Funktionale und objektorientierte Programmierkonzepte Projektaufgabe

TECHNISCHE UNIVERSITÄT DARMSTADT

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.

FOP Projekt **Gesamt:**
Pizza-Lieferservice

Verbindliche Anforderungen (für das ganze Projekt)**:**

(a) In diesem Projekt fordern wir wie in den Hausübungen Dokumentation mittels `JavaDoc`. Dokumentationen sind verpflichtend für:

1. **alle** Methoden (nicht nur in Klassen, sondern auch in **Interfaces** und **abstrakten** Klassen),
2. Konstruktoren und
3. Interfaces.

Informationen dazu finden Sie unter anderem auf Übungsblatt 03.

(b) Alle Aufgaben (außer JUnit-Tests) sind in Package `projekt` im Verzeichnis `src/main/java/` umzusetzen. Achten Sie darauf, dass Sie alle Dateien genau in diesem Package erzeugen. Die **Packages** werden in den jeweiligen **Aufgabenteilen spezifiziert**. Die **JUnit**-Tests schreiben Sie im Verzeichnis `src/test/java/`. Die Packages entsprechen den jeweiligen Packages der zu testenden Klassen.

(c) Verwenden Sie nur die vorgegebenen Modifier. Sollte keine Modifier vorgegeben sein, nutzen Sie die jeweiligen Standard-Modifier. (Keine Angabe von Modifier)

(d) Achten Sie bei Dateinamen, Bezeichnern und Strings darauf, diese **exakt** wie gefordert einzugeben - das heißt: Verändern Sie nicht die Schreibweise (auch nicht die Groß- und Kleinschreibung), fügen Sie keine Satzzeichen hinzu und übersetzen Sie nichts in eine andere Sprache, außer es wird **explizit** in der Aufgabenstellung gefordert.

(e) Die Parameter der von Ihnen zu implementierenden Methoden müssen *exakt* in der angegebenen Reihenfolge deklariert werden.

(f) Zeiten sollen nur minutengenau sein. Das heißt, wenn sich zwei Zeitpunkte nur im Sekundenbereich oder im Bereich von Sekundenbruchteilen unterscheiden, sollen sie als gleich angesehen werden.

(g) In Package `projekt.food` müssen alle Klassen, mit Ausnahme von nicht von außen instanziierbaren `final`-Klassen wie `Extras` und `FoodTypes`, `package-private` sein. Interfaces sind alle `public`.

Anmerkung:

Mit `java.util.List` und `java.util.Map` sind die Typen aus `java.util` gemeint.

Eine bekannte Pizzeria-Kette hat sich dazu entschlossen, zusätzlich einen Lieferdienst anzubieten. Sie wurden ausgewählt dieses Vorhaben in die Tat umzusetzen.

H1: Location

Erweitern Sie die bereits von Ihnen implementierte Klasse `public final class Location` im Package `package projekt.base`.

> **Hinweis:**
>
> Für die hier durchzuführenden Änderungen existieren public-Tests in `test/LocationTest.java`. Nutzen Sie die Tests, um Ihre Implementierungen auf korrekte Funktionsweise zu testen.

H1.1:

Ändern Sie die Klasse so um, dass sie das Interface `Comparable<Location>` implementiert. Fügen Sie hierzu zunächst den nötigen Befehl `implements Comparable<Location>` hinzu.

H1.2:

Implementieren Sie die Methode `public int compareTo(Location o)`, welche die aktuelle Location mit der gegebenen vergleicht.

Die Methode `compareTo` soll genau dann `0` liefern, wenn sowohl die x- als auch die y-Koordinaten beider Locations übereinstimmen. Ein negativer Wert wird genau dann geliefert, wenn die x-Koordinate der aktuellen Location kleiner als die der anderen Location ist oder die x-Koordinaten beider Locations übereinstimmen und gleichzeitig die y-Koordinate der aktuellen Location kleiner als die der anderen ist. Im letzten Fall wird ein positiver Wert geliefert.

> **Hinweis:**
>
> Überlegen Sie sich, ob die Erstellung eines statischen `Comparator<Location>` als Attribut der Klasse sinnvoll ist, um die Koordinaten zu vergleichen.

H1.3:

Implementieren Sie die Methode `public int hashCode()`.

Die Methode `hashCode` soll mindestens für alle paarweise verschiedenen[1] Locations l_1, l_2 mit Koordinaten $x, y \in \{n \in \mathbb{N} : -1024 \leq n \leq 1023\}$ unterschiedliche Werte zurückliefern. Bei Ihrer Implementierung ist Ihnen freigestellt, wie Sie dies Vorgabe umsetzen.

[1] Damit ist gemeint, dass sich die x- und/oder y-Koordinaten dieser unterscheiden.

H1.4:

Implementieren Sie die Funktion `public boolean equals(Object o)`, welche das gegebene Objekt `o` mit `this` auf Objektgleichheit überprüft und das Resultat zurückliefert.

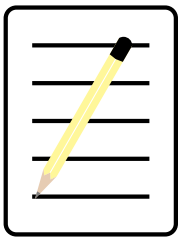
Zwei Objekte `l1`, `l2` des Typs `Location` werden als *objektgleich* bezeichnet, wenn die Koordinaten dieser übereinstimmen. Im Fall, dass `o` `null` oder nicht vom Typ `Location` ist, soll `false` geliefert werden.

