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: 5 Punkte**

Lieferservice Simulation

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.

Einleitung

Sie wurden von einem bekannten Essenslieferdienst dazu aufgetragen ein Programm zu entwickeln, welches es einem ermöglicht den Ablauf eines Lieferdienst zu simulieren und am Ende zu bewerten. Vervollständigen Sie dafür anhand der folgenden Aufgaben die Vorlage, welche wir Ihnen zu Verfügung stellen. Die Vorlage ist dabei in drei folgenden Teile aufgeteilt:

- **Application**: Von hier aus wird das eigentliche Programm gestartet.
- **Domain**: Hier befindet sich die Umsetzung des grundlegenden Problems. Bis auf Aufgabe H12 werden Sie Ihre Lösungen hier implementieren.
- **Infrastructure**: Hier befindet sich die Kommunikation mit der „Außenwelt“, also die Umsetzung der GUI und IO Operationen.

Die Domain ist dabei Schichtenweise aufgebaut. Die unterste Schicht wird von dem Interfaces `Region` (siehe H2) dargestellt und beschreibt den Aufbau des Liefergebietes. Die nächste Schicht wird von dem Interface `VehicleManager` (siehe H6) umgesetzt und ist dafür zuständig die verschiedenen Fahrzeuge des Lieferservices zu verwalten. Auf der dritten Schicht werden eingehende Bestellungen angenommen, verwaltet und den einzelnen Fahrzeugen zugewiesen. Dies wird von dem Interface `DeliveryService` (siehe H9) dargestellt. Auf der letzten Schicht befindet sich das Interface `Simulation`, welches den zeitlichen Ablauf der Simulation verwaltet.

Die Simulation basiert dabei auf einem Tick Prinzip. Es wird also konstant ein Tick Zähler hochgezählt und bei jeder Erhöhung ein Simulationsschritt ausgeführt. Ein solcher Simulationsschritt entspricht dabei z.B. der Bewegung eines Fahrzeuges, das Aufnehmen einer Lieferung, etc. Konkret wird dabei jedes mal von oben die `tick(long)` Methode der einzelnen Schichten aufgerufen, welche daraufhin alle in den nächsten Zustand übergehen. Jeder Simulationsschritt der dabei ausgeführt wird, erzeugt dabei ein `Event`. Eine Liste aller erzeugten `Events` wird am Ende von den Tick Methoden zurückgegeben. Diese Events werden am Ende der Simulation von dem `Rater` Interface (siehe H8) verwendet um die Simulation hinsichtlich bestimmter Kriterien zu bewerten.

Die einzelnen Bestellungen werden in der Simulation von der Klasse `ConfirmedOrder` dargestellt. Diese besitzt die folgenden Eigenschaften:

Attributname	Typ	Beschreibung
`location`	`Location`	Die Koordinaten des Zielorts.
`orderID`	`int`	Die ID der Bestellung.
`tickIntervall`	`TickIntervall`	Der Zeitraum, in dem die Lieferung erfolgen soll.
`actualDeliveryTime`	`long`	Der Zeitpunkt, an dem die Lieferung tatsächlich erfolgt ist.
`foodList`	`List<String>`	Die eigentliche Bestellung; eine Liste von Gerichten, die geliefert werden sollen.
`weight`	`double`	Das Gewicht der Bestellung

Was dabei genau Simuliert werden soll, wird von dem Interface `ProblemArchetype` beschrieben. Dieses besteht aus folgenden Komponenten:

- **VehicleManager**: Beschreibt den Aufbau der zugrundeliegenden Region und die verfügbaren Fahrzeugen.
- **OrderGeneratorFactory**: Beschreibt die Bestellungen, welche bei dem Lieferservice eingehen. (siehe H7)
- **SimulationLength**: Beschreibt, wie lange simuliert werden soll, also wie lange Bestellungen ausgeliefert werden.

Über dem Interface `ProblemArchetype` liegt zusätzlich die Klasse `ProblemGroup`, welche mehrere dieser Probleme zusammenfasst und festlegt welche Bewertungskriterien wie bewertet werden sollen.

Letztendlich gibt es noch das Interface `Runner` (siehe H10), welches eine Instanz der Klasse `ProblemGroup`, eine `SimulationConfig`, sowie eine Anzahl an auszuführenden Simulation bekommt. Diese ist dafür zuständig für jedes `ProblemArchetype` aus der `ProblemGroup` eine Simulation zu erstellen, diese so oft durchzuführen, wie angegeben, und am Ende für jedes Bewertungskriterium die durchschnittliche Punktzahl zu berechnen.

