Programmierung 1

im Wintersemester 2025/26

Praktikumsaufgabe 8

Interfaces und abstrakte Klassen (Kap. 8)

Review: 6. Januar 2026 (DFIW) und 7. Januar 2026 (PI)

In dieser Praktikumsaufgabe arbeiten Sie mit abstrakten Klassen und Interfaces. Der Schwerpunkt liegt auf deren Modellierung in Vererbungshierarchien und der Implementierung solcher Hierarchien.

In Aufgabe 1 entwickeln Sie ein Java-Programm, das eine Hierarchie von abstrakten und konkreten Klassen und Interfaces abbildet.

In Aufgabe 2 verfeinern und erweitern Sie den Umrechner aus der vorherigen Praktikumsaufgabe unter Verwendung eines Interfaces.

Hinweis: Da die Aufgabenstellungen der nahenden Klausur deutlich kompakter sein werden als die der bisherigen Praktikumsaufgaben, möchten wir Sie an dieser Stelle an die zunehmende Selbstständigkeit bei der Lösung dieser anspruchsvolleren Aufgaben heranführen. Die folgenden Aufgaben sind daher weniger detailliert und entsprechen eher dem Stil der Klausur.

Heads up!

Für diese und die folgenden Praktikumsaufgaben gelten weiterhin die Grundsätze der professionellen Softwareentwicklung, die Sie bereits in den vorherigen Praktikumsaufgaben kennengelernt haben. Und nach wie vor gilt: Lösen Sie die Aufgaben nicht allein, sondern mindestens zu zweit! Zum Beispiel mit Ihrer Coding-Partner:in. Denn in der Gruppe lernt es sich besser.

Wir als Ihre Lernbegleiter:innen unterstützen Sie gerne bei der Lösung der Aufgaben.

1 OOP mit abstrakten Klassen und Interfaces

Lernziele

- ☐ Sie modellieren eine Klassenhierarchie mit abstrakten Klassen und Interfaces mit Pseudo-UML.
- ☐ Sie entwickeln abstrakte Klassen mit Attributen und Methoden, die von Unterklassen überschrieben werden.
- ☐ Sie entwickeln konkrete Klassen, die von abstrakten Klassen erben und spezifische Eigenschaften implementieren.
- ☐ Sie definieren Interfaces mit Methoden, die von Klassen implementiert werden.
- ☐ Sie entwickeln eine Testklasse, um die Funktionalität der Klassen und Interfaces zu demonstrieren.
- ☐ Sie reflektieren Ihre Implementierung und diskutieren Konzepte der objektorientierten Programmierung, um Ihr Verständnis zu vertiefen.
- ☐ Sie reflektieren Ihren Lösungsansatz und diskutieren seine Stärken und Schwächen, um Ihren Lernprozess zu optimieren.

Strecklernziele

- ☐ Sie erweitern die Klassenhierarchie um spezifische Eigenschaften und Methoden, um das Verhalten von Objekten realistischer abzubilden und so Ihr Verständnis von Vererbung und Polymorphie zu vertiefen.

In dieser Aufgabe entwickeln Sie ein Java-Programm, das eine Hierarchie von Klassen und Interfaces abbildet. Ziel ist es, Grundkonzepte der objektorientierten Programmierung (OOP) wie Vererbung, Polymorphismus und die Verwendung von Interfaces praktisch anzuwenden. Dazu sollen Methoden sinnvoll überschrieben und spezifische Eigenschaften für Unterklassen implementiert werden.

Hinweis: Ähnlich wie bei der Aufgabe “Vererbung und Polymorphie” in der vorhergehenden Praktikumsaufgabe steht das Design und die Struktur der Klassenhierarchie im Vordergrund und nicht die Funktionalität (siehe Lernziele). Dies ermöglicht den Fokus auf OO-Konzepte.

Heads up!

