Programmierung 1

im Wintersemester 2024/25

Praktikumsaufgabe 7

Vererbung und Polymorphie

Review: 18. Dezember 2024

In dieser Praktikumsaufgabe erfahren Sie, wie Vererbung und Polymorphie Ihr Leben als Programmierer:in erleichtern, wenn es darum geht, dass Sie oder andere Ihren Code einfach erweitern können. Dazu definieren Sie gemeinsame Funktionalitäten zentral in einer Superklasse und verwenden diese in den Unterklassen wieder (*Vererbung*). Zusätzlich individualisieren Sie die gemeinsamen Funktionalitäten in den Unterklassen (*Polymorphie*).

In Aufgabe 1 beginnen Sie mit einer zunächst naiven Implementierung, die Sie schrittweise um Vererbung und Polymorphie erweitern und damit professionalisieren.

In Aufgabe 2 können Sie ein Konvertierungswerkzeug für verschiedene Einheiten mit textbasierter Benutzeroberfläche entwickeln und dabei Vererbung und Polymorphie direkt anwenden.

Hinweis: Die nächste Praktikumsaufgabe baut direkt auf dieser Praktikumsaufgabe auf, Sie werden den Java-Code dieser Aufgabe weiterentwickeln. Daher ist es wichtig, dass Sie die Aufgabenstellung und die Lösung dieser Aufgaben verstehen und umsetzen können.

Heads up!

Für diese und die folgenden Praktikumsaufgaben gilt:

Konventionen und Praktiken: Halten Sie sich an die Konventionen der professionellen Softwareentwicklung und folgen Sie den bewährten Verfahren. Zum Beispiel: Wählen Sie

aussagekräftige Namen. Strukturieren Sie Ihren Code durch Einrückungen, damit er übersichtlich und leicht nachvollziehbar ist. Verwenden Sie Pseudo-UML, um Ihre Klassen zu entwerfen. Verwenden Sie private Attribute, die Sie mit Settern und Gettern zugänglich machen.

Testgetriebene Entwicklung: Gehen Sie iterativ und inkrementell vor! Erstellen Sie zunächst eine Liste von Testszenarien für die geplanten Funktionalitäten. Implementieren Sie ein Testszenario aus der Liste als fehlschlagenden Test (*Red*). Entwickeln Sie dann nur so viel Code, wie nötig ist, um den Test zu bestehen (*Green*). Optimieren Sie schließlich Ihren Code (*Refactor*). Führen Sie dabei nach Änderungen alle Tests erneut aus. Wiederholen Sie diesen Zyklus für alle Testszenarien. Diese Vorgehensweise ermöglicht eine kontrollierte und schrittweise Entwicklung.

Trennung von Produktions- und Testcode: Schreiben Sie den *Testcode*, der den *Produktionscode* mit der eigentlichen Programmlogik ausführt und testet, in einer separaten Klasse "`TestDrive`".

Kontinuierliches Lernen: Nutzen Sie diese Aufgaben, um das Ergebnis Ihrer Reflexion des Feedbacks aus den vorherigen Aufgaben umzusetzen. So lernen Sie kontinuierlich und können Ihre Fähigkeiten als professionelle Softwareentwickler:in ausbauen.

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 Vererbung und Polymorphie

Lernziele

- ☐ Sie verwenden Vererbung, um Redundanz zu reduzieren und die Wartbarkeit zu erhöhen.
- ☐ Sie verwenden Polymorphie, um die Flexibilität und Erweiterbarkeit zu verbessern.
- ☐ Sie diskutieren die Vorteile von Vererbung und Polymorphie.

Ein Shopsystem muss verschiedene Zahlungsmethoden verarbeiten können, darunter Kreditkarten und PayPal. Es ist absehbar, dass in Zukunft weitere Zahlungsmethoden wie Google Pay und Apple Pay hinzukommen werden. Perspektivisch sollen neben der Bezahlung weitere Funktionen wie Rückerstattungen und Gebührenberechnungen implementiert werden. Vage angedacht ist auch die Möglichkeit, mehrere Zahlungsmethoden in einer Bestellung zu kombinieren oder Teilzahlungen zu ermöglichen.

Stellen Sie sich vor, Sie sind eine Entwickler:in, die diese Funktionalität implementiert. Als professionelle Entwickler:in kennen Sie das *Easier to Change*-Prinzip und wissen, dass auch deswegen *hohe Kohäsion* und *lose Kopplung* wichtig sind.

