F04: 🚄 MünchenÖPNV

> Diese Aufgabe ist Teil der freiwilligen inoffiziellen Zusatzaufgaben von Eric Jacob und Jonas Wende, erstellt im WS 23/24 für *IN0002: Grundlagenpraktikum Programmierung*.
> Weder sind sie durch die ÜL überprüft, noch unbedingt vollständig richtig.
> Fehler gerne melden: eric.jacob.2003@gmail.com

🎯 Lernziele

Diese Aufgabe dient der Wiederholung folgender Konzepte:

- Klassen
- Vererbung
- Interfaces

- Enums
- Text zu UML
- UML zu Code
- Graphen

📜 Backstory

Da die Pinguine bereits sehr viel Gutes über eure ausgezeichneten Programmierfähigkeiten gehört und in den letzten Wochen an euren H-Aufgaben auch gesehen haben, möchten sie euch vor der 2. bPHA besuchen. München soll ja auch eine tolle Stadt sein, und welcher Pinguin trinkt nicht gerne gelegentlich ein Bier zu seiner Schneeweißwurst? Gerade am Flughafen mit *AntarcticAir* gelandet, wollen sie euch nun am Forschungszentrum besuchen kommen - doch sie haben leider keine Ahnung vom Münchner ÖPNV-Netz. Am besten verstehen die Babypinguine, die noch nicht so gut lesen können, UML-Diagramme, wohingegen die erwachsenen Pinguine besonders Java-Code einfach begreifen — kannst du den MVV also für sie verständlich erklären?

📝 Aufgabenbeschreibung

Ziel dieser Aufgabe ist es, Vererbung und UML-Diagramme zu wiederholen. Insgesamt ist die Beschreibung absichtlich offen gehalten, sodass ihr selbst entscheiden könnt, wie genau ihr bestimmte Funktionalitäten implementiert – seid kreativ! Der Lösungsvorschlag ist also genau das - ein **Vorschlag**.

> Eigentlich war der Plan, eure fertige Klassenstruktur mit dem Anfragen der MVG-API zu testen, aber da ihr mit diesen Daten auch nicht viel machen außer anzeigen könntet, haben wir uns dazu entschieden, mehr auf Übung anstatt praktische Anwendung zu setzen — vielleicht kommt das Feature ja noch irgendwann dazu (solltet ihr dazu motiviert sein, erstellt gerne eine Pull Request :)).

Hinweis: Wir betrachten — wie bei Vererbung üblich — nur einen Ausschnitt des Münchner Verkehrsnetzes. Größtenteils beschränken wir uns dabei auf Verkehrsmittel, die zwischen Flughafen und Forschungszentrum eine sinnvolle Verbindung darstellen (dementsprechend verzichten wir auf Trams und erlauben auch nicht alle 87 Busse, die in und um München verkehren).

Hinweis: Da diese Aufgabe mit Unterpackages arbeitet, liegt sämtlicher Code in einem Oberpackage namens `oepnv`. Dieses ist bereits vorgegeben – stelle sicher, dass alle deine Klassen, sofern nicht anders in der Beschreibung angegeben, in `oepnv` liegen.

Lies die folgende Beschreibung am besten erst einmal vollständig durch, bevor du mit der Implementation beginnst.

Grundlegend ist der Münchner Nahverkehr in unserer Miniwelt in drei Kategorien unterteilt: Menschen, Linien mit Fahrzeugen und Haltestellen sowie Straßen/Schienen.

Human

Alle Menschen haben einen Namen und ein Alter — auf beide können auch die "konkreteren" Menschen direkt zugreifen. Zudem können Menschen reden und für eine gegeben Dauer schlafen. In unserer Miniwelt gibt es keine einfachen "Menschen", lediglich konkretere Einheiten wie `Driver` oder `Passenger`.