H1.5:

Implementieren Sie die Methode `public String toString()`, welche die Koordinaten der Location im Format `"<x>,<y>"` ausgibt, wobei `<x>` und `<y>` durch die entsprechenden Koordinaten ersetzt werden.

> **Verbindliche Anforderung:**
>
> Formatieren Sie den String so, dass auch tatsächlich zwei dezimale Integer in die Konsole geschrieben werden.

H1.6: Daten für Bestellungen



Wir möchten nun die Daten für Bestellungen aufnehmen. Hierzu stellen wir Ihnen die Klasse `ConfirmedOrder` bereit. Machen Sie sich zunächst mit dieser vertraut. Die Klasse `ConfirmedOrder` besitzt folgende Objektattribute:

Attributname	Typ	Beschreibung
`location`	`Location`	Die Koordinaten des Zielorts.
`orderID`	`int`	Die ID der Bestellung.
`timeInterval`	`TimeInterval`	Der Zeitraum, in dem die Lieferung erfolgen soll.
`actualDeliveryTime`	`LocalDateTime`	Der Zeitpunkt, an dem die Lieferung tatsächlich erfolgt ist.
`foodList`	`List<Food>` [2]	Die eigentliche Bestellung; eine Liste von Gerichten, die geliefert werden sollen.

Der `public`-Konstruktor hat für jedes der vier Objektkonstanten einen Parameter, und zwar in der obigen Reihenfolge. Wie gewohnt werden im Konstruktor diese auf die aktualen Parameterwerte gesetzt. Der Konstruktor geht ohne Überprüfung davon aus, dass die Referenztypen in der Parameterliste nicht auf `null` verweisen.

Zu jedem der fünf Attribute hat `ConfirmedOrder` jeweils eine `get`-Methode. `actualDeliveryTime` hat zusätzlich noch eine `set`-Methode in der aus der Vorlesung bekannten Form (ebenfalls `public`).

[2] `Food` wird in Aufgabe H?? genauer beschrieben.

Unbewertete Verständnisfrage:

Warum können wir es uns leisten, so ganz einfache Begriffe wie `DistanceCalculator`, `TimeInterval` und `ConfirmedOrder` zu verwenden? Anders gefragt: Warum müssen wir nicht befürchten, dass andere Leute dieselben Namen verwenden und es dann bei der Zusammenführung der Programme verschiedener Leute zu Namenskonflikten kommt? Falls Ihnen nichts einfällt, schauen sie nochmals in Kapitel 03a.

H2: Routing - Wo bin ich und wo will ich hin? *Interface Region*

Hinweis:

Für die hier durchzuführenden Implementierungen existieren public-Tests in `test/RegionTest.java`. Nutzen Sie die Tests, um Ihre Implementierungen auf korrekte Funktionsweise zu testen.

Der Lieferdienst benötigt eine Karte der Region, um die Ziele und Wege zur Lieferung modellieren zu können. Hierzu stellen wir Ihnen das Interface `Region` zur Verfügung. Eine Region bildet den Landstrich ab, auf dem sich sowohl Restaurent, Lieferfahrzeug, Straßen und Zielort wiederfinden lassen. Hierzu modellieren Sie einen Graphen. Ein Graph besteht aus Knoten und Kanten. Jede Kante stellt eine Straße dar. Jede Straße besitzt einen Anfang und ein Ende. Dort können weitere Straßen anknüpfen, müssen aber nicht. Diese Anfänge und Enden von Straßen sind Knoten. Knoten können also von Kanten verbunden werden, um ein Straßennetzwerk zu erzeugen.

Es gibt zwei Arten von Knoten: Die Nachbarschaften und die ”normalen”Knoten.

Nachbarschaften stellen die Orte, Stadtteile und Wohnviertel dar, an denen sich Häuser befinden, die man vom Knoten aus per Lieferfahrzeug erreichen kann. Hierzu muss natürlich überprüft werden, ob die Koordinaten einer Bestellung nahe genug an einer Nachbarschaft bzw. einem Stadtteil liegen, damit dorthin eine Lieferung erfolgen kann. Die „normalen“ Knoten stellen Autobahnkreuze dar. Für die Lieferung sind sie eigentlich uninteressant, sie sind eher Mittel zum Zweck, das Lieferfahrzeug muss ja schließlich irgendwie zur Nachbarschaft kommen. Ein Haus bzw. ein Zielort kann zwar in der Nähe eines Autobahnkreuzes liegen, aber der Lieferant kann nicht einfach so auf dem Standstreifen anhalten, über die Leitplanke und die Lärmschutzwand klettern und Pizza liefern.

Damit ein Lieferfahrzeug vom Restaurant zu einem Zielort und der dem Zielort nächstgelegenen Nachbarschaft kommen kann, muss es Straßen befahren. Dazu passiert es die Kanten und Knoten des Graphen.

