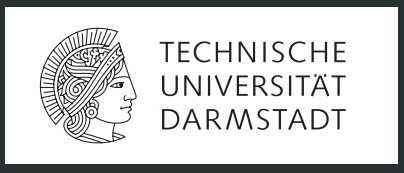
Funktionale und objektorientierte Programmierkonzepte
Übungsblatt 05



Entwurf

Achtung: Dieses Dokument ist ein Entwurf und ist noch nicht zur Bearbeitung/Abgabe freigegeben. Es kann zu Änderungen kommen, die für die Abgabe relevant sind. Es ist möglich, dass sich **alle** Aufgaben noch grundlegend ändern. Es gibt keine Garantie, dass die Aufgaben auch in der endgültigen Version überhaupt noch vorkommen und es wird keine Rücksicht auf bereits abgegebene Lösungen genommen, die nicht die Vorgaben der endgültigen Version erfüllen.

Hausübung 05

Gesamt: 37 Punkte

Klassen und Interfaces

Beachten Sie die Seite *Verbindliche Anforderungen für alle Abgaben* im Moodle-Kurs.

Verstöße gegen verbindliche Anforderungen führen zu Punktabzügen und können die korrekte Bewertung Ihrer Abgabe beeinflussen. Sofern vorhanden, müssen die in der Vorlage mit `TODO` markierten `crash`-Aufrufe entfernt werden. Andernfalls wird die jeweilige Aufgabe nicht bewertet.

Die für diese Hausübung relevanten Verzeichnisse sind `src/main/java/h05` und ggf. `src/test/java/h05`.

Unbewertete Verständnisfrage:

In dieser Hausübungen verwenden wir die Begriffe *Objektattribute* und *Klassenattribute*, vgl. Kapitel 03b, Folien 168-184. Machen Sie sich den Unterschied zwischen diesen beiden Begriffen unbedingt klar!

Einleitung

Die bisherigen Übungen basierten auf `FopBot`. In dieser Übung lösen Sie sich nun von `FopBot` und erstellen alle Dateien des Quellcodes selber (also nur die Dateien im `/src`-Ordner). Ziel dieser Hausübung ist es, Typen von Fortbewegungsmitteln und deren Zusammenhänge zu modellieren. Allerdings wird das Modell weder umfassend noch wirklich realitätsnah sein, da dies den Rahmen dieser Hausübung sprengen würde.

Aufgaben

H1: Drei Interfaces — 8 Punkte

H1.1: Interface `ElectricallyDriven` — 2 Punkte

Schreiben Sie in einer Datei `ElectricallyDriven.java` ein `public`-Interface `ElectricallyDriven` mit

- den booleschen, parameterlosen Methoden `standardVoltageChargeable` und `highVoltageChargeable` sowie mit
- der rückgabelosen Methode `letsGo`, die einen Parameter `additionalChargeVolume` vom formalen Typ `byte` und einen Parameter `distance` vom formalen Typ `int` hat.

Unbewertete Verständnisfrage:

Warum hat `ElectricallyDriven` nicht einfach nur eine einzige boolesche Methode zur Abfrage, ob ein Elektromobil mit normaler Spannung oder mit Hochspannung geladen werden kann, sondern gleich zwei Methoden dafür?[a]

[a] Falls Ihnen diese Frage zu leicht vorkommt: Designfehler dieser Art passieren durchaus häufig in der Praxis. Merkt man manchmal erst sehr viel später, nämlich wenn der erste Fall auftritt, der nicht in das ursprünglich gewählte Design passt. Aber dann ist es vielleicht zu spät, weil zu viel Code darauf aufbaut und nachträglich geändert werden müsste.

H1.2: Interface `FuelDriven` — 3 Punkte

Schreiben Sie in einer Datei `FuelType.java` eine Enumeration `FuelType` mit den Konstanten `GASOLINE`, `DIESEL` und `LPG`.

Schreiben Sie in einer Datei `FuelDriven.java` ein `public`-Interface `FuelDriven` mit

- der parameterlosen Methode `getFuelType` mit Rückgabetyp `FuelType` und
- der Methode `getAverageConsumption`, die einen Parameter `speed` vom Typ `double` und eine Rückgabe vom Typ `double` hat.