Im ersten Schritt erstellen Sie zunächst eine naive Implementierung, die ohne Vererbung und Polymorphie auskommt. Anschließend erweitern Sie die Implementierung schrittweise um Vererbung und Polymorphie, um von deren Vorteilen zu profitieren.

Hinweis: Da in dieser Aufgabe das Design und die Struktur der Klassenhierarchie und nicht die Funktionalität im Vordergrund steht (siehe Lernziele), wird die Zahlung nur simuliert, indem eine Ausgabe auf der Konsole erzeugt wird. In einer realen Implementierung würden die Zahlungen über die APIs der jeweiligen Zahlungsdienstleister abgewickelt. Die Anwendungslogik wird daher sehr schlank sein, so dass hier ausnahmsweise auf eine testgetriebene Entwicklung verzichtet werden kann. Wenn Sie aufgefordert werden, die von Ihnen implementierten Zahlungsmethoden zu verwenden, verwenden Sie den bewährten `TestDrive`-Ansatz mit einer `main`-Methode, um den gesamten Code mit allen Klassen zu kompilieren und das Zusammenspiel der einzelnen Klassen und Methoden anhand der Ausgaben auf der Konsole nachzuvollziehen.

Heads up!

Lösen Sie jede der drei Teilaufgaben 1.1 bis 1.3 in einem eigenen Java-Projekt, entweder als

separate Projekte im OneCompiler oder mit `javac` und `java` in verschiedenen Verzeichnissen auf Ihrem lokalen Rechner. So können Sie die Implementierungen jederzeit vergleichen und die Unterschiede zwischen den verschiedenen Ansätzen leichter nachvollziehen.

Wie bei der testgetriebenen Entwicklung gehen Sie *iterativ* vor und bauen *inkrementell* auf früheren Implementierungen auf, um Ihre Implementierung schrittweise zu verbessern: Naive Implementierung → Einführung von Vererbung → Einführung von Polymorphie

Tipp: Kopieren Sie den Code nach den Teilaufgaben 1.1 und 1.2 jeweils in ein neues Projekt, damit Sie nicht jedes Mal von vorne anfangen müssen. So können Sie den Code der vorherigen Teilaufgabe als Ausgangspunkt für die nächste Teilaufgabe verwenden.

1.1 Naive Implementierung

Implementieren Sie jede Zahlungsmethode separat, ohne Vererbung oder Polymorphie zu verwenden!

1.1.1 Die Klassen `CreditCard` und `PayPal`

Entwickeln Sie im Folgenden die beiden Klassen `CreditCard` und `PayPal`.

Die Klasse `CreditCard` soll die folgenden Attribute und Methoden enthalten:

`cardHolder` Der Name der Karteninhaber:in als `String`.

`payWithCreditCard(double amount)` Bezahlt den übergebenen Betrag mit der Kreditkarte. Hier wird nur eine Ausgabe auf der Konsole simuliert.

Für die Karteninhaberin Jana würde z. B. folgende Ausgabe auf der Konsole erscheinen: `Jana paid 100.00 EUR by credit card.`

Die Klasse `PayPal` soll analog die folgenden Attribute und Methoden enthalten:

`accountHolder` Der Name der PayPal-Kontoinhaber:in als `String`.

`payWithPayPal(double amount)` Führt die Zahlung des übergebenen Betrags mit PayPal aus. Hier wird nur eine Ausgabe auf der Konsole simuliert.

Für den PayPal-Nutzer Florian würde z. B. folgende Ausgabe auf der Konsole erscheinen: `Florian paid 100.00 EUR using PayPal.`

1.1.2 UML-Klassendiagramm der naiven Implementierung

Erstellen Sie ein UML-Diagramm der beiden Klassen `CreditCard` und `PayPal`!

Verwenden Sie *Pseudo-UML*, wie es im Buch Java von Kopf bis Fuß dargestellt ist.

1.1.3 Java-Code der naiven Implementierung

Implementieren Sie die beiden Klassen `CreditCard` und `PayPal` in Java!

1.1.4 Verwendung der naiven Implementierung

Verwenden Sie die beiden Klassen wie folgt!