Damit überhaupt eine Bestellung ausgeliefert werden kann, muss anhand der Koordinaten der Bestellung der nächstgelegene Nachbarschaftsknoten gefunden werden, zu dem die Bestellung dann geliefert wird. Dafür wird eine Funktion zur Bestimmung der Distanz zwischen Koordinaten und Knoten verwendet. Dieses Ergebnis wird dann mit dem Radius des jeweiligen Knotens verglichen. Ist die Distanz zwischen den Koordinaten der Bestellung und dem Nachbarschaftsknoten größer als der im Nachbarschaftsknoten definierte maximale Entfernungsradius, kann keine Lieferung erfolgen.

In den folgenden Unteraufgaben implementieren Sie das Interface `Region` als `class RegionImpl implements Region`. Die Klassendatei hierfür stellen wir Ihnen inklusive einiger Basismethoden und den benötigten Attributen bereits zur Verfügung. Hierzu müssen Sie ggf. auskommentierten Code „reaktivieren“.

Um die Karte zu modellieren, benötigen Sie die bereitgestellte Liste `allEdges`, die alle Kanten enthält, die HashMaps `nodes` und `edges` und die Collections `unmodifiableNodes` und `unmodifiableEdges`. In der HashMap `nodes` werden den Locations die entsprechenden Knoten zugewiesen, in der HashMap `edges` werden einer Edge zwei Locations zugewiesen.

Erinnerung:

Basisinformationen zur Arbeit mit Maps finden Sie in Foliensatz 07.

Anmerkung:

Notiz am Rande:
In einer HashMap können immer nur zwei Werte einander zugeordnet werden, daher müssen wir den Umweg gehen, in die eigentliche HashMap eine Location und dann eine HashMap mit einer Location und der Edge zu packen.

Verbindliche Anforderung:

Alle Implementierungen von Funktionen, die Objekte der Typen `Map`, `List`, `Collection` (inklusive Subtypen wie `Set`) zurückgeben, **müssen** unmodifiable-Versionen zurückgeben. Sie erhalten diese durch den Aufruf von `Collections.unmodifiable*`, wobei * dann Map, List, Collection,... ist.

Unbewertete Verständnisfrage:

Mittels der Methoden `unmodifiable*` in der Klasse `Collections` erstellen Sie eine Variante des Objekts, die keinen modifizierenden Methodenaufruf erlaubt. Warum könnte dies für Sie hier von Vorteil sein?

H2.1:

Implementieren Sie die Funktion `public @Nullable Node getNode(Location location)`, welche den Knoten aus der Map `nodes` einer übergebenen Location zurückgeben soll.

Verbindliche Anforderung:

`public @Nullable Node get(Location location)` soll unter Verwendung der Methode `get` des Interfaces `Map`.

Anmerkung:

Mit der Annotation `@Nullable` zeigen Sie an, dass ein Referenztyp auch `null` sein darf.

H2.2:

Implementieren Sie die Methode `void putNode(NodeImpl node)`, die einen Knoten in die Map `nodes` hinzufügen soll. Wenn der übergebene Knoten nicht in dieser Region liegt soll eine `IllegalArgumentException` geworfen werden. Als Message-Text verwenden Sie bitte „`Node "+ node + "has incorrect region`“

H2.3:

Implementieren Sie die Funktion `public @Nullable Edge getEdge(Location locationA, Location locationB)`, die die Kante zweier Locations zurückgeben soll.

Hinweis:

Achten Sie darauf, dass die zwei Locations auch in umgekehrter Reihenfolge übergeben werden können und die Funktion trotzdem die korrekte Kante liefern muss.

H2.4:

Implementieren Sie die Methode `void putEdge(EdgeImpl edge)`, die eine übergebene Kante sowohl in die zweidimensionale Map `edges` als auch in die eindimensionale Liste `allEdges` hinzufügt. Wenn die übergebene Edge nicht in dieser Region (`this`) liegt, soll eine `IllegalArgumentException` geworfen werden.

H2.5:

Implementieren Sie zunächst die Funktionen `public Collection<Node> getNodes()` und `public Collection<Edge> getEdges()`, welche die jeweilige Collection (z.B. `unmodifiableNodes`) zurückgeben sollen.

H2.6:

Implementieren Sie die Methode `public boolean equals(Object o)`. Sie soll das übergebene Objekt mit dem aktuellen Objekt (`this`) vergleichen. Sie soll `true` zurückgeben, wenn das übergebene Objekt `o` gleich `this` ist. Falls das übergebene Objekt `null` oder nicht vom Typ `RegionImpl` ist, soll `false` zurückgegeben werden. Ansonsten tritt der Standardfall ein. Es soll `Objects.equals()` mit `nodes`, `o.nodes` und einmal mit `edges`, `o.edges` aufgerufen

werden. Beachten Sie, dass `o` dazu von `Object` zu `RegionImpl` gecastet werden muss.

H2.7:

Implementieren Sie die Funktion `public int hashCode()`, welche den Hashcode der Knoten und Kanten zurückgeben soll.

> **Hinweis:**
>
> In Java können Hash-Codes durch den Aufruf von `Objects.hash(...)` erzeugt werden.

H3: Routing - Knoten ohne Ende *Interface Node*

> **Hinweis:**
>
> Für die hier durchzuführenden Implementierungen existieren public-Tests in `test/NodeTest.java`. Nutzen Sie die Tests, um Ihre Implementierungen auf korrekte Funktionsweise zu testen.