Modellieren Sie Ihre Klassenhierarchie mit Pseudo-UML und beginnen Sie so früh wie möglich mit der Implementierung der Testklasse, um die Funktionalität Ihrer Implementierung zu überprüfen.

1.1 Abstrakte Superklasse `Animal`

Implementieren Sie eine abstrakte Klasse `Animal` mit den folgenden Eigenschaften und Methoden!

- Ein Attribut `name` (Typ `String`) und die zugehörigen Getter- und Setter-Methoden.
- Eine abstrakte Methode `makeSound()`, die von allen konkreten Unterklassen implementiert werden muss.
- Eine Methode `eat()`, die das Standardverhalten "essen" beschreibt und von Unterklassen erweitert werden kann.

Beispielausgabe:

```
Bambi is eating.
```

1.2 Abstrakte und konkrete Unterklassen von `Animal`

Erstellen Sie zwei abstrakte Klassen `Feline` und `Canine`, die jeweils von `Animal` erben!

Überschreiben Sie in beiden Klassen die Methode `eat()` und erweitern Sie das Verhalten:

`Feline`: Ausgabe von "leckt die Pfoten" nach dem Fressen.

`Canine`: Ausgabe von "heult" nach dem Fressen.

Implementieren Sie die folgenden konkreten Unterklassen:

`Feline`: `Cat` und `Lion`

`Canine`: `Dog` und `Wolf`

Jede konkrete Klasse muss die Methode `makeSound()` mit einem spezifischen Verhalten definieren.

Beispielausgabe:

```
Whiskers meows.
Whiskers is eating.
Whiskers licks its paws after eating.

Buddy barks.
Buddy is eating.
Buddy howls after eating.
```

1.3 Konkrete Klasse `Dragon`

Entwickeln Sie eine konkrete Klasse `Dragon`, die direkt von `Animal` erbt!

Implementieren Sie darin die Methode `makeSound()` mit dem Verhalten "brüllt und spuckt Feuer".

Beispielausgabe:

```
Smaug roars and breathes fire!
```

1.4 Interface `Pet`

Definieren Sie ein Interface `Pet` mit einer Methode `play()`!

Die haustierähnlichen Klassen `Cat` und `Dog` implementieren das Interface mit spezifischen Verhaltensweisen.

Beispielausgabe:

```
Whiskers is playing with a ball of yarn.
```

```
Buddy is fetching the ball.
```

1.5 Abstrakte Superklasse `Robot`

Entwickeln Sie eine abstrakte Klasse `Robot` mit den folgenden Eigenschaften und Methoden!

- Ein Attribut `model` (Typ `String`) und entsprechende Getter- und Setter-Methoden.
- Eine abstrakte Methode `performTask()`, die in den Unterklassen spezifisch implementiert wird.

1.6 Konkrete Klasse `RoboDog`

Implementieren Sie eine Klasse `RoboDog`, die von `Robot` erbt und das Interface `Pet` implementiert!

- Fügen Sie ein Attribut `isCharged` (Typ **`boolean`**) hinzu, das den Ladezustand repräsentiert, und stellen Sie Getter- und Setter-Methoden bereit.
- Die Methode `performTask()` steuert das Verhalten des Roboters in Abhängigkeit vom Ladezustand.
- Die Methode `play()` soll ebenfalls vom Ladezustand abhängen.

Beispielausgabe:

```
R2-D2 is guarding the house.
R2-D2 is playing fetch in simulation mode.

R2-D2 needs recharging before performing tasks.
R2-D2 cannot play while discharged.
```

1.7 Testklasse `AnimalTestDrive`

Vervollständigen Sie die Testklasse `AnimalTestDrive`, die die Funktionalität der Klassen und Interfaces im Vererbungsbaum demonstriert!

Dort werden:

- Instanzen aller Klassen erzeugt.
- Die Attribute über Setter-Methoden initialisiert
- Alle Methoden der Instanzen aufgerufen, um das Verhalten zu testen.