Passenger

Selbstverständlich braucht der Münchner ÖPNV Fahrgäste. Diese sind (meistens – wir verzichten auf Haustiere) Menschen. Pro Fahrgast wird gespeichert, ob dieser gerade sitzt und ob er ein valides Ticket hat. Letzteres muss der `TicketInspector` abfragen können (er kann allerdings keine Tickets für ungültig erklären – unsere Fahrgäste sind also immer ehrlich und wissen, ob ihr Fahrschein gültig ist).

Eine Lieblingsbeschäftigung der Fahrgäste ist es, sich über den ÖPNV aufregen zu können. Dabei reden sie jedoch meist ins Nichts (a.k.a nur auf die Konsole).

Zudem können sie Fahrzeuge betreten (`board(Vehicle)`) und diese auch wieder verlassen (`exit()`).

OEPNVEmployee

Jeder Mitarbeiter des ÖPNV hat ein Gehalt, das er jedoch lieber geheim hält. Zudem hat er die Möglichkeit zu streiken — wie genau ein solcher Streik aussieht, hängt jedoch immer vom konkreteren Beruf ab. Auch hier gilt: es gibt keine reinen `OEPNVEmployees`.

Driver

Ein Fahrer ist ein `OEPNVEmployee`, kann Announcements machen und besitzt potenziell mehrere `DriversLicense`s, die ihn zum Fahren einer bestimmten Fahrzeugart berechtigen. Zudem kann ein Fahrer selbstverständlich ein Fahrzeug fahren (`drive(Vehicle)`) und die Frage, ob er eine angefragte Art von Fahrzeug fahren darf, mit `true`/`false` beantworten (`isLicensedFor(Vehicle)`).

Scheduler

Ein `Scheduler` ist ein `OEPNVEmployee`, kann Netzpläne erstellen (`createNetPlan(Line[], TrackSegment[])`) und gibt den erstellten Netzplan als `String` zurück. Außerdem kann ein `Scheduler` im Falle einer Störung Schienenersatzverkehr mit Bussen veranlassen (`establishReplacementService(Line): Line`). Dafür muss er die betroffene Linie kennen und gibt die Linie, die den SEV übernimmt, zurück. Zudem hat er der Fähigkeit, Durchsagen zu machen.

TicketInspector

Ein `TicketInspector` ist natürlich ein `OEPNVEmployee` und schreibt sich für den Privatgebrauch auf, wie viele Strafen wegen Schwarzfahren in seiner gesamten beruflichen Laufbahn er bereits verteilt hat. Außerdem kann er in Fahrzeugen Ticketkontrollen durchführen (`inspectTickets(Vehicle)`) und gibt dabei die Anzahl an Schwarzfahrern bei dieser Kontrolle zurück. Außerdem sagt er beim Betreten eines Fahrzeugs stets an, dass die Fahrgäste bitte ihre Tickets raussuchen sollen, um die Kontrolle zu beschleunigen.

Hilfsklassen und co

DriversLicense

Dieses Enum speichert die möglichen Führerscheine für U-Bahn (`U_BAHN_LICENSE`), S-Bahn (`S_BAHN_LICENSE`) und Bus (`BUS_LICENSE`), die ein `Driver` haben kann.

MakeAnnouncements

Dieses Interface gibt die Methode `announce(String)` vor, welche der Fähigkeit, eine Durchsage machen zu können, entspricht.

Line

Diese Klasse speichert Informationen für eine bestimmte Linie (z.B. für alle U6-U-Bahnen oder alle X660-Busse) so, dass Fahrzeuge, die sich je eine Linie speichern, auf alle Attribute (Start- und Zielstation sowie `LineNumber`) zugreifen können.

Zudem speichert eine `Line` alle `Station`en in der Reihenfolge, in der sie auf der Linie liegen (bereits vorgegeben). Dafür wird die bereits implementierte Methode `initStations()` verwendet, die auf den Objektattributen arbeitet und eine `List<Station>` zurückgibt.

Eine Linie kann nie ohne Fahrzeug existieren.