Um die Region mit Leben zu füllen und z.B. Autobahnkreuze oder Nachbarschaften realisieren zu können benötigen wir einen Typen für unsere Knoden. Implementieren Sie hierzu das Interface `Node` als `class NodeImpl implements Region.Node`. Die Klassendatei inklusive der Attribute, des Konstruktors und einiger Standardmethoden geben wir Ihnen vor.

Ein Knoten besitzt eine Region, zu der er gehört. Außerdem hat er einen Namen, eine Location (also Koordinaten x und y) und ein Set von Koordinaten, mit denen der Knoten verbunden ist.

H3.1:

Implementieren Sie die Methoden `public Region getRegion()`, `public Location getLocation()` und `public String getName()`, die das jeweilige Attribut zurückgeben sollen.

H3.2:

Implementieren Sie die Funktion `public @Nullable Region.Edge getEdge(Region.Node other)`, welche die Edge, die den (aktuellen) Knoten mit dem übergebenen Knoten verbindet, zurückgeben soll.

> **Hinweis:**
>
> Benutzen Sie für diese Rückgabe das Attribut `region`

H3.3:

Implementieren Sie die Funktion `public Set<Region.Node> getAdjacentNodes()`, welche alle Knoten zurückliefern soll, die mit dem (aktuellen) Knoten verbunden sind.

> **Hinweis:**
>
> Benutzen Sie für diese Collection das Attribut `region`

H3.4:

Implementieren Sie die Funktion `public Set<Region.Edge> getAdjacentEdges()`, welche alle mit dem (aktuellen) Knoten verknüpfte Kanten zurückgibt.

> **Hinweis:**
>
> Benutzen Sie für diese Collection das Attribut `region`

H3.5:

Implementieren Sie die Methode `public int compareTo(Region.Node o)`. Sie soll im Attribut `location` die Methode `compareTo`, mit der Location von `o` übergeben, aufrufen und diesen Wert zurückgeben.

H3.6:

Implementieren Sie die Methode `public boolean equals(Object o)`. Diese Methode vergleicht das übergebene Objekt mit dem aktuellen Objekt. Wenn beide gleich sind, soll `true` zurückgegeben werden, falls `o` `null` oder

nicht vom Datentyp `NodeImpl` ist, soll `false` zurückgegeben werden. Anderenfalls soll `Objects.equals` mit `name`, `node.name` und `Object.equals` mit `location` und `node.location` und `Objects.equals` mit `connections` und `node.connections` aufgerufen und zurückgegeben werden.

H3.7:

Implementieren Sie die Methode `public int hashCode()`. Sie soll den Hashcode von `name`, `location` und `connections` generieren und zurückgeben.

> **Hinweis:**
>
> In Java können Hash-Codes durch den Aufruf von `Objects.hash(...)` erzeugt werden.

H3.8:

Implementieren Sie die Funktion `public String toString()`, welche einen String zurückgeben soll, der Namen, Location und Connections in einem sinnvollem Format enthält.

H4: Routing - Kantige Angelegenheit *Interface Edge*

> Für die hier durchzuführenden Implementierungen existieren public-Tests in `test/EdgeTest.java`. Nutzen Sie die Tests, um Ihre Implementierungen auf korrekte Funktionsweise zu testen.

Damit in der Region auch Straßenverbindungen modelliert werden können, müssen Sie im letzten Schritt noch das Interface `Edge` als `class EdgeImpl implements Region.Edge` implementieren.

Wir geben Ihnen wieder die Klasse `EdgeImpl` inklusive Attribute, Konstruktor und einiger Standardmethoden vor.

H4.1:

Implementieren Sie die Methoden `public Region getRegion()`, `public Duration getDuration()`, `public Location getLocationA()`, `public Location getLocationB()` und `public String getName()`. Sie sollen ihr jeweils zugehöriges Attribut zurückgeben.

H4.2:

Implementieren Sie die Funktionen `public Region.Node getNodeA()` und `public Region.Node getNodeB()`. Sie sollen den Knoten, der zur entsprechenden Location (`locationA` oder `locationB`) der Region gehört, zurückgeben.

H4.3:

Implementieren Sie die Funktion `public int compareTo(Region.Edge o)`. Diese soll die aktuelle Edge mit der übergebenen Edge `o` vergleichen und als Integer das Vergleichsergebnis zurückgeben. Nutzen Sie einen `Comparator`.

H4.4:

Implementieren Sie die Funktion `public boolean equals(Object o)`. Sie soll prüfen, ob das übergebene Objekt diesem Objekt entspricht. Bei Gleichheit soll true zurückgegeben werden, falls das Übergebene Objekt `null` ist oder einen anderen Datentyp als `EdgeImpl` hat, soll false zurückgegeben werden. Anderenfalls soll `Objects.equals` auf `name`, `edge.name` und `Objects.equals` auf `location A`, `edge.locationA` und `Objects.equals` auf `locationB`, `edge.locationB` und `Objects.equals` auf `duration`, `edge.duration` aufgerufen und zurückgegeben werden.