Zusammengefasste Beispielausgabe:

```
Smaug roars and breathes fire!
Smaug is eating.

Whiskers meows.
Whiskers is eating.
Whiskers licks its paws after eating.
Whiskers is playing with a ball of yarn.

Buddy barks.
Buddy is eating.
Buddy howls after eating.
Buddy is fetching the ball.

Simba roars.
Simba is eating.
Simba licks its paws after eating.

Ghost howls.
Ghost is eating.
Ghost howls after eating.

R2-D2 is guarding the house.
R2-D2 is playing fetch in simulation mode.
```

```
R2-D2 needs recharging before performing tasks.
R2-D2 cannot play while discharged.
```

Hinweis: In der Beispielausgabe werden die Namen der in der Testklasse initialisierten Tiere und Roboter verwendet. Wenn Sie andere Namen verwenden, kann Ihre Ausgabe daher abweichen.

1.8 Klassenhierarchie erweitern und verfeinern

In dieser optionalen Vertiefungsaufgabe haben Sie die Möglichkeit, Ihre bestehenden Klassen noch ausdrucksstärker und realitätsnäher zu gestalten. Ziel ist es, spezifische Eigenschaften und Methoden hinzuzufügen, die das Verhalten und die Eigenschaften der Tiere und Objekte weiter verfeinern.

Hinweis: Betrachten Sie die folgenden Anforderungen als Anregung, Ihre Klassen und Methoden nach Belieben zu erweitern und zu verfeinern. Lassen Sie Ihrer Kreativität freien Lauf – schließlich ist Softwareentwicklung eine hochkreative Ingenieursdisziplin!

1.8.1 Erweiterung der Klasse `Animal`

Erweitern Sie die Klasse `Animal` um Attribute und Methoden!

- Ein Attribut `age` (Typ **`int`**) für das Alter des Tieres.
- Ein Attribut `weight` (Typ **`double`**) für das Gewicht des Tieres.
- Eine Methode `move()`, die standardmäßig ausgibt: `Rex moves.`

1.8.2 Verfeinerung der spezifischen Klassen

Verfeinern Sie die spezifischen Klassen mit neuen Eigenschaften und Verhaltensweisen!

`Cat`: Fügen Sie ein Attribut `favoriteToy` (Typ `String`) und eine Methode `hunt()` hinzu, die das Jagdverhalten beschreibt.

`Dog`: Fügen Sie ein Attribut `breed` (Typ `String`) und eine Methode `guard()` hinzu, die das Bewachen des Hauses simuliert.

`Lion`: Fügen Sie ein Attribut `prideSize` (Typ **`int`**) hinzu und verfeinern Sie die Methode `makeSound()`, um ein lautes Brüllen zu simulieren.

`Dragon`: Fügen Sie die Attribute `firePower` (Typ **`int`**) und `wingSpan` (Typ **`double`**) hinzu. Implementieren Sie die Methoden `fly()` und `breatheFire()`, um das Verhalten eines Drachens darzustellen.

`RoboDog`: Fügen Sie ein Attribut `batteryLife` (Typ **`int`**) und die beiden Methoden `recharge()` und `diagnostics()` hinzu, um den Ladezustand zu verwalten.

Diese Aufgabe bietet Ihnen somit die Möglichkeit, Ihr Verständnis von Vererbung, Polymorphismus und Klassenerweiterung zu vertiefen. Außerdem wird Ihre Fähigkeit gefördert, realistische Anforderungen in eine objektorientierte Lösung umzusetzen.

1.8.3 Anpassung der Testklasse `AnimalTestDrive`

Passen Sie die Testklasse `AnimalTestDrive` an, um die neuen Eigenschaften und Methoden zu demonstrieren!

1.9 Reflexion im Coding-Team

Diskutieren Sie im Coding-Team, um Ihr Verständnis zu vertiefen!