H1: Location — 5 Punkte

Vervollständigen Sie zunächst in den folgenden Aufgaben die Klasse `public final class Location` im Package `projekt.base`.

H1.1: — 2 Punkte

Implementieren Sie die Methode `public int compareTo(Location o)` der Klasse `Location`, 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.

H1.2: — 1 Punkt

Implementieren Sie die Methode `public int hashCode()` der Klasse `Location`.

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 diese Vorgabe umsetzen.

H1.3: — 1 Punkt

Implementieren Sie die Funktion `public boolean equals(Object o)` der Klasse `Location`, 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.

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

> **Hinweis:**
>
> Zur Überprüfung, ob beide Objekte vom selben Typ sind, reicht es nicht aus zu überprüfen, ob `o instanceof Location`, da diese Abfrage auch `true` zurückgibt, wenn o ein Subtyp von `Location` ist. Überprüfen Sie stattdessen, ob die Methode `getClass()` aufgerufen auf das momentane Objekt und dem übergebenem Objekt den selben Wert zurückgibt.

H1.4: **1 Punkt**

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

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

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 Restaurant, 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 drei Arten von Knoten: Die Nachbarschaften, Restaurantknoten 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. Restaurantknoten stellen die einzelnen Restaurants dar bei denen Bestellungen aufgegeben werden und später von einem Fahrzeug abgeholt werden und zu dem entsprechenden Lieferort gefahren werden. 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` in der Klasse `RegionImpl`. Die Klassendatei hierfür stellen wir Ihnen inklusive einiger Basismethoden und den benötigten Attributen bereits zur Verfügung.

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 zwei Locations einer Kante zugewiesen. Dabei ist die, laut der `compareTo` Methode kleinere Location, in der äußeren Map enthalten und größere Location in der inneren Map.

> **Erinnerung:**
>
> Basisinformationen zur Arbeit mit Maps finden Sie in Foliensatz 07.

Anmerkung:

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` arbeiten.

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` mit der Botschaft `"Node <node> has incorrect region"` geworfen werden, wobei `<node>` mit der String Repräsentation der übergebenen Node ersetzt werden soll.

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, oder einer ihrer Kanten nicht in dieser Region (`this`) liegt, soll eine `IllegalArgumentException` mit der Botschaft `"Edge <edge> has incorrect region"` geworfen werden, wobei `<edge>` mit der String Repräsentation der übergebenen Node ersetzt werden soll. Wenn `nodeA`, bzw. `nodeB` nicht in der Region ist, soll `"Edge <edge>"` jeweils mit `"NodeA <nodeA>"`, bzw. `"NodeB <nodeB>"` ausgetauscht werden und `<nodeA>`, bzw. `nodeB`, mit der

Position des Knotens. Wenn mehrere dieser Komponenten nicht in dieser Region enthalten ist, ist es Ihnen überlassen, welche der passenden Botschaften gewählt werden.

H2.5:

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

H2.6:

Implementieren Sie die Methode `public boolean equals(Object o)`, die überprüft, ob das übergebene Objekt (`o`) gleich dem aktuellen Objekt (`this`) ist.

Sie gibt `true` zurück, sollten `o` und `this` dem Gleichheitsoperator (`==`) nach identisch sein oder sollte `Objects.equals()` sowohl für die Werte `this.nodes` und `o.nodes`, als auch für die Werte `this.edges` und `o.edges`, `true` zurückliefern. Falls das übergebene Objekt dagegen `null` oder nicht vom Typ `RegionImpl` ist, oder oben genannte Bedingung nicht erfüllt wird, soll `false` zurückgegeben werden.

> **Hinweis:**
>
> Beachten Sie, dass für den Vergleich `o` von `Object` zu `RegionImpl` gecastet werden muss. `o.{edges,nodes}` ist hier also eine Abstraktion, bei der sich `o` auf das bereits gecastete Objekt bezieht.

H2.7:

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

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

H3: Routing - Knoten ohne Ende *Interface Node*

Um die Region mit Leben zu füllen und z.B. Autobahnkreuze oder Nachbarschaften realisieren zu können benötigen wir einen Typen für unsere Knoten. Implementieren Sie hierzu das Interface `Node` in der Klasse `NodeImpl`.