Vehicle

Um den ÖPNV betreiben zu können, gibt es Fahrzeuge. Besonders wichtig ist natürlich, dass sich Fahrzeuge verspäten können – deshalb implementiert Vehicle `Delayable`. Jedes Fahrzeug, egal ob Bus, U- oder S-Bahn, speichert die eigene Anzahl an Rädern möglichst platzsparend (dabei haben Busse 6 Räder, S-Bahnen 28 und U-Bahnen 16), sowie die Linie, zu der es gehört. Für Wartungsarbeiten ist zudem hinterlegt, aus welchem Jahr ein Fahrzeug stammt (dafür kannst du den in Java-Utils gegebenen Typen `Date` verwenden – beim Erstellen eines neuen `Date`s enthält dieses automatisch das aktuelle Datum). Außerdem hat jedes Fahrzeug einen `Driver` und speichert die aktuellen Fahrgäste. Bedenke dabei, dass die Anzahl an Fahrgästen sehr stark variieren kann und Fahrgäste an beliebigen Stationen ein- und aussteigen können (und dies auch im Code können).

`Vehicle` gibt außerdem die Methode `move()` vor – wie genau sich ein Fahrzeug bewegt ist allerdings davon abhängig, ob es ein Bus, eine S- oder eine U-Bahn ist. Zudem kann ein Fahrzeug die Türen öffnen und schließen – auch dies ist vom konkreten Fahrzeugtypen abhängig.

UBahn

U-Bahnen speichern ihr Modell als Text – da besonders die zahlreichen Informatikstudenten in München dem Motto “Ich mag Züge” folgen und bei jeder U-Bahn wissen wollen, welches Modell diese hat, wird dieses öffentlich einsehbar gespeichert.

Zudem kann eine U-Bahn das *Münchner Fenster* anzeigen lassen.

SBahn

S-Bahnen sind für Zug-Nerds ebenso interessant wie U-Bahnen – entsprechend gilt für das Modell der S-Bahn selbiges wie bei `UBahn`.

Zudem können sich S-Bahnen teilen (`splitTrains()`) – das tut die S1 beispielsweise in Neufahrn oder die S2 in Dachau. Zurückgegeben werden dabei die "neu entstandenen" S-Bahnen. Entsprechend kann ein `SBahn` sich auch mit einer anderen S-Bahn verbinden (`connectTrains(SBahn)`), sofern diese die gleiche `LineNumber` und das gleiche Modell hat – ob die Verbindung erfolgreich war, gibt die Methode zurück.

Bus

Busse haben die Fähigkeit, die rechte Seite (mit den Türen) abzusenken, um den Einstieg zu erleichtern.

LineNumber

Liniennummern dienen der Zuordnung von Fahrzeugen und Fahrgästen der Orientierung.

Um die Klassenstruktur übersichtlicher zu gestalten, liegen alle Unterklassen von `LineNumber` (nicht aber `LineNumber`!) in einem separaten Package in `oepnv` (bereits vorgegeben) namens `linenumbers` (noch nicht vorgegeben). Dieses kannst du durch einen Rechtsklick auf `oepnv` und dann `New > Package` erstellen.

Das Interface `LineNumber` hat dabei keine besonderen Eigenschaften, ermöglicht allerdings die Generalisierung der Bus-, U- und S-Bahn-Liniennummern (sodass in `Line` eine `LineNumber` als Datentyp ausreicht, anstatt für alle 3 Untertypen eine Variable anzulegen, die möglicherweise belegt wird).

> *Vergleich:*

```
interface A {}
class B implements A {}
enum C implements A {TYPE1, TYPE2}
//...

A bIsAnA = new A();
A cIsAnA = B.TYPE1;
```

Wichtig: Da die konkreten Liniennummern (siehe unten) Enums sind, können diese keine Klassen extenden; nur Interfaces implementieren. Daher ist `LineNumber` ein Interface.

BusLineNumber

Dieses Enum implementiert `LineNumber` und speichert die in unserer Miniwelt vorhandenen und relevanten Buslinien:

```
B292,
B230,
B690,
X201,
X660
```

Hinweis: Eigentlich haben `292`, `230` und `690` kein `B` (so wie Expresslinien ein `X`) vor der Nummer. Allerdings müssen Enums in Java mit einem Buchstaben beginnen, daher das `B` für "Bus".