H1.3: Interface `HybridVehicle` — 3 Punkte

Schreiben Sie in einer Datei `DriveType.java` eine `public`-Enumeration `DriveType` mit zwei Konstanten: `FUEL_BASED` und `ELECTRICAL`.

Schreiben Sie in einer Datei `HybridVehicle.java` nun ein Interface `HybridVehicle`, welches

- die Interfaces `FuelDriven` und `ElectricallyDriven` erweitert und zudem
- eine parameterlose Methode `getPreferredDriveType` mit Rückgabetyp `DriveType` hat.

H2: Klasse `MeansOfTransport` — 7 Punkte

Schreiben Sie in einer Datei `TransportType.java` eine Enumeration `TransportType` mit den Konstanten `BICYCLE`, `CAR`, `VESSEL` und `AIRCRAFT`.

Schreiben Sie in einer Datei `MeansOfTransport.java` eine abstrakte `public`-Klasse `MeansOfTransport` mit

- einem `protected`-Attribut `transportType` vom statischen Typ `TransportType`,
- einer parameterlosen `public`-Objektmethode `getTransportType`, die einfach den momentanen Wert von Attribut `transportType` zurückliefert und

- einer abstrakten `public`-Objektmethode `letMeMove`, die einen Parameter `distance` vom Typ `int` und Rückgabetyp `int` hat.

Einen Konstruktor hat die Klasse `MeansOfTransport` nicht.

Überschreiben Sie außerdem in der Klasse `MeansOfTransport` die Methode `toString` von Klasse `Object` (vgl. Kapitel 03b, Folien 194-207), und zwar so, dass sie den folgenden String zurückgibt (ohne Anführungszeichen):

`"I am a <transport type>."` bzw. `"I am an <transport type>."`

Der Substring `<transport type>` steht oben nur als Platzhalter und wird in der tatsächlichen Ausgabe mit dem Namen der Fortbewegungsmittelart in Attribut `transportType` ersetzt – und zwar so, dass der erste Buchstabe des Namens groß und der Rest des Strings klein geschrieben wird (siehe Hinweise unten). Falls `transportType` `null` ist, so wird `<transport type>` mit dem String `"undefined"` (wieder ohne Anführungszeichen) ersetzt.

Verbindliche Anforderungen:

- Sollte `transportType` später einmal um weitere Fortbewegungsmittelarten erweitert werden, soll Ihre Implementation der Methode `toString` auch mit diesen unvorhersehbaren Konstanten korrekt arbeiten, ohne dass `toString` dafür geändert werden müsste. Dabei dürfen Sie nicht davon ausgehen, dass die Namen weiterer Konstanten irgendwelche Konventionen erfüllen. Falls das erste Zeichen ein Buchstabe (groß oder klein) ist, wird es zu einem Großbuchstaben, alle weiteren Buchstaben werden zu Kleinbuchstaben. Dabei soll der unbestimmte Artikel `a` durch `an` ersetzt werden, falls das erste Zeichen im Namen ein Vokal ist.[a] Jedes Sonderzeichen (also jedes Zeichen, das nicht Klein- oder Großbuchstabe oder Ziffer ist), wird zu einem Leerzeichen. Desweiteren können Sie hierbei davon ausgehen, dass das erste Zeichen niemals ein Sonderzeichen ist.
- Zur Klassifikation und zur Umwandlung von Zeichen verwenden Sie ausschließlich die Möglichkeiten in Kapitel 01b, Folie 180 sowie Folien 228-232, insbesondere keine Funktionalität aus der Java-Standardbibliothek oder anderswoher.

[a] Diese Verwendung von `a/an` ist natürlich nicht hundertprozentig korrekt.

***Hinweise*:**

- Von Klasse `java.lang.Enum`, erbt jede Enumeration eine parameterlose Methode `name`, die den Namen der Konstante als `String` zurückliefert.
- Sie können Zeichen arithmetisch auf Klein- oder Großbuchstabe oder auch auf Ziffer testen. Zum Beispiel können Sie für ein Zeichen testen, ob es eine Ziffer ist, indem Sie testen, ob das Zeichen sowohl `>='0'` als auch `<='9'` ist (Hochkommas nicht vergessen, `'0'` ist ungleich `0` usw.!)
- Fügen Sie eine Konstante wie `TH7S_is_A_tEsT` in `MeansOfTransportType` ein, um zu überprüfen, ob Ihre Implementation von `toString` die verbindliche Anforderung erfüllt. Es ist anzuraten, dass Sie diese Konstante nach dieser Überprüfung nicht ganz entfernen, sondern für etwaige spätere nochmalige Überprüfung nur auskommentieren.

