den Funktionen:

- *testStudentAdressFormat():* Ein Test der Adressformatierung, diesmal in einem Studenten-Objekt anstelle eines eigenen Adress-Objektes.
- *testStudentPhoneNumber():* Die Abfrage der Standard-Telefonnummer und deren korrekte Darstellung.
- *testStudentInternationalPhoneNumber():* Überprüfung der Darstellung der Telefonnummer im internationalen Format mit Ländervorwahl.

```java
package de.dhbw.t2inf3001.pe.TestDE;

import static org.junit.Assert.*;

public class StudentTestDE {
    private Student createStudent() {
        return new Student("111", "Torsten", "Schmied", "Gneisenaustr. 4 Jasmindorf 41232 DE", "092217 84457", new DE_Factory());
    }

    @Test
    public void testStudentAdressFormat() {
        Student s = createStudent();
        String expected = "Gneisenaustr. 4\n41232 Jasmindorf";
        assertEquals(expected, s.getAddress());
    }

    @Test
    public void testStudentPhoneFormat() {
        Student s = createStudent();
        String expected = "092217-84457";
        assertEquals(expected, s.getPhone());
    }

    @Test
    public void testStudentInternationalPhoneFormat() {
        Student s = createStudent();
        String expected = "+49-92217-84457";
        assertEquals(expected, s.getIntlPhone());
    }
}
```

Es wird empfohlen, für jede neu hinzugefügte Sprache diese spezifischen Tests der Formate anzulegen. Für alle Sprachen bisher (DE, GB, US, FR) sind diese Tests bereits vorhanden. Aufgrund des *Abstract Factory Pattern* sollte aber eigentlich garantiert sein, dass alle *Language Factories* grundlegend funktionsfähig sind. Lediglich die Darstellung müsste überprüft werden um Probleme zu vermeiden.

5. Dateistruktur

Die Studentenverwaltung ist modular aufgebaut, es gibt zwei übergeordnete Verzeichnisse *src* und *test*. Unter dem src-Pfad befinden sich alle für den Betrieb relevanten Programmbestandteile. Der test-Pfad enthält hingegen alle für die Unit-Tests erforderlichen Bestandteile.
Bislang verfügt der Programmentwurf über 28 Klassen, die auf 12 Packages verteilt sind. Die Aufteilung in die Packages erfolgt unter anderem aufgrund der Darstellungssprachen. Und erlaubt es einfacher, neue Sprachen hinzuzufügen. So kann z. B. für eine neue Darstellung einfach ein neues Package angelegt werden. Anschließend können die benötigten drei Java-Dateien kopiert und auf die neue Formatierung angepasst werden.

Programmentwurf						
src						
pe	**exceptions**	**Interfaces**	**LangDE**	**LangFR**	**LangGB**	**LangUS**
• DateStore.java • Main.java • Student.java • datastore.csv	• StudentNotFoundException.java	• Address.java • LanguageFactory.java • PhoneNumber.java	• DE_Address.java • DE_Factory.java • DE_PhoneNumber.java	• FR_Address.java • FR_Factory.java • FR_PhoneNumber.java	• GB_Address.java • GB_Factory.java • GB_PhoneNumber.java	• US_Address.java • US_Factory.java • US_PhoneNumber.java
test						
pe	**TestDE**	**TestFR**	**TestGB**	**TestUS**		
• StudentTestGeneral.java	• AddressTestDE.java • StudentTestDE.java	• AddressTestFR.java • StudentTestFR.java	• AddressTestGB.java • StudentTestGB.java	• AddressTestGB.java • StudentTestGB.java		