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.

> **Hinweis:**
>
> Die notwendigen Informationen über den Aufbau können Sie in `region` abfragen.

H3.1:

Implementieren Sie die Methoden `public Region getRegion()`, `public Location getLocation()` und `public String getName()`, die die jeweiligen Attribute zurückgeben sollen.

H3.2:

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

Knoten und der übergebene Knoten nicht verbunden sind, soll stattdessen `null` zurückgegeben werden.

H3.3:

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

H3.4:

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

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 (`this`).

Falls `o` `null` oder nicht vom Datentyp `NodeImpl` ist, gibt `equals` `false` zurück.

Sind dagegen `o` und `this` dem Gleichheitsoperator (`==`) nach identisch oder wird `Objects.equals` mit `this.name`, `o.name` und mit `this.location`, `o.location` sowie mit `this.connections`, `o.connections` aufgerufen und alle drei Vergleiche werten zu `true` aus, so gibt auch die Methode `equals` `true` zurück, ansonsten `false`.

> **Hinweis:**
>
> Beachten Sie, dass für den Vergleich `o` von `Object` zu `NodeImpl` gecastet werden muss. `o.{connections,location,name}` ist hier also eine Abstraktion, bei der sich `o` auf das bereits gecastete Objekt bezieht.

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 von mehreren Objekten, durch den Aufruf von `Objects.hash(...)`, erzeugt werden.

H3.8:

Implementieren Sie die Funktion `public String toString()`, welche den String `"NodeImpl (name='<name>', location='<location>', connections='<connections>')"`, wobei die Substrings umgeben von < und > mit der String-Repräsentation des jeweiligen Attributes ersetzt werden.

> **Hinweis:**
>
> Achten Sie darauf, dass die Hochkommas (`'`) im String **enthalten** sein sollen, und nicht ersetzt werden.

H4: Routing - Kantige Angelegenheit *Interface Edge*

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

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, oder `null`, wenn kein zugehöriger Knoten existiert.

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. Erstellen Sie dafür zwei Comparator, von denen einer die Attribute `nodeA` und einer die Attribute `nodeB` vergleicht. Kombinieren Sie danach diese beiden Comparator mit der `default` Methode `thenComparing` des Interfaces `Comparators`, wobei zuerst der Comparator, der die `nodeA` Attribute vergleicht, benutzt wird.

H4.4:

Implementieren Sie die Funktion `public boolean equals(Object o)`. Sie soll prüfen, ob das übergebene Objekt diesem Objekt entspricht. Bei Gleicheit nach Gleichheitsoperator (`==`) 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 `locationA`, `edge.locationA` und `Objects.equals` auf `locationB`, `edge.locationB` und `Objects.equals` auf `duration`, `edge.duration` aufgerufen und per `&&` verknüpft zurückgegeben werden.

H4.5:

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

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

H4.6:

Implementieren Sie die Methode `public String toString()`. Diese hat als Rückgabewert den String `"EdgeImpl (name='<name>', locationA='<locationA>', locationB='<locationB>', duration='<duration>')"` wobei wie üblich `"<...>"` durch die passenden Attribute ersetzt werden soll.

> **Hinweis:**
>
> Achten Sie darauf, dass die Hochkommas (`'`) im String **enthalten** sein sollen, und nicht ersetzt werden.

H5: Hab mein Wage, voll gelade...

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.

> **Verbindliche Anforderung:**
>
> Implementieren Sie diese Methode nicht in der Klasse `VehicleImpl`, sondern als `default` Methode in dem Interface `Vehicle`!

H5.2: Bestellungen ein- und ausladen

Implementieren Sie die rückgabelose Methode `public void loadOrder(ConfirmedOrder order)`, die eine Bestellung auf das Fahrzeug lädt, indem Sie die Bestellung zu der `orders` Liste hinzufügt. Wenn das Fahrzeug schon voll beladen ist, oder das Fahrzeug mit dem übergebenen Essen die maximale Zuladung überschreitet, soll eine `VehicleOverloadedException` 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 und sich kein anderer Knoten in der Queue befindet. Falls dies der Fall ist, soll eine `IllegalArgumentException` geworfen werden.