1.9.1 Abstrakte Klassen und Interfaces

Wie tragen abstrakte Klassen und Interfaces in Ihrer Lösung zu einer sinnvollen, flexiblen und wiederverwendbaren Klassenhierarchie bei?

- Wie unterscheidet sich die Rolle der abstrakten Klassen von der Rolle der Interfaces in Ihrer Lösung und warum wurden diese Konzepte gerade so kombiniert?
- Wie haben die abstrakten Klassen und Interfaces die Struktur Ihrer Lösung beeinflusst und wo zeigt sich ihr Einfluss in der Implementierung?
- Welche Vorteile bietet die Kombination von abstrakten Klassen und Interfaces für die Flexibilität und Wartbarkeit Ihres Codes? Denken Sie auch an das Prinzip *Easier to Change*!
- Wie könnten Sie ähnliche Konzepte in einem anderen Kontext einsetzen, um komplexe Anforderungen effektiv umzusetzen?

1.9.2 Lernen und Umgang mit neuen Konzepten

Was haben Sie bei der Arbeit an dieser Aufgabe über das Lernen und den Umgang mit neuen Konzepten gelernt?

- *Erfolge erkennen:* Was hat in Ihrem Lernprozess besonders gut funktioniert und wie haben Sie diese Fortschritte erreicht?
- *Schwächen aufdecken:* Bei welchen Konzepten oder Aufgaben hatten Sie Schwierigkeiten und was hat Ihnen gefehlt, um diese Hürden zu überwinden?
- *Lernerfolge sichern:* Wie können Sie die erfolgreich gelernten Inhalte nachhaltig sichern und wiederholen, um optimal vorbereitet zu sein?
- *In die Zukunft blicken:* Welche Strategien oder Methoden könnten Sie in Zukunft anwenden, um ähnliche Herausforderungen noch effektiver zu meistern? Welche kennen Sie aus dem Themenblock “Lernen lernen”? Welche könnten Sie anwenden?

2 Interface statt Superklasse

Strecklernziele

- ☐ Sie führen ein Interface ein, um das Verhalten von Klassen abstrakt zu beschreiben.
- ☐ Sie verwenden ein Interface, um polymorphes Verhalten zu ermöglichen.
- ☐ Sie refaktorisieren bestehenden Code, um die Flexibilität und Erweiterbarkeit zu verbessern.
- ☐ Sie reflektieren praktische Aspekte der Verwendung von Interfaces und abstrakten Klassen in der Softwareentwicklung.

In der letzten Praktikumsaufgabe konnten Sie einen Umrechner implementieren, der mit Hilfe einer Superklasse `ConversionStrategy` eine Basis für die verschiedenen Umrechnungen wie Celsius in Fahrenheit oder Kilometer in Meilen bereitstellt.

Nun knüpfen Sie an die vorherige Aufgabe und deren Lösung an. Ziel ist es, Ihr Verständnis für die Umstrukturierung und Erweiterung bestehender Klassenhierarchien zu vertiefen. Der bewusst reduzierte Detaillierungsgrad dieser Aufgabe fordert Sie auf, eigenständig Lösungswege zu entwickeln und die erlernten Konzepte kreativ anzuwenden. Dies bereitet Sie sowohl auf die Klausur als auch auf praktische Anwendungen in der Softwareentwicklung vor.

In dieser Aufgabe ändern Sie die bestehende Klassenhierarchie so ab, dass die Konvertierungslogik nicht mehr auf einer gemeinsamen Oberklasse basiert, sondern über ein Interface abstrahiert wird. Das Interface ist wie die Superklasse für alle konkreten Umrechnungen (Strategien) gleich.[1] Durch diese Änderung können Sie lernen, wie Interfaces und Polymorphie die Flexibilität und Erweiterbarkeit von Code verbessern können.