- Erstellen Sie zwei Instanzen `creditCard` und `payPal` der beiden Klassen.
- Führen Sie jeweils eine Zahlung mit einem Betrag von 100,00 EUR aus. Rufen Sie dazu die entsprechende Methode direkt auf den beiden Instanzen `creditCard` und `payPal` auf.
- *Gedankenexperiment:* Wie könnten Sie eine Schleife implementieren, die die beiden Zahlungsmethoden in einer `ArrayList` mit dem Namen `paymentMethods` speichert und dann für jede Zahlungsmethode die Zahlung ausführt, indem sie über die Objekte in `paymentMethods` iteriert?

Hinweis: Falls Sie die Schleife implementieren möchten, benötigen Sie eine `ArrayList` vom Typ `Object`, den Operator **`instanceof`** und einen Cast.

1.1.5 Redundanz und fehlende Einheitlichkeit

Diskutieren Sie die folgenden Schwächen der naiven Implementierung im Coding-Team!

Redundanz: Die Klassen `CreditCard` und `PayPal` enthalten sehr ähnliche Attribute und Methoden. Dies führt zu Redundanz und erhöht den Wartungsaufwand, insbesondere wenn neue Zahlungsmethoden wie `GooglePay` hinzugefügt werden.

Fehlende Einheitlichkeit: Es gibt keine gemeinsamen Methoden, um auf die Inhaber:innen der Zahlungsmethoden zuzugreifen oder die Zahlung durchzuführen. Dies erschwert die Verwendung der beiden Klassen in einer Schleife erheblich.

Umständliche Verarbeitung: Die Iteration über verschiedene Zahlungsmethoden in einer Schleife wäre aufwändig und würde die Verwendung fortgeschrittener Konzepte wie **`instanceof`** und Cast erfordern. Diese Konzepte machen den Code schwer lesbar und schwer wartbar, insbesondere wenn die Anzahl der verschiedenen Zahlungsmethoden steigt, z. B. über Google Pay, Apple Pay, Klarna, etc.

1.2 Einführung von Vererbung

Reduzieren Sie die Redundanz und fehlende Einheitlichkeit der naiven Implementierung durch die Einführung einer Superklasse!

1.2.1 Die Superklasse `PaymentMethod`

Die Superklasse soll die redundanten Attribute `cardHolder` und `accountHolder` durch das gemeinsame Attribut `holder` ersetzen.

Erstellen Sie im Folgenden eine Superklasse namens `PaymentMethod` mit dem Attribut `holder` vom Typ `String`. Lassen Sie dann die beiden Klassen `CreditCard` und `PayPal` über **`extends`** von `PaymentMethod` erben. Entfernen Sie schließlich die beiden redundanten Attribute.

Hinweis: Die beiden Methoden in den beiden Unterklassen zur Ausführung der Zahlung bleiben zunächst unverändert. Verwenden Sie den Getter aus der Superklasse, um auf das Attribut `holder` für die dortige Ausgabe zuzugreifen.

1.2.2 UML-Klassendiagramm mit Vererbung

Erstellen Sie ein UML-Diagramm der drei Klassen `PaymentMethod`, `CreditCard` und `PayPal`!

Verwenden Sie wieder Pseudo-UML.

1.2.3 Java-Code mit Vererbung

Implementieren Sie die drei Klassen mit Hilfe von Vererbung in Java!

1.2.4 Verwendung der Implementierung mit Vererbung

Erweitern Sie die Verwendung der beiden Instanzen `creditCard` und `payPal` um den einheitlichen Zugriff auf die Inhaber:in mit Hilfe des Getters der Superklasse!

Zeigen Sie zuerst die jeweilige Inhaber:in in der Konsole an und führen Sie dann die Zahlung aus.

1.2.5 Neue Zahlungsmethode hinzufügen

Fügen Sie eine neue Zahlungsmethode `GooglePay` hinzu!

Diese soll die Methode `payWithGooglePay(double amount)` enthalten, die den übergebenen Betrag mit Google Pay bezahlt. Auch hier wird nur eine Ausgabe auf der Konsole simuliert.

Passen Sie Ihr UML-Diagramm und Ihren Java-Code entsprechend an und verwenden Sie die neue Zahlungsmethode in Ihrem Code.

1.2.6 Verbesserte Struktur durch Vererbung

Diskutieren Sie im Coding-Team die Vorteile, die sich aus der Einführung der Vererbung ergeben!