H4.5:

Implementieren Sie die Funktion `public int hashCode()`, welche den Hashcode von `name`, `locationA`, `locationB` und `duration` erstellen und zurückgeben soll.

H4.6:

Implementieren Sie die Methode `public String toString()`, welche einen String zurückgeben soll, der Namen, beide Locations und die Dauer in einem sinnvollen Format enthält.

H5: Hab mein Wage, voll gelade...

> **Hinweis:**
>
> Für die hier durchzuführenden Implementierungen existieren public-Tests in `test/VehicleTest.java`. Nutzen Sie die Tests, um Ihre Implementierungen auf korrekte Funktionsweise zu testen.

Um die Lieferungen zu den Kund:innen zu bringen werden Fahrzeuge benötigt. Hierfür stellen wir Ihnen das Interface `Vehicle` zur Verfügung. Sie implementieren das Interface in der Klasse `VehicleImpl`. Ein Fahrzeug weiß, wo es sich gerade befindet - auf einer bestimmten Straße oder auf einem bestimmten Knoten. Außerdem lässt sich jedes Fahrzeug durch eine eindeutige Identifikationsnummer identifizieren. Die maximale Zuladung des Fahrzeugs in Kilogramm wird ebenso wie das geladene Essen im Fahrzeug gespeichert.

H5.1: Das Zünglein an der Wage

Implementieren Sie die Methode `public double getCurrentWeight()` im Interface `Vehicle`, die das aktuelle Gewicht der Zuladung des Fahrzeugs zurückgibt, indem das Gewicht der einzelnen geladenen Essen aufsummiert wird. Wichtig: Implementieren Sie diese Methode nicht in der Klasse `VehicleImpl`!

H5.2: Bestellungen ein- und ausladen

Implementieren Sie die rückgabelose Methode `public void loadOrder(ConfirmedOrder order)`, die eine Bestellung auf das Fahrzeug läd. Wenn das Fahrzeug schon voll beladen ist oder das Fahrzeug mit dem übergebenen Essen die maximale Zuladung überschreitet soll eine `VehicleOverloadedException` geworfen werden. Außerdem soll geprüft werden, ob das Essen, welches auf das Fahrzeug geladen werden soll, überhaupt kompatibel bzw. ladefähig ist. Wenn nicht, so soll eine `FoodNotSupportedException` geworfen werden.

Implementieren Sie außerdem die rückgabelose Methode `public void unloadOrder(ConfirmedOrder order)`, welche die übergebene Bestellung aus dem Attribut `orders` löscht.

H5.3: Ein Weg nach vorn

Um unsere Pizza-Auslieferungsfahrzeuge möglichst effektiv zu verwalten, versorgen wir sie, neben den auszuliefernden Bestellungen, auch noch mit den Wegbeschreibungen, wohin sie die Bestellungen liefern sollen. Um diesen Vorgang so zu gestalten, dass ein Fahrzeug nacheinander mehrere Bestellungen ausfahren kann, müssen die Routenanweisungen, die an das Fahrzeug gegeben werden, in einer Warteschlange gespeichert werden. Wenn ein Vehicle dann an seinem Zielort angekommen ist, wird die nächste Route abgefahren.

Um dies zu modellieren, implementieren Sie die Methode `public void moveQueued(Region.Node node, Consumer<? super Vehicle> arrivalAction)`.

Zur Fehlervermeidung sollten Sie ganz zu Beginn Ihrer Implementierung prüfen, ob der Knoten, der an `moveQueued()` übergeben wird, dem Knoten entspricht, auf dem sich das Fahrzeug gerade befindet. Falls der Übergebene Knoten der Knoten ist, auf dem sich das Fahrzeug aktuell befindet, soll eine `IllegalArgumentException` geworfen werden.

Weiterhin soll die Methode dem Attribut `moveQueue` ein neues Objekt vom Typ `PathImpl` hinzufügen. Dieses Objekt stellt die Route dar, die zu der Queue des Fahrzeugs hinzugefügt wird. Um eine Routenführung zu gewährleisten, soll als Routenstartpunkt der Letzte Knoten der letzten Route in der Queue verwendet werden. Somit kann das Auto direkt nach Abarbeitung der letzten Route mit der neuen Route weitermachen. Nutzen Sie zur weiteren Routenberechnung diesen letzten Knoten der letzten Route und berechnen Sie mit dem `PathCalculator` die Route.
Fügen Sie dann die berechnete Route zur Queue hinzu.

H5.4: Auf anderen Wegen

In der Methode `public void moveDirect(Region.Node node, Consumer<? super Vehicle> arrivalAction)` soll zunächst die Warteschlange, also `moveQueue`, geleert werden. Falls der Übergebene Knoten der Knoten ist, auf dem sich das Fahrzeug aktuell befindet, soll eine `IllegalArgumentException` geworfen werden. Im anderen Fall wird `moveQueued()` mit dem Knoten der Parameterübergabe (also `node`) und der `arrivalAction` aufgerufen.

Im Fall, dass sich das Fahrzeug aktuell auf keinem Knoten, sondern eine Kante befindet, soll sich das Fahrzeug zunächst bis zum nächsten Knoten bewegen, wofür `moveQueued` entsprechend aufgerufen wird. Der nächste Knoten wird dann als Startknoten für den eigentlichen Aufruf für `moveQueue` verwendet.