Hinweis: Dieser Vorgang wird Refactoring (dt. Refaktorisierung) genannt. Refactoring ist eine wichtige Aktivität bei der Entwicklung qualitativ hochwertiger Softwarelösungen. Dabei wird die interne Struktur des Codes verändert, ohne dass sich das von außen beobachtbare Verhalten der Software ändert. Ziel ist es, den Code besser lesbar, wartbar und erweiterbar zu machen – er wird *easier to change*. Nicht umsonst ist die Refaktorisierung ein zentraler Bestandteil der testgetriebenen Entwicklung (TDD).

[1] Ein solches *Strategie*-Interface definiert typischerweise eine Methode, die ein Kontext zur Ausführung einer Strategie verwendet. In unserem Fall sind die Strategien die Umrechnungen und der Kontext ist die Benutzerschnittstelle, die in der Klasse `TextInterface` implementiert ist. Je nach konkretem Kontext, d. h. je nachdem, welche Umrechnung die Benutzer:in auswählt, wird die entsprechende Strategie und damit die entsprechende Umrechnung ausgeführt.

2.1 Interface `ConversionStrategy` einführen

Erstellen Sie ein Interface `ConversionStrategy`, das die aus der vorherigen Praktikumsaufgabe bekannte Methode `double convert(double value)` deklariert!

Ersetzen Sie die gleichnamige Superklasse aus der vorherigen Implementierung durch dieses Interface.

Hinweis: Um Namenskonflikte zu vermeiden, löschen Sie die alte Klasse oder benennen Sie sie um, bevor Sie das neue Interface mit demselben Namen erstellen.

2.2 Anpassung der Strategien

Passen Sie alle existierenden konkreten Umrechnungsstrategien so an, dass sie das Interface `ConversionStrategy` implementieren!

Dies betrifft also zum Beispiel konkrete Klassen wie `CelsiusToFahrenheitStrategy` und `KilometerToMileStrategy`.

Stellen Sie sicher, dass die Methode `convert()` unverändert bleibt.

2.3 Integration in die Benutzerschnittstelle

Aktualisieren Sie die Klasse `TextInterface` so, dass die verfügbaren Konvertierungsstrategien weiterhin verwendet werden können!

Da sich die Signatur der Methode `convert()` nicht ändert, muss der Aufruf nicht angepasst werden.

Hinweis: Für die erweiterte Menüführung (Anzeige der Beschreibungen) sind im nächsten Schritt jedoch Anpassungen erforderlich.

2.4 Auswahl der Umrechnungsstrategie durch die Benutzer:in

Erweitern Sie Ihre Benutzerschnittstelle um die polymorphe Verwendung des Interface, so dass die Benutzer:in die Umrechnung frei wählen kann!

Heads up!

Fügen Sie dem Interface eine Methode `String getDescription()` hinzu, die eine kurze Beschreibung der Umrechnung zurückgibt. Sie können diese Beschreibung dann in der Benutzerschnittstelle anzeigen, um der Benutzer:in die Auswahl zu erleichtern. Speichern Sie weiterhin alle verfügbaren Umrechnungen (Strategien) in einer `List` und geben Sie diese Liste aus. Anhand der Eingabe einer passenden Zahl durch die Benutzer:in kann dann die entsprechende Umrechnungsstrategie aus der Liste ausgewählt und ausgeführt werden.

Ausschnitt aus der angepassten Klasse `TextInterface`:

```
private List<ConversionStrategy> strategies =
    List.of(
        new CelsiusToFahrenheitStrategy(),
        new FahrenheitToCelsiusStrategy(),
        new KilometerToMileStrategy(),
        new MileToKilometerStrategy());
```

Auszug aus den über die Benutzeroberfläche angebotenen Umrechnungen:

```
===== Hauptmenü =====
1) Umrechnung
2) Beenden
Bitte geben Sie entweder 1 oder 2 ein: 1
===== Umrechner =====
Verfügbare Umrechnungen:
1) Celsius zu Fahrenheit
2) Fahrenheit zu Celsius
3) Kilometer zu Meilen
4) Meilen zu Kilometer
Bitte wählen Sie eine Option:
```

2.5 Testen der Implementierung

Überprüfen Sie Ihre Implementierung, indem Sie eine Testklasse schreiben, die das neue Design verwendet!