Einheitliche Struktur: Die Superklasse `PaymentMethod` enthält das Attribut `holder` und den zugehörigen Getter. Dies ermöglicht einen einheitlichen Zugriff auf die Inhaber:in der Zahlungsmethode in den Unterklassen `CreditCard`, `PayPal` und `GooglePay` und reduziert die Redundanz im Code und erhöht die Lesbarkeit durch eine einheitliche Struktur.

Erweiterbarkeit: Neue Zahlungsmethoden wie `GooglePay` können einfach hinzugefügt werden, indem sie von der Superklasse `PaymentMethod` erben. Dies erleichtert die Wartung und Erweiterung des Codes.

1.3 Einführung von Polymorphie

Erhöhen Sie die Flexibilität und Erweiterbarkeit der Implementierung durch die Einführung von Polymorphie!

1.3.1 Die polymorphe Methode `processPayment()`

Führen Sie im Folgenden die abstraktere Methode `processPayment()` in der Superklasse ein, die die drei spezifischeren Methoden `payWithCreditCard()`, `payWithPayPal()` und `payWithGooglePay()` ersetzt.

Hinweis: Diese neue Methode kann eine allgemeine Ausgabe wie die folgende erzeugen: `Jana paid 100.00 EUR.` Im folgenden *Kapitel Interfaces und abstrakte Klassen* werden wir sehen, wie das Schlüsselwort **`abstract`** verwendet wird, um wirklich abstrakte Klassen und Methoden in Java zu definieren.

1.3.2 UML-Klassendiagramm mit Vererbung und Polymorphie

Erstellen Sie ein UML-Diagramm der vier modifizierten Klassen `CreditCard`, `PayPal`, `GooglePay` und `PaymentMethod`!

Verwenden Sie wieder Pseudo-UML.

1.3.3 Java-Code mit Vererbung und Polymorphie

Implementieren Sie die vier Klassen mit Hilfe von Vererbung in Java!

1.3.4 Verwendung der Implementierung mit Vererbung und Polymorphie

Iterieren Sie über Instanzen der drei Zahlungsmethoden und führen Sie die jeweilige Zahlung durch einen polymorphen Methodenaufruf aus!

Erstellen Sie eine geeignete `ArrayList`, fügen Sie die Instanzen der drei Zahlungsmethoden hinzu und iterieren Sie über die Liste, um die Zahlung auszuführen.

1.3.5 Neue polymorphe Methode hinzufügen

Fügen Sie in der Superklasse `PaymentMethod` und in den Unterklassen eine neue Methode `refundPayment()` ein, die den übergebenen Betrag zurückerstattet!

Die entsprechenden Methoden in den verschiedenen Klassen sollen eine zugehörige Ausgabe wie die folgende erzeugen:

- `Jana received a refund of 50.00 EUR.`
- `Jana received a refund of 50.00 EUR by credit card.`
- `Jana received a refund of 50.00 EUR via PayPal.`
- `Jana received a refund of 50.00 EUR via Google Pay.`

Passen Sie Ihr UML-Diagramm und Ihren Java-Code entsprechend an und verwenden Sie die neue Methode in Ihrem Code.

1.3.6 Flexibilität und Erweiterbarkeit durch Polymorphie

Diskutieren Sie im Coding-Team die Vorteile der Einführung von Polymorphie!

Einheitliche Verarbeitung: Alle Zahlungsmethoden können über einheitliche Methoden wie `processPayment()` und `refundPayment()` verarbeitet werden. Die konkreten Zahlungsmethoden kümmern sich um die spezifischen Details der Zahlung und Rückerstattung. Dies vereinfacht die Verarbeitung beliebiger Zahlungsmethoden und erhöht die Flexibilität des Zahlungssystems.

Erweiterbarkeit ohne Änderungen: Neue Zahlungsmethoden (z. B. Apple Pay) können hinzugefügt werden, ohne dass der bestehende Code angepasst werden muss. Dies ermöglicht eine einfache Erweiterung des Zahlungssystems.

1.4 Reflexion im Coding-Team

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

1.4.1 Refactoring-Prozess

Wann ist eine iterative und inkrementelle Einführung von Vererbung und Polymorphie sinnvoll?

- In dieser Aufgabe wurden Vererbung und Polymorphie schrittweise eingeführt. Dies war insbesondere aufgrund des didaktischen Aufbaus der Aufgabe sinnvoll, um die Vorteile von Vererbung und Polymorphie schrittweise zu verdeutlichen. Wann würden Sie in einem realen Projekt eine ähnliche Vorgehensweise wählen?

