

Übungsblatt 04

Vererbung und Polymorphie, Sichtbarkeit, Kapselung, Information hiding


Aufgabe 1 (Sichtbarkeit und Zugriffsschutz)

[3 Punkte]

Beschreiben Sie die Begriffe Sichtbarkeit und Zugriffsschutz in ihren eigenen Worten. Welche Möglichkeiten bietet Java dazu und was sind die Unterschiede zwischen den einzelnen Möglichkeiten?

Abgabe



 at/ac/uibk/pm/gXX/zidUsername/s04/e01/exercise1.txt

Aufgabe 2 (Vererbung und Interfaces)

[18 Punkte]

a) 8 Punkte Im ersten Teil dieser Aufgabe implementieren Sie eine einfache Klasse User wie auch Validatoren, die prüfen, ob ein User gewisse Eigenschaften erfüllt.

- Implementieren Sie eine Klasse User mit folgenden Attributen: username, password, mailAddress und phoneNumber. Als Datentyp können Sie für alle Attribute String verwenden. Implementieren Sie weiters einen Konstruktor mit 4 Parametern, der alle 4 Attribute entsprechend initialisiert, und eine get-Methode für jedes Attribut.
- Implementieren Sie eine Klasse UsernameValidator. Die einzige Aufgabe dieser Klasse ist es zu überprüfen, ob ein Username gültig ist. Implementieren Sie dazu eine Methode `public boolean isValid(User user)`, die prüft, ob der Username mindestens 3 Zeichen lang ist.
- Implementieren Sie eine Klasse namens PasswordValidator, mit der überprüft werden kann, ob das Passwort gültig ist. Implementieren Sie dazu eine Methode `public boolean isValid(User user)`, die prüft, ob das Passwort ungleich dem Usernamen ist.
- Implementieren Sie eine Klasse namens PhoneNumberValidator, mit der überprüft werden kann, ob die Telefonnummer gültig ist. Implementieren Sie dazu eine Methode `public boolean isValid(User user)`, die prüft, ob die Telefonnummer ungleich null ist.
- Implementieren Sie eine Klasse namens MailAddressValidator, mit der überprüft werden kann, ob die Mail-Adresse gültig ist. Implementieren Sie dazu eine Methode `public boolean isValid(User user)`, die prüft, ob die Mail-Adresse ein @-Zeichen enthält.

- Sicher ist Ihnen aufgefallen, dass `UsernameValidator`, `PasswordValidator`, `PhoneNumberValidator` und `MailAddressValidator` sehr ähnlich aussehen. Alle vier Klassen *validieren* ein `User`-Objekt und haben eine Methode `public boolean isValid(User user)`.

Um die Vorteile der objektorientierten Programmierung auszunutzen, bietet es sich an, `Validator` als Abstraktion der vier konkreten Klassen oben einzuführen. Entscheiden Sie sich für eine der folgenden Möglichkeiten:

- `Validator` ist eine konkrete Klasse. `UsernameValidator`, `PasswordValidator`, etc. erben von `Validator`.
- `Validator` ist eine abstrakte Klasse. `UsernameValidator`, `PasswordValidator`, etc. erben von `Validator`.
- `Validator` ist ein Interface. `UsernameValidator`, `PasswordValidator`, etc. implementieren das Interface `Validator`.

Setzen Sie Ihre gewählte Lösung im Code um und begründen Sie Ihre Entscheidung in `exercise2.txt`. Zeichnen Sie außerdem ein UML-Klassendiagramm, das alle Klassen dieser Aufgabe enthält (`exercise2.pdf`).

Abgabe



`at/ac/uibk/pm/gXX/zidUsername/s04/e02/User.java`
`at/ac/uibk/pm/gXX/zidUsername/s04/e02/UsernameValidator.java`
`at/ac/uibk/pm/gXX/zidUsername/s04/e02/PasswordValidator.java`
`at/ac/uibk/pm/gXX/zidUsername/s04/e02/PhoneNumberValidator.java`
`at/ac/uibk/pm/gXX/zidUsername/s04/e02/MailAddressValidator.java`
`at/ac/uibk/pm/gXX/zidUsername/s04/e02/Validator.java`
`at/ac/uibk/pm/gXX/zidUsername/s04/e02/exercise2.txt`
`at/ac/uibk/pm/gXX/zidUsername/s04/e02/exercise2.pdf`

- b) 6 Punkte Im zweiten Teil der Aufgabe geht es darum, *zusammengesetzte* Validatoren zu bauen, die das Ergebnis von zwei Validatoren verknüpfen. Beim Anlegen eines zusammengesetzten Validators werden per Konstruktor zwei Validatoren übergeben und als Objektvariablen gespeichert. Diese beiden Validatoren werden dann nacheinander aufgerufen und deren Ergebnisse verknüpft. Damit kann man beispielsweise prüfen, ob ein User sowohl Validator A als auch Validator B genügt (UND-Verknüpfung).