Falls kein Fehler geworfen wurde, 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 Berechnung der Route von diesem letzten Knoten der letzten Route und dem übergebenen Knoten, die Methode `vehicleManager.getPathCalculator().getPath(Node, Node)`.

Fügen Sie dann die berechnete Route zur Queue hinzu, indem Sie mit der eben berechneten Route und dem im Parameter `arrivalAction` übergebenem Objekt ein Objekt der Klasse `PathImpl` erzeugen.

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?

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 ist die Region und ein Set von Fahrzeugen gespeichert.

Der `VehicleManager` benutzt dabei die Klasse `AbstractOccupied` und deren Subklassen um Knoten, auf denen sich Fahrzeuge befinden können, darzustellen. Dabei wird Aggregation verwendet, die Klasse `OccupiedNode` hat also eine „has-a“ Beziehung zu der Klasse `Node`. Die eigentliche Komponente, also der Knoten, oder die Kante, auf die sich ein Objekt der Klasse `AbstractOccupied` bezieht, lässt sich mit der Methode `getComponent()` abfragen.

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

H6.1:

Implementieren Sie die Methode `toOccupiedNodes`. Diese kriegt eine Sammlung von Nodes, die auf Objekte vom Typ `OccupiedNodeImpl` abgebildet werden sollen. Diese Abbildung soll in einer *nicht* modifizierbaren `Map` gespeichert werden, welche am Ende zurückgegeben wird. Handelt es sich bei einem Knoten um einen Subtyp der Klasse `Region.Restaurant`, so wird der Knoten auf ein neues `OccupiedRestaurantImpl` Objekt abgebildet. 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. Die Konstruktoren der Klassen `Occupied*` kriegen zusätzlich zu der eingekapselten Komponenten den momentanen `VehicleManager` übergeben.

Implementieren Sie ebenfalls die ähnlich funktionierende Methode für Kanten, `toOccupiedEdges`. Hier wird allerdings jede Kante immer auf ein Objekt vom Typ `OccupiedEdgeImpl` abgebildet.

H6.2:

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

H6.3:

Die Methode `getOccupied` bekommt eine Komponente der `Region` übergeben 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 is null!"` 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: <subtype>"` geworfen, wobei `<subtype>` mit dem Klassennamen des Parameters ersetzt wird.

> **Hinweis:**
>
> Den Klassennamen von einem Objekt können Sie über die Methode `getClass().getName()` abfragen.

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 <type> for <component>"`. Der Substring `<type>` soll passend durch `"node"` bzw. `"edge"` ersetzt werden und `<component>` durch die String-Repräsentation der übergebenen Komponente ersetzt werden.

H6.4:

Vervollständigen Sie zum Schluss die Methode `getOccupiedNeighborhood`, welche ähnlich zur Methode `getOccupied` aus der vorherigen Aufgabe funktioniert. Auch hier muss wieder geprüft werden, ob der aktuale Parameter `null` ist und ggf. eine `NullPointerException` mit der Nachricht `"Component is null!"` geworfen werden. Wenn `occupiedNodes` keinen entsprechenden Schlüsselwert hat, oder der mit dem Parameter assoziierte Wert kein Subtyp von `OccupiedNeighborhood` ist, soll eine `IllegalArgumentException` mit der Nachricht `"Component <component> is not a neighborhood"` geworfen werden, wobei `<component>` mit der String-Repräsentation des aktualen Parameters ersetzt wird. Ansonsten soll einfach der in der Map zugeordnete Wert zurückgegeben werden.

Vervollständigen Sie mit der selben Logik - natürlich angepasst - die Methode `getOccupiedRestaurant`.

H7: Was gibt es heute zu Essen?

Als nächstes wollen wir die Möglichkeit haben automatisch zufällige Bestellungen zu generieren. Dazu finden Sie im Package `projekt.delivery.archetype` das Interface `OrderGenerator`, welches die Methode `generateOrders(long)` mit Rückgabetyp `List<ConfirmedOrder>` definiert. Die Methode `generateOrders(long)` generiert dabei die Bestellungen, die bei dem übergebenen Tick aufgegeben werden.