- Bei welcher Vorgehensweise, die Sie bereits kennen, würden Sie nach und nach eine Superklasse, dann Unterklassen und schließlich abstraktere Methoden einführen?

- An welchem Punkt würden Sie in einem realen Projekt mit der Einführung von Vererbung und Polymorphie beginnen?

- Wann würden Sie Vererbung und Polymorphie nicht einsetzen?

1.4.2 Redundanz, Einheitlichkeit, Flexibilität und Erweiterbarkeit

Was ist unter Redundanz, Einheitlichkeit, Flexibilität und Erweiterbarkeit zu verstehen?

- Wie erkennen Sie als Entwickler:in, ob ein Code redundant ist? Wie reduzieren Sie Redundanzen?

- Woran erkennen Sie mangelnde Konsistenz im Code? Wie bekommen Sie als Entwickler:in eine solche zu spüren? Wie stellen Sie Konsistenz her?

- Wann ist eine Implementierung flexibel? Wie erreichen Sie Flexibilität in der Implementierung?

- Wie erkennen Sie eine erweiterbare Implementierung? Wie ermöglichen Sie die Erweiterbarkeit einer Implementierung?

1.4.3 Polymorphie ohne Vererbung

Kann es Polymorphie ohne Vererbung geben?

- Wie hängen Vererbung und Polymorphie zusammen?
- Können Sie sich konzeptionell vorstellen, wie Polymorphie ohne Vererbung in Java umgesetzt werden könnte?
- Wie unterstützt Vererbung die Umsetzung von Polymorphie?
- Was haben Polymorphie durch Vererbung und Methodenüberladung gemeinsam?

1.4.4 Vorteile von Polymorphie

Was sind die Vorteile der Polymorphie?

Hier ist ein Beispielcode, der ohne Polymorphie auskommt, um über mehrere Zahlungsmethoden zu iterieren, um die konkreten Zahlungsmethoden auszuführen.:

```
1  import java.util.ArrayList;
2
3  public class NoPolymorphismTestDrive {
4    public static void main(String[] args) {
5      // Erstellen von Instanzen der Zahlungsmethoden
6      CreditCard creditCard = new CreditCard();
7      PayPal payPal = new PayPal();
8      GooglePay googlePay = new GooglePay();
9
10     // Setzen der Eigenschaften der konkreten Zahlungsmethoden
11     payPal.setHolder("Alice");
12     creditCard.setHolder("Bob");
13     googlePay.setHolder("Charlie");
14
15     // Hinzufügen der konkreten Zahlungsmethoden zu einer ArrayList
16     ArrayList<Object> paymentMethods = new ArrayList<>();
17     paymentMethods.add(creditCard);
18     paymentMethods.add(payPal);
```

```
19      paymentMethods.add(googlePay);
20
21      // Iterieren über die ArrayList und konkrete Zahlungen ausführen
22      double amount = 100.0;
23      for (Object paymentMethod : paymentMethods) {
24        if (paymentMethod instanceof CreditCard) {
25          CreditCard cc = (CreditCard) paymentMethod;
26          cc.payWithCreditCard(amount);
27        } else if (paymentMethod instanceof PayPal) {
28          PayPal pp = (PayPal) paymentMethod;
29          pp.payWithPayPal(amount);
30        } else if (paymentMethod instanceof GooglePay) {
31          GooglePay gp = (GooglePay) paymentMethod;
32          gp.payWithGooglePay(amount);
33        } else {
34          System.out.println("Unknown payment method: " + paymentMethod);
35        }
36      }
37    }
38  }
```

- Wie beurteilen Sie die Lesbarkeit im Vergleich zur Lösung mit der polymorphen Methode `processPayment()`?
- Was sind die Vorteile der Polymorphie in diesem Fall?
- Was ist der Unterschied bezüglich der Erweiterbarkeit um neue Zahlungsmethoden? Was muss geändert werden, wenn eine neue Zahlungsmethode hinzugefügt wird?
- Warum ist der Ansatz ohne Polymorphie fehleranfällig?
- Welcher Zusammenhang besteht zwischen Polymorphie und dem Prinzip *Easier to Change* sowie zwischen *hoher Kohäsion* und *loser Kopplung*?