UBahnLineNumber

Dieses Enum implementiert `LineNumber` und speichert die in unserer Miniwelt vorhandenen und relevanten U-Bahn-Linien:

```
U2,
U3,
U6
```

SBahnLineNumber

Dieses Enum implementiert `LineNumber` und speichert die in unserer Miniwelt vorhandenen und relevanten S-Bahn-Linien:

```
S1,
S8
```

Hilfsklassen und co

Delayable

Was wäre der Münchner Nahverkehr ohne Verspätungen? Daher gibt es in `Delayable` die Methode `delay(int, Reason)`, die ein Fahrzeug um eine gegebene Minutenzahl wegen eines gegebenen Grunds verspäten kann (das sagt der `Driver` des `Vehicle`s an).

Reason

Dieses Enum gibt mögliche Gründe für Verspätungen an. Diese sind:

```
VERDI_STRIKE,
GDL_STRIKE,
SNOWFALL,
RAIN,
SUNSHINE,
LACK_OF_DRIVERS,
DELAY_OF_PREVIOUS_TRAIN,
PERSONS_ON_TRACKS,
CABLE_DAMAGE,
SWITCH_DAMAGE,
DOOR_DAMAGE,
SIGNAL_DISTURBANCE,
NO_REASON
```

Wir haben beim Erstellen auf Sonderfälle wie “Kühe auf den Gleisen” verzichtet – fühlt euch frei, eigene Gründe zu ergänzen.

Track

Damit unsere Fahrzeuge auch auf einer Straße bzw. Schiene fahren können, gibt es `TrackSegment`s. Ein `Track` (Strecke) ist eine strukturierte lineare Ansammlung an `TrackSegment`s.

TrackSegment

Jedes Segment hat eine Breite und Länge, sowie zwei Koordinaten (ein Array ist ausreichend). Denke daran, Breite, Länge und die Koordinaten genau zu speichern (Vgl. eine typische Koordinate sieht so aus: `48.2658797`). Außerdem speichert jedes `TrackSegment`, für welche `Vehicles` es geeignet ist.

Der `Scheduler` muss bei der Erstellung des Fahrplans in der Lage sein, für ein übergebenes `Vehicle` und `TrackSegment` abzufragen, ob dieses für das entsprechende Segment geeignet ist (mache die Methode statisch).

SwitchSegment

Weichen sind `TrackSegments`, die zudem speichern, ob sie aktuell geschalten sind (wir betrachten nur Weichen, die genau 2 mögliche Ausrichtungen haben). Zudem kann die Weiche von außen umgeschaltet werden.

Station

Das Enum `Station` speichert alle in unseren Miniwelt möglichen Haltestellen (bereits vorgegeben).

Zudem gibt `Station` eine Methode (mit `default`-Implementierung!) vor (noch zu implementieren):

- `isIn(Station): boolean`: Die Pinguine in unserer Miniwelt sind durchaus sportlich; so ist es für sie kein Problem, ein wenig durch den Ort laufen zu müssen, um ihren Umstieg zu erreichen. Diese Methode vergleicht, ob die aktuelle Station und die übergebene im selben Ort liegen. Dies wäre z.B. bei `GARCHING` und `GARCHING_HOCHBRUECK` der Fall, aber auch bei `GARCHING_FORSCHUNGSZENTRUM` und `GARCHING_HOCHBRUECK`.

  Tipp: mit `.split("abc")` kann man einen String an allen Stellen “abc” teilen. Zurückgegeben wird ein Array mit allen Einzelteilen. “abc” wird dabei entfernt. Beispiel: `“hallo welt”.split("l")` gibt `["ha", "", "o we", "t"]` zurück.