Stellen Sie sicher, dass die Umrechnungen für alle Strategien korrekt durchgeführt werden.

Hinweis: Da Sie nur die interne Struktur des Umrechners geändert haben, sollten alle vorhandenen Tests weiterhin erfolgreich sein. Glückwunsch, Sie konnten Ihre Tests dann ohne weitere Änderungen oder Erweiterungen als Sicherheitsnetz (wieder-)verwenden, um effektiv und effizient sicherzustellen, dass Ihre Änderungen keine unerwünschten Nebeneffekte haben! Und ein Hoch auf das Prinzip *Easier to change*! Sie hatten keine Tests, die Sie hätten ausführen können? Kein Problem, dann ist jetzt ein guter Zeitpunkt, um von Anfang an Tests zu schreiben – und noch besser, wenn Sie ab sofort Test Driven Development (TDD) anwenden.

2.6 Weitere Umrechnungsstrategien

In dieser optionalen Aufgabe können Sie weitere Umrechnungsstrategien hinzufügen!

Erweitern Sie den Umrechner z. B. um die Umrechnung von Celsius in Kelvin. Implementieren Sie die Strategien und integrieren Sie sie in die Benutzerschnittstelle.

2.7 Reflexion im Coding-Team

Diskutieren Sie im Coding-Team, um Ihr Verständnis zu vertiefen!

2.7.1 Flexibilität und lose Kopplung

Warum fördern Interfaces Flexibilität und lose Kopplung?

- Welche Vorteile bieten Interfaces gegenüber abstrakten Klassen, insbesondere im Hinblick auf Flexibilität und lose Kopplung?
- Wie helfen Interfaces, Implementierungen voneinander zu entkoppeln und Abhängigkeiten zu minimieren?
- Warum ist lose Kopplung ein wichtiges Prinzip in der Softwareentwicklung und wie unterstützt sie die Erweiterbarkeit von Code?

2.7.2 Modularität und Testbarkeit

Wie unterstützt die Verwendung von Interfaces die Modularität und Testbarkeit von Software?

- Was bedeutet Modularität im Softwaredesign und warum ist sie wichtig?
- Wie ermöglichen Interfaces die Isolation einzelner Komponenten, z. B. für Unit-Tests oder die Entwicklung neuer Features?
- Wie können Interfaces helfen, Änderungen in einem Teil des Codes durchzuführen, ohne andere Teile zu beeinflussen?

2.7.3 Wartbarkeit und langfristige Qualität

Wie beeinflusst die Verwendung von Interfaces die Wartbarkeit von Software?

- Warum ist es wichtig, Code so zu schreiben, dass zukünftige Änderungen oder Erweiterungen leicht implementiert werden können?
- Wie können Interfaces helfen, den Code sauber und lesbar zu halten?
- Welche Herausforderungen können entstehen, wenn auf lose Kopplung und klare Trennung von Verträgen und Implementierungen verzichtet wird?

2.7.4 Wiederverwendbarkeit und Erweiterbarkeit

Wie fördern Interfaces die Wiederverwendbarkeit und Erweiterbarkeit von Code?

- Wie erleichtern Interfaces die Einführung neuer Anforderungen, z. B. durch das Hinzufügen neuer Umrechnungsstrategien?
- Wie können Sie sicherstellen, dass Ihre Wahl zwischen Interface und abstrakter Klasse zukünftigen Anforderungen gerecht wird?
- Welche praktischen Einschränkungen oder Herausforderungen könnten sich ergeben, wenn nur Interfaces verwendet werden?