2 Umrechner mit textbasierter Benutzerschnittstelle

Strecklernziele

- ☐ Sie entwerfen und implementieren eine Klassenhierarchie für eine Familie von austauschbaren Algorithmen.
- ☐ Sie implementieren eine textbasierte Benutzerschnittstelle für die Anwendung der Algorithmen.
- ☐ Sie diskutieren, wie das *Easier to Change*-Prinzip durch die Klassenhierarchie unterstützt wird.
- ☐ Sie diskutieren, wie eine textbasierte Benutzerschnittstelle einen schnellen Nutzen und frühes Feedback ermöglicht.

Aus der Vorlesung kennen Sie den Umrechner, der Temperaturangaben in Celsius und Fahrenheit umrechnet und diese Funktionalität über eine textbasierte Benutzerschnittstelle (CLI) anbietet.

In dieser Aufgabe implementieren Sie den Umrechner selbst. Dabei verwenden Sie Vererbung und Polymorphie, um die interne Struktur des Codes, d.h. welche Klassen wofür existieren und wie diese zusammenarbeiten, leicht erweiterbar und gleichzeitig sehr flexibel zu gestalten.

Heads up!

Softwaredesign: Achten Sie auf die Einhaltung des *Easier to Change*-Prinzips und auf eine *hohe Kohäsion* und *lose Kopplung* der Klassen. Ein geeignetes Klassendesign kennen Sie aus der Vorlesung.

Kommandozeile: Verwenden Sie für Ihre Lösung anstelle des OneCompilers einen Texteditor und die Kommandozeile, um Ihre Java-Dateien zu kompilieren und auszuführen. Schließlich wollen Sie die Funktionalität einer *textbasierten* Benutzerschnittstelle testen.

2.1 Klassenhierarchie

Erstellen Sie im Folgenden eine Klassenhierarchie verschiedener Umrechner, die jeweils zwischen zwei Maßeinheiten umrechnen!

2.1.1 Superklasse `ConversionStrategy`

Entwickeln Sie die Superklasse `ConversionStrategy`, die die Umrechnung eines Wertes von einer Einheit in eine andere durchführt!

Zu diesem Zweck besitzt die Klasse die einzige Methode `convert(float value)`, die den übergebenen Wert umwandelt und das Ergebnis zurückgibt. Sie stellt also eine allgemeine Umrechnungsstrategie dar, die durch Unterklassen konkretisiert wird.

Erstellen Sie zunächst ein UML-Diagramm der Klasse, bevor Sie sie in Java implementieren!

Hinweis: Da wir das Schlüsselwort **`abstract`**[1] noch nicht behandelt haben, können Sie die Methode `convert()` als identische Abbildung implementieren, die einfach den Eingabewert `value` zurückgibt. In den Unterklassen wird die Methode dann überschrieben, um die konkrete Umrechnung durchzuführen.

Sie fragen sich, warum die Klasse `ConversionStrategy` heißt? Spoiler-Alarm: In dieser Aufgabe und in den folgenden Praktikumsaufgaben werden wir uns Schritt für Schritt dem Entwurfsmuster *Strategie*[2] nähern.

2.1.2 Konkrete Umrechnungen in Unterklassen

Entwickeln Sie Unterklassen für die Umrechnung zwischen Grad Celsius und Fahrenheit sowie zwischen Kilometern und Meilen!

Benennen Sie diese Klassen wie folgt und führen Sie die entsprechenden Umrechnungen durch:

`CelsiusToFahrenheitStrategy` Grad Celsius → Grad Fahrenheit.

[1] Mit **`abstract`** kann eine Methode sagen: "Ich bin nur eine Vorlage. Meine genaue Funktionsweise wird später festgelegt, aber ich muss in den Unterklassen mit Leben gefüllt werden!"

[2] Es ist eines von vielen Entwurfsmustern, die Ihnen helfen, Ihren Code zu strukturieren und zu organisieren. Genauer gesagt gehört es zu den Verhaltensmustern (engl. behavioral design pattern) und ermöglicht den Austausch von Algorithmen zur Laufzeit. Ideal für eine Anwendung, bei der die Benutzer:innen später zwischen verschiedenen Umrechnungen wählen können sollen, oder!?

`FahrenheitToCelsiusStrategy` Grad Fahrenheit → Grad Celsius.