H6: Wo ist eigentlich mein Auto?

> **Hinweis:**
>
> Für die hier durchzuführenden Implementierungen existieren public-Tests in `test/VehicleManagerTest.java`. Nutzen Sie die Tests, um Ihre Implementierungen auf korrekte Funktionsweise zu testen.

Damit sichergestellt ist, dass die Software jederzeit weiß, wo in der Region sich welches Fahrzeug befindet, stellen wir Ihnen das Interface `VehicleManager` zur Verfügung. Im Vehicle-Manager sind die Region, ein Entfernungsberechner und ein Set von Fahrzeugen gespeichert.

Die folgenden Aufgaben werden in der Implementation von `VehicleManger`, `VehicleManagerImpl` im Package `projekt.delivery.routing`, umgesetzt.

H6.1:

`toOccupiedNodes` kriegt eine Sammlung von Nodes, die auf Objekte vom Typ `OccupiedNodeImpl` abgebildet werden und in einer *nicht* modifizierbaren `Map` gespeichert werden sollen, welche am Ende zurückgegeben wird. Handelt es sich bei dem Knoten um eine Komponente des in `warehouse` gespeicherten Warenhauses, so wird der Knoten einfach auf `warehouse` gemappt. Ist das nicht der Fall, aber der Knoten ist ein Subtyp von `Region.Neighborhood`, wird dieser in einem `OccupiedNeighborhoodImpl`-Objekt eingekapselt. Sind beide Bedingungen nicht erfüllt, wird der Knoten in einem neuen `OccupiedNodeImpl`-Objekt gespeichert.

Ähnlich funktioniert die Methode für Kanten, `toOccupiedEdges`. Hier wird allerdings jede Kante immer auf ein Objekt vom Typ `OccupiedEdgeImpl` abgebildet und diese Zuordnungen wiederum in eine unveränderbaren Map gespeichert.

H6.2:

Die Methode `getAllOccupied`, ebenfalls in `VehicleManagerImpl`, soll eine *ummodifizierbare* Menge mit allen in den Attributen `occupiedNodes` und `occupiedEdges` gespeicherten Werten zurückgeben.

H6.3:

Die Methode `getOccupied(C)` bekommt eine Komponente der `Region` und soll für diese die `Occupied`-Variante zurückgeben. Hierfür darf der aktuale Wert des Parameters nicht `null` sein. Falls er es doch ist, soll eine `NullPointerException` mit der Nachricht `"component"` geworfen werden. Handelt es sich bei der übergebenen Komponente weder um ein Objekt vom Typ `Region.Node`, noch um eines vom Typ `Region.Edge`, wird eine `IllegalArgumentException` mit der Botschaft `"Component is not of recognized subtype: "` plus den Klassennamen des Parameters geworfen.

Treten diese beiden Fälle nicht ein, der aktuale Parameter ist also entweder vom Typ `Region.Node` oder `Region.Edge`, muss noch geprüft werden, ob für diesen Knoten bzw. diese Kante ein Wert in der `occupiedNodes`- bzw. `occupiedEdges`-Map existiert. Existiert ein solcher Wert, wir dieser einfach zurückgegeben. Sollte kein solcher Wert in der Map vorhanden sein, wird eine `IllegalArgumentException` mit der Nachricht `"Could not find occupied {1} for " + component`. Der Substring `"{1}"` soll passend durch `"node"` bzw. `"edge"` ersetzt werden.

H6.4:

`getOccupiedNeighborhood` funktioniert ähnlich zu `getOccupied(C)` aus der vorherigen Aufgabe. Auch hier muss wieder geprüft werden, ob der aktuale Parameter `null` ist und ggf. eine `NullPointerException` mit der Nachricht `"component"` geworfen werden. Wenn `occupiedNodes` keinen entsprechenden Schlüsselwert hat, oder der mit `component` assoziierte Wert kein Subtyp von `OccupiedNeighborhood` ist, soll eine `IllegalArgumentException` mit der Nachricht `"Component " + component + " is not a neighborhood"` geworfen werden. Ansonsten soll einfach der in der Map zugeordnete Wert zurückgegeben werden.

H7: Einmal Lieferdienst zum Mitnehmen, bitte!

H7.1: BogoDeliveryService und BasicDeliveryService

H7.2: Rater

Das Gerüst für die Modellierung unseres Lieferdienstes steht also. Damit wir einen Lieferdienst simulieren können, müssen Sie das Interface `DeliveryService` implementieren. Wir geben Ihnen die Klassen `AbstractDeliveryService`, `FopEx extends AbstractDeliveryService` und `BogoDeliveryService extends AbstractDeliveryService` vor.
Ihre Aufgabe besteht darin, in den folgenden Teilaufgaben die Implementierungen zu vervollständigen, indem Sie in `FopEx` eine Implementierung von `DeliveryService` erstellen, welche besser als `BogoDeliveryService` ist.
In der Klasse `FopEx` befindet sich ein `VehicleManager vehicleManager`, der es Ihnen ermöglicht, die Region mit den Fahrzeugen und deren Positionen zu verwalten. Außerdem enthält die Klasse ein Attribut `Rater rater`, welcher die Lösungen für Lieferprobleme bewertet. Für die eigentliche Simulation benötigen Sie noch die Attribute `Simulation simulation` und `SimulationConfig simulationConfig`.