Unbewertete Verständnisfrage:

- ***Fehlermeldungen besser verstehen*:** Wie auf Übungsblatt 3 versuchen Sie in einer separaten Datei ein Objekt der Klasse `MeansOfTransport` mit `new` einzurichten. Passt das zu Ihrem Verständnis von Kapitel 03b, Folien 82-89?
- Warum implementieren wir nicht einfach `letMeMove` in `MeansOfTransport`, sondern machen `MeansOfTransport` stattdessen abstrakt?
- Warum ist `MeansOfTransport` eine abstrakte Klasse und kein Interface?

H3: Abgeleitete / Implementierende Klassen 22 Punkte

H3.1: `FuelDriven` durch `FuelDrivenVehicle` implementieren 6 Punkte

Schreiben Sie in einer Datei `FuelDrivenVehicle.java` eine nicht-abstrakte `public`-Klasse `FuelDrivenVehicle`, die die Klasse `MeansOfTransport` erweitert und das Interface `FuelDriven` implementiert.[1]

Für die Methode `getFuelType` hat `FuelDrivenVehicle` ein `private`-Attribut `fuelType` vom Typ `FuelType`. Die Methode `getFuelType` soll den aktualen Wert dieses Attributs zurückliefern.

Die Methode `getAverageConsumption` liefert `0` zurück, falls der aktuale Wert des Parameters `speed` negativ ist, `20`, falls er größer als `200` ist, und `0.1*speed` bei Werten im Intervall `[0...200]`.

Klasse `FuelDrivenVehicle` hat ein `private`-Attribut `fillingLevel` vom Typ `int` und eine zugehörige `public get`-Methode mit dem üblichen Namen `getFillingLevel`. Zudem hat sie eine rückgabelose Methode `fillUp` mit einem Parameter `fillValue` vom formalen Typ `int`, die `fillingLevel` um den aktualen Wert dieses Parameters hochsetzt, falls letzterer Wert positiv ist (andernfalls hat `fillUp` keinen Effekt).

Falls der aktuale Wert des Parameters `distance` negativ ist, hat die Methode `letMeMove` keinen Effekt außer der Rückgabe (die Rückgabe wird am Ende dieses Absatzes spezifiziert). Falls `distance` nicht negativ, aber kleiner als `10*fillingLevel` ist, verringert `letMeMove` den Wert von `fillingLevel` um `distance/10` (ganzzahlige Division), andernfalls setzt `letMeMove` das Attribut `fillLevel` auf 0. In jedem dieser Fälle wird das Produkt aus dem Wert, um den `fillingLevel` reduziert wurde, und `10` zurückgeliefert.[2]

Schreiben Sie in `FuelDrivenVehicle` einen `public`-Konstruktor mit einem Parameter `fuelType` vom formalen Typ `FuelType`, einem Parameter `transportType` vom formalen Typ `TransportType` sowie einem Parameter `fillingLevel` vom Typ `int`. Mit den aktualen Werten dieser drei Parameter werden die gleichnamigen Attribute initialisiert.

***Hinweis*:** Beachten Sie, dass die Ihnen bisher bekannte Möglichkeit, mit `super(...)` die Attribute der Basisklasse zu initialisieren, hier nicht geht, da die Basisklasse keinen entsprechenden Konstruktor hat. Aber das betroffene Attribut ist dieses Mal `protected`, damit geht es auch.

[1] In einer realen Vererbungshierarchie zu diesem Thema würde man wahrscheinlich zwischen `MeansOfTransport` und `FuelDrivenVehicle` noch eine Klasse `Car` einfügen, die für Autos mit Verbrennermotor und Elektromotor gleichermaßen schon die gemeinsame Funktionalität definiert, zum Beispiel alles zu Rädern und zur Karosserie. Aber hier geht es nur ums Prinzip, dafür reichen einige wenige Klassen und Interfaces.

[2] Sie können die Rückgabe also interpretieren als die Distanz, um die das Fahrzeug tatsächlich bewegt wurde, unter der Annahme, dass `10` der korrekte Umrechnungsfaktor ist.