H7.1: Ein typischer Freitagabend

Implementieren Sie nun die Klasse `FridayOrderGenerator`, welche das Interface `OrderGenerator` implementiert. Diese soll deterministisch die Bestellungen eines typischen Freitagabends darstellen. Konkret soll die im Konstruktorparameter `orderCount` übergebene Anzahl an Bestellungen normalverteilt auf die ticks 0 bis `lastTick` verteilt werden. `lastTick` ist dabei ebenfalls einer der Konstruktorparameter. Zwei Aufrufe der Methode `generateOrders(long)` sollen dabei immer die selben Bestellungen zurückgeben, wenn der selbe Wert übergeben wird. Falls `generateOrders` mit einer negativen Zahl aufgerufen wird, soll eine `IndexOutOfBoundsException` geworfen werden, welche im Konstruktor die negative Zahl übergeben bekommt.

Eine normalverteilte Zufallszahl können sie mit `random.nextGaussian(double, double)` erzeugen. Der erste Parameter der Methode ist der Erwartungswert, der zweite die Varianz. Für die Varianz wird dem Konstruktor bereits ein Wert übergeben. Für den Erwartungswert können Sie einen eigenen Wert wählen.

> **Hinweis:**
>
> Wenn Sie als Erwartungswert 0.5 angeben, und jedes mal eine neue Zahl generieren, wenn die Methode eine Zahl kleiner 0 oder größer 1 zurückliefert, erhalten Sie normalverteilte Zufallszahlen zwischen 0 und 1.

Die Bestellungen sollen dabei mit folgenden Werten erzeugt werden:

- **Location**: Die Position eines zufälligen Elementes aus der Collection, die von `vehicleManager.getOccupiedNeighborhoods()` zurückgeliefert wird.
- **Restaurant**: Ein zufälliges Element aus der Collection, die von `vehicleManager.getOccupiedRestaurants()` zurückgegeben wird.

- **DeliveryInterval**: Ein Objekt von Typ `TickIntervall`, welches als Startpunkt den Tick, zu dem die Order ausgeliefert wird, hat und als Endpunkt den Startpunkt plus die Tickanzahl, welche im Konstruktorparameter `deliveryDifference` übergeben wird, hat.
- **FoodList**: Eine Liste mit zufälligen Elementen aus den verfügbaren Gerichten des Restaurants. Die Größe der Liste soll dabei ebenfalls zufällig gewählt werden und zwischen 1 (inklusive) und 10 (exklusive) liegen. Die im einen Restaurant des Typen `VehicleManager.OccupiedRestaurant` verfügbaren Gerichte können Sie über `restaurant.getComponent().getAvailableFood()` abfragen.
- **Weight**: Eine zufällige `double` Zahl zwischen 0 und dem Wert des Konstruktorparameters `maxWeight`.

Unbewertete Verständnisfrage:

Wir unterscheiden hier bei den Bestellungen nicht zwischen dem Zeitpunkt, an dem sie bestellt werden, und dem Zeitpunkt, an dem sie ausgeliefert werden können. Denken Sie, dass es sinnvoll wäre dennoch diese Unterscheidung zu treffen?

Verbindliche Anforderung:

Alle Zufallszahlen müssen mit dem bereits vorhanden Attribut von Typ `Random` erzeugt werden.

H8: Habe ich einen guten Job gemacht?

Nachdem eine Simulation durchgeführt wurde, soll es möglich sein, das Ergebnis dieser zu bewerten. Bewertet wird anhand der Kriterien im Enum `RatingCriteria`. Die einzelnen Prüfer, welche die Simulation anhand dieser Kriterien bewerten, werden von dem Interface `Rater` beschrieben. Dieses erweitert das Interface `SimulationListener`. `SimulationListener` definiert die Methode `void onTick(List<Event>, long)`, welche nach jedem Tick der Simulation aufgerufen wird. Der erste Parameter ist eine Liste, welche alle Events enthält, die in diesem Tick geschehen sind und im zweiten Parameter den Tick um den es sich handelt. Zusätzlich dazu definiert das Interface `Rater` die Methode `double getScore()`, welche am Ende der Simulation aufgerufen wird und die Bewertung für die Simulation zurückgibt. Die Bewertung ist dabei eine Zahl im Intervall $[0, 1]$, wobei 1 das beste und 0 das schlechteste Ergebnis ist. Alle Dateien für diese Aufgabe finden Sie im Package `projekt.delivery.rater`.

H8.1: Habe ich alle Bestellungen ausgeliefert?