🪜 Aufgaben

1. Text zu UML

Wandle die obige Beschreibung in ein UML-Diagramm um. Dafür kannst du zum Beispiel Apollon verwenden. Getter, Setter und Konstruktoren musst du nicht in dein UML-Diagramm schreiben, kannst es aber natürlich trotzdem tun.

Hinweis: Das Package `oepnv` musst du nicht für jede Klasse angeben. Unter-Packages allerdings schon!

In S- und U-Bahn wird das Modell `public` gespeichert. Welche Vor- und Nachteile kann das haben?

2. UML zu Code

Erstelle anhand deines UML-Diagramms nun die entsprechenden Java-Klassen und implementiere diese korrekt in `MuenchenOEPNV-Template`. Stelle sicher, dass alle Klassen im richtigen Package liegen – sollte die Zeile `package xx.yy;` nicht automatisch von IntelliJ eingefügt werden, musst du sie selbst dazuschreiben.

Achte darauf, Konstruktoren, Getter und Setter zu ergänzen, wo dies sinnvoll ist.

Hinweis: Manche Klassen, Enums und Interfaces existieren im Template bereits – sind aber, wenn nicht anders angegeben, unvollständig/leer. Sie dienen nur dazu, dass IntelliJ nicht direkt beim Öffnen des Projekts sämtlichen Code rot unterstreicht.

3. Expansion

Aufgrund der näherrückenden ÜPA haben es die Tutoruine leider nicht mehr geschafft, einen sinnvollen Anwendungszweck für die modellierte Miniwelt zu entwerfen (dementsprechend gibt es hierfür keine Lösungen). Dennoch ist es eine gute Übung, wenn ihr hier kreativ werdet und selbst noch Erweiterungen bzw. sinnvolle Funktionalitäten ergänzt. Hier einige Ideen, was denkbar wäre, mit zugehöriger Schwierigkeitsstufe:

- Modelliere einen Tagesablauf, wie er im Münchner Nahverkehr vorkommen könnte (effektiv: teste deinen Code durch einige Funktionsaufrufe). *(einfach)*
- Füge weitere Teile des ÖPNVs in München hinzu (z.B. Tram, Techniker, ...). Passe ggf. die Vererbungen und Interfaces an. *(einfach)*
- Anstatt bei `isIn()` anhand der Strings auf die Zugehörigkeit einer Station in einem Ort zu prüfen, kann man für jede "Hauptstation" (z.B. `GARCHING`) alle "Unterstationen" (`GRACHING_HOCHBRUECK`, `GARCHING_SONNENSTRASSE`, `GARCHING_FORSCHUNGSZENTRUM`, ...) in einer durch die Hauptstation identifizierten Liste speichern und anhand dieser überprüfen, ob zwei Stationen in laufbarer Distanz liegen. *(medium)*
- Gegeben zwei Stationen, finde eine/alle mögliche(n) Verbindung(en). *(schwer)*
  Tipp: Dafür kannst du dich mit Graphen, Tiefen- und Breitensuche beschäftigen (nur bedingt Teil von PGdP).

🔍 Lösungsvorschlag

Siehe `Solution 1 - UML Diagram` bzw. `Solution 2 - UML in Java` für die entsprechenden Teilaufgaben (du kannst auch direkt UML zu Code bearbeiten, wenn du mit UML-Diagrammen bereits vertraut bist).

Antwort zu "In S- und U-Bahn wird das Modell `public` gespeichert. Welche Vor- und Nachteile kann das haben?":

Durch `public` kann das Attribut von außen gelesen, als auch überschrieben werden. Man benötigt entsprechend keine Getter/Setter – allerdings könnten die Informatik-Nerds das Modell einer S-Bahn auch einfach verändern, ohne, dass diese tatsächlich umgebaut wird.

🗃️ Anhang

MVG-API

Die API als Schnittstelle ist leider noch im Aufbau (es gibt wohl einige Umwege über Python, aber das ist sehr umständlich). Allerdings gibt es hier Datensätze zu Fahrtzeiten und -plänen des MVG — falls ihr diese parsen und in eurer Miniwelt speichern wollt: fühlt euch frei.