Unbewertete Verständnisfragen:

***Fehlermeldungen besser verstehen*:**

- Kommentieren Sie den Methodenrumpf von `letMeMove` in `FuelDrivenVehicle` einmal aus (also Semikolon anstelle des Methodenrumpfes). Macht die Fehlermeldung für Sie Sinn?
- Schreiben Sie nun `abstract` vor die Methode, ändern aber den Kopf der Klasse noch nicht. Macht die daraus resultierende Fehlermeldung für Sie Sinn?
- Schreiben Sie `abstract` nun auch vor den Kopf der Klasse und versuchen analog zu früheren Hausübungen, in einer separaten Datei ein Objekt von `FuelDrivenVehicle` zu erzeugen. Dann sollte `FuelDrivenVehicle.java` fehlerfrei kompilieren, diese separate Datei aber nicht.
- Wenn Sie die Methode `letMeMove` wieder in die ursprüngliche, nichtabstrakte Form bringen, aber die Klasse `FuelDrivenVehicle` immer noch abstrakt lassen, sollte `FuelDrivenVehicle.java` weiterhin fehlerfrei kompilieren. Das heißt, eine Klasse kann abstrakt sein, obwohl sie keine abstrakten Methoden hat. Können Sie sich einen Sinn dahinter vorstellen?[a]

Vergessen Sie nicht, alle Fehler wieder rückgängig zu machen.

[a] Im Kapitel zu GUIs werden wir ein sinnvolles Beispiel sehen: Adapter-Klassen für Listener-Interfaces.

Exkurs (`UnsupportedOperationException`):

In Aufgabe H3.1 ändert die Methode `fillUp` für negative Werten des Parameters `fillValue` keine Attribute des Objekts `FuelDrivenVehicle`. Der Aufruf wird quasi ignoriert. Es ist aber auch vorstellbar, dass man erlauben möchte, dass die Methode `fillUp` auch negative Parameterwerte akzeptiert und man dadurch die Möglichkeit hat, Kraftstoff aus dem Auto zu entfernen.

Hierbei kann es in der Praxis gut sein, dass man diese Möglichkeit in der Basisklasse noch nicht implementieren, sondern erst durch eine Unterklasse umsetzen möchte.

Besonders nützlich ist dabei die `UnsupportedOperationException` der Java-Standardbibliothek, die es einfach ermöglicht, optionale Funktionalitäten von Klassen zu ermöglichen. Hierbei wird in der Basisklasse `FuelDrivenVehicle` die Methode `fillUp` dahingehend angepasst, dass im Falle eines negativen Parameterwerts für `fillValue` nicht mehr nichts gemacht wird, sondern stattdessen eine `UnsupportedOperationException` geworfen wird. Diese Exception zeigt, wie der Name schon ahnen lässt, an, dass eine Operation nicht unterstützt ist. In diesem Fall das Reduzieren des Attributs `fillingLevel`.

Eine Klasse, die `FuelDrivenVehicle` erweitert, kann dabei nun `fillUp` überschreiben, sodass keine `UnsupportedOperationException` mehr geworfen wird und stattdessen der Wert von `fillingLevel` um `fillValue` reduziert wird oder welches Verhalten hier sonst noch vorstellbar ist.

H3.2: `ElectricallyDriven` durch `ElectricBoat` implementieren — 7 Punkte

Beachten Sie die verbindlichen Anforderungen am Ende von H3.2, bevor sie mit H3.2 loslegen!

Schreiben Sie in einer Datei `ElectricBoat.java` eine Klasse `ElectricBoat`, die die Klasse `MeansOfTransport` erweitert und das Interface `ElectricallyDriven` implementiert. Zusätzlich soll `ElectricBoat` das Interface `IntSupplier` aus der Java-Standardbibliothek implementieren. Schauen sie sich dazu noch einmal Kapitel 03a, Folien 10-46 an und suchen Sie im Internet nach dem exakten Namen des Packages, in dem `IntSupplier` zu finden ist.

Die Klasse `ElectricBoat` hat ein `private`-Attribut `specificType` vom Typ `byte` sowie zwei `private`-Attribute vom Typ `int`: `currentCharge` und `capacity`. Für jedes der drei Attribute hat `ElectricBoat` eine `public` `get`-Methode mit dem üblichen Namensmuster.