`KilometerToMileStrategy` Kilometer → Meilen.

`MileToKilometerStrategy` Meilen → Kilometer.

Erweitern Sie zuerst das UML-Diagramm um diese Klassen, bevor Sie sie in Java implementieren!

Hinweis: Vergessen Sie nicht, TDD zu verwenden, um die korrekte Funktionalität der Klassen sicherzustellen!

2.2 Benutzerschnittstelle

Binden Sie zumindest die Umrechnung für eine Maßeinheit in eine Richtung in eine textbasierte Benutzerschnittstelle ein, die den Wert abfragt, die Umrechnung durchführt und das Ergebnis ausgibt!

Verwenden Sie den Code aus der Vorlesung und ersetzen Sie den dortigen Code für die Umrechnung durch Ihre Klassen aus dieser Aufgabe.

Tipp: Im einfachsten Fall genügt es, die Klasse `TemperatureConverter` zu entfernen und stattdessen eine eigene Umrechnungsstrategie in die Klasse `TextInterface` zu integrieren. Soll die Funktionalität erhalten bleiben, müssen zwei Klassen integriert werden, eine für jede Richtung der Umrechnung.

Wenn Sie möchten, können Sie bereits jetzt die Funktionalität so erweitern, dass über die Benutzerschnittstelle zwischen den verschiedenen Umrechnungen gewählt werden kann. Die nächste Praktikumsaufgabe wird diese Erweiterung explizit und detailliert behandeln.

2.3 Reflexion im Coding-Team

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

2.3.1 Flexible Algorithmen durch Vererbung und Polymorphie

Wie trägt das *Easier to Change*-Prinzip dazu bei, dass die Algorithmen flexibel austauschbar und erweiterbar sind?

- Wie unterstützt die Klassenhierarchie das *Easier to Change*-Prinzip? Wie hängen die Konzepte der Vererbung und der Polymorphie mit dem Prinzip zusammen? Wie tragen diese beiden Konzepte zur Kohäsion und Kopplung der Klassen bei?

- Wie können konkret die Algorithmen der Klassenhierarchie im Code flexibel ausgetauscht und erweitert werden? Wie kann z. B. eine neue Umrechnung von Grad Celsius in Kelvin hinzugefügt werden?

2.3.2 Textbasierte Benutzerschnittstelle für schnellen Nutzen

Welche Vorteile bietet die Implementierung einer textbasierten Benutzerschnittstelle gegenüber einer ausgefeilteren Benutzerschnittstelle?

- Welche Vorteile bietet eine textbasierte Benutzerschnittstelle gegenüber einer anspruchsvolleren Benutzerschnittstelle wie z.B. einer grafischen Benutzerschnittstelle unter Windows oder macOS?

- Wie wirkt sich die Implementierung einer textbasierten Benutzerschnittstelle auf die schnelle Nutzbarkeit der Anwendung aus?

- Wie unterstützt eine textbasierte Benutzerschnittstelle schnelles Prototyping und das frühe Einholen von Feedback? Welche Vorteile ergeben sich aus dem frühzeitigen Testen einer einfachen textbasierten Benutzerschnittstelle?

- Inwiefern kann eine textbasierte Benutzerschnittstelle als Basis für spätere Erweiterungen dienen, z.B. eine grafische Benutzerschnittstelle oder eine Webanwendung? Welche Designentscheidungen im Code könnten diese Erweiterungen erleichtern?

2.3.3 Identische Abbildung als Vorlage für konkrete Umrechnungen

Warum ist die identische Abbildung in der Superklasse `ConversionStrategy` eine sinnvolle Vorlage für die Unterklassen?

- Warum ist es sinnvoll, die Methode `convert()` in der Superklasse als identische Abbildung zu implementieren? Welche Alternativen gibt es für die konkrete Implementierung?
- Welche Vorteile hat es, wenn die Methode in der Superklasse vorhanden ist und in den Unterklassen überschrieben wird? Was wäre, wenn sie dort nicht vorhanden wäre?
- Welche Nachteile sehen Sie in der Notwendigkeit, die Methode als identische Abbildung zu implementieren, obwohl sie in den Unterklassen überschrieben wird? Welche Vorteile hätte es, wenn man festlegen könnte, dass die Methode in den Unterklassen implementiert werden muss, ohne dass eine konkrete Implementierung in der Oberklasse erforderlich ist?