Vervollständigen Sie die Klasse `AmountDeliveredRater`, welche eine Simulation anhand des Bewertungskriteriums `AMOUNT_DELIVERED`, also ob wie viele Bestellungen tatsächlich ausgeliefert wurden, bewertet. Sie besitzt das Attribut `factor` vom Typ `long`, welches angibt, wie hoch der Anteil der ausgelieferten Bestellungen sein muss, um eine Bewertung größer 0 zu erhalten.

Die Methode `getScore()` berechnet die Bewertung dann wie folgt:

$$\text{score} = \begin{cases} 1 - \frac{\text{undeliveredOrders}}{\text{totalOrders} \cdot (1 - \text{factor})} & \text{undeliveredOrders} > \text{totalOrders} \cdot (1 - \text{factor}) \\ 0 & \text{sonst} \end{cases}$$

Wobei totalOrders die Anzahl an insgesamt aufgegebenen Bestellungen ist und undeliveredOrders die Anzahl an aufgenommen Bestellungen, die aber nicht ausgeliefert wurden, ist. Betrachten Sie dafür in der Methode `onTick` Events vom dynamischen Typ `DeliverOrderEvent`, welche angeben, dass eine Bestellung geliefert wurde und Events vom dynamischen Typ `OrderReceivedEvent`, welche angeben, dass eine Bestellung aufgenommen wurde. Speichern Sie die notwendigen Informationen aus diesen beiden Eventarten in passenden Objektattributen, damit sie diese Werte in der Methode `getScore` verwenden können, um die Bewertung auszurechnen.

H8.2: War ich pünktlich?

Vervollständigen Sie die Klasse `InTimeRater`, welche eine Simulation anhand des Bewertungskriteriums `IN_TIME`, also ob die Bestellungen pünktlich ausgeliefert wurde, bewertet. Sie besitzt die Attribute `ignoredTicksOff` und `maxTicksOff` vom Typ `long`.
`ignoredTicksOff` gibt dabei an, wie hoch die Toleranz bei der Lieferzeit ist. Bei einem Wert von `ignoredTicksOff = 5`, zählt eine Bestellung, die vier Ticks zu spät ausgeliefert wurde, als pünktlich und eine Bestellung, die acht Ticks zu spät ausgeliefert wurde als drei Ticks zu spät.
`maxTicksOff` gibt an, was die maximale Tickanzahl ist, die bei Verspätungen berücksichtigt wird. Bei einem Wert von `maxTicksOff = 25`, zählt eine Bestellung, die 40 oder auch 1000 Ticks zu spät ausgeliefert wurde, als 25 Ticks zu spät.

Die Methode `getScore()` berechnet die Bewertung dann wie folgt:

$$\text{score} = 1 - \frac{\text{actualTotalTicksOff}}{\text{maxTotalTicksOff}}$$

Wobei maxTotalTicksOff die Summe der Verspätungen aller Bestellungen in Ticks ist, wenn alle Bestellungen die maximale Verspätung erreicht hätten. Entsprechend ist actualTotalTicksOff die Summe der tatsächlichen Verspätungen aller Bestellungen in Ticks. Bestellungen, welche aufgenommen wurden, aber noch nicht geliefert wurden zählen dabei so, als ob diese die maximale Verspätung erreicht hätten.

Sie können dafür in der `tick` Methode die selben Events betrachten, wie in H8.1.

H8.3: Wie viel bin ich gefahren?

Vervollständigen Sie die Klasse `TravelDistanceRater`, welche eine Simulation anhand des Bewertungskriteriums `TRAVEL_DISTANCE`, also wie viel weit die einzelnen Fahrzeuge gefahren sind, bewertet. Sie besitzt das Attribut `factor` vom Typ `long`, welches angibt, wie viel der weiter unten berechneten maximalen Strecke, die gefahren werden sollte, tatsächlich gefahren werden darf.

Die Methode `getScore()` berechnet die Bewertung also folgt:

$$\text{score} = \begin{cases} 1 - \frac{\text{actualDistance}}{\text{worstDistance} \cdot \text{factor}} & \text{actualDistance} < \text{worstDistance} \cdot \text{factor} \\ 0 & \text{sonst} \end{cases}$$

Wobei actualDistance insgesamt gefahrene Strecke alle Fahrzeuge ist und worstDistance die Summe der Strecken von dem Restaurant zum Lieferort und zurück für alle ausgelieferten Bestellungen ist. Bestellungen, welche noch nicht geliefert wurden, zählen dabei nicht zu worstDistance hinzu.