Die Methode `standardVoltageChargeable` liefert genau dann `true` zurück, wenn der momentane Wert von `specificType` gleich 6, 11, 12 oder 22 ist; `highVoltageChargeable` hingegen liefert genau dann `true` zurück, wenn der momentane Wert n von `specificType` restlos durch 2 und $n + 1$ durch 3 teilbar ist.[3]

Methode `letsGo` addiert den aktualen Wert von `additionalChargeVolume` auf `currentCharge`, aber nur bis maximal `capacity`. Dann ruft Methode `letsGo` noch `letMeMove` mit dem aktualen Wert des eigenen Parameter `distance` auf. Die Rückgabe dieses Ausdrucks wird nicht verwendet, das heißt, `letMeMove` wird in `letsGo` so aufgerufen, als wäre es eine `void`-Methode.

Bei jedem Aufruf von `letMeMove` wird `currentCharge` um den Wert 1 verringert, aber nicht ins Negative und auch nicht um mehr als `distance/100`. Analog zu `FuelDrivenVehicle` liefert `letMeMove` hier den Wert zurück um den `currentCharge` reduziert wurde zurück.

Die parameterlose Methode `getAsInt` (definiert in `IntSupplier`) liefert die Differenz aus `capacity` (Minuend) und `currentCharge` (Subtrahend) zurück.

Daneben soll `ElectricBoat` noch eine `public`-Methode `setSpecificType` mit einem Parameter `specificType` vom Typ `byte` und Rückgabetyp `byte` haben. Falls der aktuale Wert des Parameters `specificType` kleiner 0 ist, soll das Attribut `specificType` den Wert 0 bekommen, bei einem Wert größer 30 soll es den Wert 30 bekommen – ansonsten den aktualen Wert des Parameters. Rückgabe ist der vorherige Wert des Attributs `specificType`.

Der `public`-Konstruktor von `ElectricBoat` hat ebenfalls einen Parameter `specificType` vom Typ `byte` und ruft damit einfach `setSpecificType` auf. Das Attribut `transportType` der Basisklasse `MeansOfTransport` wird im Konstruktor auf `VESSEL` gesetzt. Der zweite und der dritte Parameter des Konstruktors, `currentCharge` und `capacity`, sind vom Typ `int` und initialisieren die beiden gleichnamigen Attribute. Falls der aktuale Wert des Parameters `capacity` negativ (also kleiner 0) ist, wird das Attribut `capacity` allerdings auf 0 gesetzt. Selbiges gilt auch für Parameter `currentCharge` und das gleichnamige Attribut. Falls danach der aktuale Wert des Parameters `currentCharge` größer als der Wert des Attributs `capacity` ist, wird der Wert des Attributs `currentCharge` auf den Wert des Attributs `capacity` gesetzt.

Verbindliche Anforderungen:

- Der Rumpf von `standardVoltageChargeable` bzw. `highVoltageChargeable` besteht jeweils aus einer einzigen Anweisung, nämlich einer `return`-Anweisung. Die oben beschriebene Logik von `standardVoltageChargeable` bzw. `highVoltageChargeable` ist also vollständig in dem Ausdruck nach `return` zu realisieren (siehe Kapitel 01b, Folien 164-171).
- In `setSpecificType` ist der Bedingungsoperator nicht erlaubt.

Unbewertete Verständnisfragen:

- Was sagt der Compiler dazu, wenn Sie `getAsInt` mit einer Referenz von `MeansOfTransport` aufrufen, die auf ein Objekt von `ElectricBoat` verweist? Warum ist das sinnvoll, obwohl diese Methode und das von ihr gesetzte Attribut für `ElectricBoat` doch definiert sind? Vergleichen Sie dazu Kapitel 03b, Folien 141-168.
- Was würde passieren, wenn Sie das Attribut `capacity` im Konstruktor oben nicht auf 0 setzen würden im Fall, dass das Attribut `capacity` negativen Wert hat? Probieren Sie es einmal aus, indem Sie die Setzung des Attributs `capacity` im Konstruktor auf 0 auskommentieren, den Konstruktor mit einem negativen aktualen Wert für den Parameter `capacity` aufrufen und sich dann den Wert des Attributs `capacity` auf der Konsole ausgeben lassen. Schauen Sie bei Interesse auch schon einmal in Kapitel 03c, Folien 146-154.