- Implementieren Sie eine Klasse `AndValidator` mit einem Konstruktor, der zwei Objekte des Typs `Validator` entgegennimmt und diese als Objektvariablen speichert. Wie alle Validatoren ist auch `AndValidator` selbst vom Typ `Validator`. Die Methode `public boolean isValid(User user)` gibt `true` zurück, wenn `isValid` beider Validatoren `true` ist.
- Implementieren Sie eine Klasse `OrValidator` mit einem Konstruktor, der zwei Objekte des Typs `Validator` entgegennimmt und diese als Objektvariablen speichert. Wie alle Validatoren ist auch `OrValidator` selbst vom Typ `Validator`. Die Methode `public boolean isValid(User user)` gibt `true` zurück, wenn `isValid` zumindest für einen der beiden Validatoren `true` ist.

- Im ersten Teil der Aufgabe hatten mehrere Klassen eine Gemeinsamkeit (die Methode `isValid`) und wurden daher zu einem gemeinsamen Typ `Validator` zusammengefasst. Ähnlich verhält es sich mit den beiden Klassen `AndValidator` und `OrValidator`. Beide Klassen sind zusammengesetzte Validatoren und halten intern zwei Validatoren, an die sie eine Aufgabe delegieren. Es bietet sich also an, auch diese beiden Klassen zu einem gemeinsamen Typ `BinaryValidator` zusammenzufassen. Entscheiden Sie sich für eine der folgenden Möglichkeiten:

- `BinaryValidator` ist eine konkrete Klasse. `AndValidator` und `OrValidator` erben von `BinaryValidator`.
- `BinaryValidator` ist eine abstrakte Klasse. `AndValidator` und `OrValidator` erben von `BinaryValidator`.
- `BinaryValidator` ist ein Interface. `AndValidator` und `OrValidator` implementieren das Interface `BinaryValidator`.

Setzen Sie Ihre gewählte Lösung im Code um und begründen Sie ihre Entscheidung in `exercise2.txt`. Ergänzen Sie das UML-Klassendiagramm (`exercise2.pdf`) aus Aufgabe 2a um die neuen Klassen dieser Unteraufgabe.

Hinweis



Online Tools für UML Diagramme:

- <https://app.diagrams.net/>
- <https://apollon.ase.in.tum.de/>

Abgabe



- `at/ac/uibk/pm/gXX/zidUsername/s04/e02/BinaryValidator.java`
- `at/ac/uibk/pm/gXX/zidUsername/s04/e02/AndValidator.java`
- `at/ac/uibk/pm/gXX/zidUsername/s04/e02/OrValidator.java`
- `at/ac/uibk/pm/gXX/zidUsername/s04/e02/exercise2.txt`
- `at/ac/uibk/pm/gXX/zidUsername/s04/e02/exercise2.pdf`

- c) 4 Punkte Verwenden Sie die Klassen von oben, um einen Validator zu bauen, der folgende Bedingungen prüft. Ein gültiger User...

- hat einen gültigen Usernamen **UND**
- hat ein gültiges Passwort **UND**
- hat eine gültige Telefonnummer **ODER** eine gültige Mail-Adresse.

Implementieren Sie ein kleines Testprogramm (`Exercise2Application.java`) und verwenden Sie ihren Validator um 5 verschiedene User zu validieren. Decken Sie dabei verschiedene Fälle ab (gültiger Username, ungültiges Passwort, ...).

Abgabe



- `at/ac/uibk/pm/gXX/zidUsername/s04/e02/Exercise2Application.java`

Aufgabe 3 (Statischer und Dynamischer Typ)


[9 Punkte]

Für diese Aufgabe werden die 3 Klassen A, B und C (A.^{OLAT}java, B.^{OLAT}java, C.^{OLAT}java) verwendet, die in folgender Beziehung zueinander stehen: C erbt von B, und B erbt von A. Sehen Sie sich die Implementierungen dieser Klassen sowie die Klasse Test an (Test.^{OLAT}java) und beantworten Sie folgende Fragen:

- a) 6 Punkte Welche Ausgabe erzeugt die Klasse Test und warum? Begründen Sie, warum was ausgegeben wird.
- b) 3 Punkte Erklären Sie anhand dieses Beispiels den Unterschied zwischen dynamischem und statischem Binden.

Abgabe



 at/ac/uibk/pm/gXX/zidUsername/s04/e03/exercise3.txt

Wichtig: Laden Sie bitte Ihre Lösung in OLAT hoch und geben Sie mittels der Ankreuzliste auch unbedingt an, welche Aufgaben Sie gelöst haben. Die Deadline dafür läuft am Vortag des Proseminars um 16:00 ab.