H7.3:

Implementieren Sie in Klasse `FopEx` die Methode `void tick(List<ConfirmedOrder> newOrders)`. In dieser Methode soll auf neue Orders reagiert werden. Die Orders sollen auf die wartenden Fahrzeuge verteilt und ausgeliefert werden. Nach einer Lieferung soll das Fahrzeug wieder zur Pizzeria zurückgeschickt werden.
Als Beispiel können Sie sich die eher ineffiziente Implementierung `BogoDeliveryService` ansehen.

Verbindliche Anforderungen:

Sei $N > 0$ die Anzahl der Zielorte in Attribut `destinations` und $K > 0$ die Anzahl der Auslieferungsfahrzeuge. Dann muss `a` die folgenden Bedingungen erfüllen:

1. Jedes Fahrzeug bekommt eine Sequenz von Zielorten zum Anfahren (die auch leer sein kann), also:

   Es ist `a.length ==` K und `a[`i`] != null` für alle $i \in \{0, \ldots, K-1\}$, aber `a[`i`].length ==` 0 ist möglich.

2. Jeder Zielort wird von maximal einem Fahrzeug und von diesem auch nur einmal angefahren, also:

   Jeder Wert $0, \ldots, N-1$ kommt in maximal einem `a[`i`]` ($i \in \{0, \ldots, K-1\}$) vor und darin auch nur genau einmal. Andere `int`-Werte kommen in der Rückgabe nicht vor.[a]

3. Bei keinem Fahrzeug wird die Ladekapazität überschritten, also:

   Für $i \in \{0, \ldots, K-1\}$ sei C_i die Kapazität von Fahrzeug i und für $\ell \in \{0, \ldots,$`a[`i`].length`$-1\}$ sei B^i_ℓ der Bedarf am Zielort mit Nummer `a[`i`][`j`]`. Dann muss gelten:

   $$\sum_{j=0}^{\texttt{a[i].length}-1} B^i_j \;\leq\; C_i\,.$$

4. Das Lager ist am Ursprung des Koordinatensystems, also am Punkt $(0, 0)$. Jedes Fahrzeug $i \in \{0, \ldots, K-1\}$ fährt am Ursprung los und fährt die Zielorte `a[`0`]`$, \ldots,$`a[a[`i`].length`-1`]` in dieser Reihenfolge an. Wir gehen davon aus, dass die Zeitdauer des Fahrzeugs an jedem Zielort gleich 0 ist. Der Zeitpunkt, zu dem das Fahrzeug einen Zielort anfährt, darf aber nicht vor Beginn des Zeitfensters dieses Zielortes sein; gegebenenfalls überbrückt das Fahrzeug die Zeit bis zum Beginn des Zeitfensters mit Warten. Ansonsten sind keine Wartezeiten oder Verzögerungen in unserem einfachen Modell vorgesehen, ohne diese absichtlichen Wartezeiten würde das Fahrzeug also einfach von einem Ort zum nächsten in exakt dem Zeitraum fahren, den das Distanzattribut für diese beiden Orte angibt (*Erinnerung*: Distanzangaben werden als Zeiten interpretiert).

   Mathematisch heißt das, dass folgende zwei Bedingungen für jedes $i \in \{0, \ldots, K-1\}$ erfüllt sein müssen. Für $\ell \in \{0, \ldots,$`a[`i`].length`$-1\}$ sei S^i_ℓ der Anfang des Zeitfensters am Zielort Nummer `a[`i`][`ℓ`]` und T^i_ℓ der (natürlich nicht vorgegebene, sondern zu berechnende) Zeitpunkt, zu dem dieser Zielort angefahren wird. Dann muss gelten:

   a) $T^i_\ell \geq S^i_\ell$ für jedes $\ell \in \{0, \ldots,$`a[`i`].length`$-1\}$ und

   b) $T^i_\ell - T^i_{\ell-1} \geq D($`a[`$i$`][`$\ell-1$`],a[`$i$`][`$\ell$`]`$)$ für jedes $\ell \in \{1, \ldots,$`a[`i`].length`$-1\}$.

[a] Die einzelnen `int`-Arrays bilden also eine Partition der Zielorte, siehe auch `https://de.wikipedia.org/wiki/Partition_(Mengenlehre)`.