[3] Dass diese Spezifikation sachlich unsinnig ist und nur der Übung dient, ist Ihnen sicherlich klar.

H3.3: `FuelDriven` und `ElectricallyDriven` zugleich implementieren — 6 Punkte

Ab jetzt unterscheiden wir im Rest dieses Hausübungsblattes zwischen Objekt- und Klassenattributen, siehe Kapitel 03b, Folien 168-184.

Schreiben Sie in einer Datei `HybridType1.java` eine Klasse `HybridType1`, die keine `extends`-Klausel hat, aber sowohl `FuelDriven` als auch `ElectricallyDriven` implementiert. Die Klasse `HybridType1` hat vier `private`-Klassenattribute: `fuelType` vom Typ `FuelType`, `averageConsumption` vom Typ `double` sowie `standardVoltageChargeable` und `highVoltageChargeable` vom Typ `boolean`. Die entsprechenden vier lesenden Methoden von `FuelDriven` bzw. `ElectricallyDriven` liefern einfach jeweils den momentanen Wert der zugehörigen Klassenvariable. Zu `fuelType` und `averageConsumption` hat `HybridType1` auch die üblichen `public` `set`-Methoden. Daneben hat `HybridType1` zwei rückgabelose, parameterlose `public`-Methoden `toggle{Standard,High}VoltageChargeable`, die den Wert der jeweils zugehörigen Klassenvariable negieren. Die Methode `letsGo` hat einen leeren Methodenrumpf. Ein Konstruktor wird für `HybridType1` nicht implementiert.

Schreiben Sie in einer Datei `HybridType2.java` eine Klasse `HybridType2`, die `MeansOfTransportation` erweitert und sowohl `FuelDriven` als auch `ElectricallyDriven` implementiert. Die Klasse `HybridType2` hat eine `private`-Objektvariable `hybridObject` vom Typ `HybridType1`. Der `public`-Konstruktor von `HybridType2` richtet ein Objekt von `HybridType1` ein und lässt `hybridObject` darauf verweisen. Alle Aufrufe von Methoden, die `HybridType2` von `FuelDriven` oder `ElectricallyDriven` erbt, delegieren ihre Aufgabe einfach an die jeweils entsprechende Methode von `hybridObject`. Die Methode `letMeMove` liefert 0 zurück und hat sonst keinen Effekt.

Unbewertete Verständnisfragen:

***Fehlermeldungen besser verstehen*:** Richten Sie eine Referenz vom Typ `HybridVehicle` ein und lassen Sie sie auf ein Objekt von `HybridType1` verweisen. Was sagt der Compiler? Was ist also generell nicht möglich bei einer Klasse, die zwei Interfaces implementiert, im Vergleich zu einer Klasse, die ein beide erweiterndes Interface implementiert?

***Verständnisfrage am Rande*:** Richten Sie zwei Referenzen von `HybridType1` ein, ändern Sie den Wert von `averageConsumption` mittels der zugehörigen `set`-Methode und lassen Sie sich vorher und hinterher mittels `get`-Methode den Wert von `averageConsumption` über beide Referenzen ausgeben (also insgesamt vier Aufrufe der `get`-Methode). Entsprechen die Ergebnisse Ihrem Verständnis von Klassenattributen aus Kapitel 03b?

H3.4: `HybridVehicle` implementieren — 3 Punkte

Schreiben Sie in einer Datei `HybridType3.java` eine Klasse `HybridType3`, die keine `extends`-Klausel hat und `HybridVehicle` implementiert. Die Funktionalität von `HybridType3` ist völlig identisch zu der von `HybridType1` – inklusive der Tatsache, dass die Attribute Klassenattribute und nicht Objektattribute sind und sich entsprechend verhalten. Für die Klassenvariable `preferredDriveType` schreiben Sie analog eine `public`-Methode `togglePreferredDriveType`, die keine Parameter und Rückgabe besitzt, aber immer zwischen `FUEL_BASED` und `ELECTRICAL` umschaltet.

Verbindliche Anforderungen:

- Der Rumpf von `togglePreferredDriveType` besteht aus einer einzigen Anweisung und zwar dem Bedingungsoperator.