H8: La pizzeria de FOP

Attribut	`name`	`menu`

`LOS_FOPBOTS_HERMANOS`	`"Los Fopbots Hermanos"`	`Pizza.Margherita`
		`Pasta.SPAGHETTI`
		`Pasta.RIGATONI`

`JAVA_HUT`	`"Java Hut"`	`Pizza.Margherita`
		`Pasta.SPAGHETTI`
		`Pasta.RIGATONI`

`PASTAFAR`	`"Pastafar"`	`Pasta.SPAGHETTI` `Pasta.RIGATONI`

`PALPAPIZZA`	`"Palpapizza"`	`Pizza.Margherita`
		`Pasta.SPAGHETTI`
		`Pasta.RIGATONI`

`ISENJAR`	`"Isenjar"`	`Pizza.Margherita`
		`Pasta.SPAGHETTI`
		`Pasta.RIGATONI`

`MIDDLE_FOP`	`"Middle Fop"`	`Pizza.Margherita`
		`Pasta.SPAGHETTI`
		`Pasta.RIGATONI`

`MOUNT_DOOM_PIZZA`	`"Mount Doom Pizza"`	`Pizza.Margherita`
		`Pasta.SPAGHETTI`
		`Pasta.RIGATONI`

Tabelle 1: Pizzerias

H9: Die GUI

Dokumentation:

Sie listen alle der obigen Funktionalitäten, die Sie realisiert haben, in einer PDF-Datei auf. Zu jeder dieser Funktionalitäten beschreiben Sie, wie man sie in Ihrer GUI ansteuern und benutzen kann, und zwar so präzise, dass es bei der Bewertung leicht ist, Ihre Realisierung nachzuvollziehen.

Verbindliche Anforderungen:

1. Die Anforderungen aus dem Leitfaden GUI werden strikt eingehalten.
2. Das MVC-Pattern aus Kapitel 14 wird strikt umgesetzt.
3. Nur die Module `java.base` und `java.desktop` dürfen verwendet werden, keine weitere Funktionalität aus der Java-Standardbibliothek oder aus anderen Bibliotheken.
4. Optionen, die momentan nicht sinnvoll sind (z.B. eine Probleminstanz hinausschreiben, wenn momentan keine da ist), werden nicht angeboten, also entweder gar nicht gezeigt oder nur ausgegraut (und ohne Reaktion) angezeigt.

H9.1: Grundfunktionalitäten der GUI

Schreiben Sie für das Programm eine grafische Benutzeroberfläche, die es erlaubt, interaktiv mit den Lieferungen umzugehen. Sie soll mindestens folgende Funktionalitäten bieten:

1. Die Geschwindigkeit, mit der ein Tick ausgeführt wird, soll mittels eines Sliders auswählbar sein.
2. Eine Karte anzeigen, die die Region und den momentanen Zustand der Lieferungen und Fahrzeuge abbildet:
   a) Der Koordinatenursprung $(0, 0)$ für Knoten bzw. `Location` ist dabei in der Mitte der Zeichenfläche.
   b) Knoten werden so in der Karte platziert, dass ihre `Location` der Offset von diesem Ursprung aus ist.
   c) Verbindungen zwischen zwei Knoten werden als gerade Linie zwischen ihren Positionen gezeichnet.
   d) Die Zoomstufe kann durch scrollen verändert werden. (Möglichst mit maximalem und minimalem Zoom)
3. Einen Dialog anzeigen, in den der Nutzer Daten eingeben kann, um während der Simulation Bestellungen aufzugeben.

Ihre GUI enthält geeignete Komponenten (Buttons, Menüs usw.), um diese Optionen anzuwählen. Sie benutzen `BorderLayout`, um die Darstellungsfläche für Probleminstanzen und Lösungen im Bereich `CENTER` und andere Komponenten wie Buttons und Menüs in anderen Bereichen als `CENTER` zu platzieren.

H9.2: Erweiterungsmöglichkeiten für die GUI

Darüber hinaus bringen folgende Funktionalitäten weitere Punkte:

1. Erweiterung der Karte, Fahrzeugpositionen in Echtzeit anzeigen und aktualisieren:
   a) Fahrzeuge, die gerade eine Lieferung ausführen, sollen auf den jeweiligen Knoten angezeigt werden, wenn sie vor Ort sind, oder an der entsprechenden Position[2] auf der Verbindung, wenn sie gerade von einem Knoten zum nächsten fahren.
   b) Die Positionen der Fahrzeuge sollen sich mit jedem Tick aktualisieren.
2. eine Editiermöglichkeit von Probleminstanzen in der GUI, also die Möglichkeit existierende Probleminstanzen direkt in der GUI zu ändern:
   a) einen existierenden Knoten mit Drag & Drop verschieben, alle Daten des Zielortes außer den Koordinaten bleiben gleich
   b) Änderung der Daten eines existierenden Punktes über ein Popup-Fenster
   c) Verschiebung eines Zielortes aus der Tour eines Fahrzeugs in die Tour eines anderen Fahrzeugs
3. *„Intelligentes" Editieren*: bei der Erstellung oder Verschiebung eines Knotens wird ein Fahrzeug und ein geeignetes Zeitfenster angeboten.
4. Ansicht der Karte verändern:
   a) Hinein- und Herauszoomen mit Buttons $+/-$
   b) (vor allem im hineingezoomten Zustand) den im Fenster gezeigten Ausschnitt durch Mausaktionen zu verschieben
   c) die Ergebnisse mehrerer Algorithmen parallel neben- oder übereinander anzeigen

[2] „An der entsprechenden Position" heißt hierbei, dass wenn ein Fahrzeug 20% der Strecke hinter sich hat, dieses auch in der Karte an 20% der Distanz zwischen den beiden Knoten eingezeichnet werden soll.