Sie können dafür in der `tick` Methode Events vom Typ `OrderReceivedEvent`, wie in H8.1 und H8.2, betrachten, um worstDistance zu berechnen. Um actualDistance zu berechnen, betrachten Sie Events vom Typ `ArrivedAtNodeEvent` und fragen Sie die gefahrene Strecke über `arrivedAtNodeEvent.getLastEdge().getDuration()` ab.

> **Hinweis:**
>
> Über das Attribut `pathCalculator` können Sie mit der Methode `getPath(Region.Node, Region.Node)` den Pfad von einem Knoten zu einem anderen bestimmen. Über das Attribut `region` können Sie mit der Methode `getNode(Location)` den Knoten an einer bestimmten Position erhalten und mit `getEdge(Region.Node, Region.Node)` die Kante zwischen zwei Knoten erhalten.

H9: Einmal Lieferdienst zum Mitnehmen, bitte!

Für den Lieferservice existiert nun die grundlegende Verwaltung für Fahrzeuge und Bestellungen, sowie eine algorithmische Beschreibung für die Problemstellung. Als nächstes müssen Sie noch die Planung der Auslieferungsrouten für die aufgenommenen Bestellungen implementieren. Diese Planung wird von dem Interface `DeliveryService` im Package `projekt.delivery.deliveryService` dargestellt. Für dieses finden Sie in der Klasse `BogoDeliveryService` bereits eine einfache Implementation.

> **Hinweis:**
>
> Alle Klassen, die Sie in dieser Aufgabe implementieren werden, erben von der abstrakten Klasse `AbstractDeliveryService`, welche die meisten Funktionalitäten bereits implementiert. Insbesondere besitzt sie das Attribut `pendingOrders`, welche alle noch nicht abgearbeiteten Bestellungen enthält und von den Subklassen ebenfalls verwaltet werden muss.

H9.1: BasicDeliveryService

Implementieren Sie zunächst in der Klasse `SimpleDeliveryService`, welche von der abstrakten Klasse `AbstractDeliveryService` erbt, die Methode `List<Event> tick(long, List<ConfirmedOrder>)`. Diese kriegt den momentanen Tick der Simulation und die neu hinzugekommen Bestellungen übergeben. Zuerst wird die Methode `tick(long)` des Attributes `vehicleManager` aufgerufen, wodurch alle Fahrzeuge eine Bewegung durchführen. Die Rückgabe dieses Aufrufes wird zwischengespeichert und zum Schluss zurückgegeben. Danach werden alle neuen Bestellungen dem Attribut `pendingOrders` hinzugefügt, welche daraufhin so sortiert wird, das die Bestellung mit der frühsten Auslieferungszeit als erstes vorkommen. Den Beginn der Auslieferungszeit einer Bestellung können Sie dabei über `order.getTimeInterval().getStart()` abfragen. Zum Schluss wird auf jedes Fahrzeug, welches sich im Lagerhaus befindet, so viele Bestellungen aus der Liste `pendingOrders` aufgeladen, wie es seine Kapazität zulässt. Vergessen Sie dabei nicht die aufgeladenen Bestellungen aus dem Attribut `pendingOrders` wieder zu entfernen. Die Bestellungen werden dabei in der Reihenfolge aufgeladen in der Sie in `pendingOrders` vorkommen. Mit der Methode `moveQeueued` aus dem Interface `Vehicle` können Sie ein Fahrzeug dazu auffordern zu einer bestimmten Position zu fahren. Wenn ein Fahrzeug alle Bestellungen ausgeliefert hat, soll es wieder zum Lagerhaus fahren.

H9.2: Ihr eigener DeliveryService

In dieser Aufgabe implementieren Sie in der Methode `tick` Klasse `OurDeliveryService` ihren eigenen DeliveryService. Wie genau dieser aussieht ist dabei Ihnen überlassen. Die einzige Anforderung ist, dass das Aufrufen der `tick` Methode des VehicleManagers, das Hinzufügen der neuen Bestellungen zur Liste `pendingOrders`, sowie die Rückgabe der Methode genauso funktioniert, wie in der H9.1.

Bewertet wird Ihre Implementation anhand der Anzahl an Problemstellungen, die ausreichend gut von dem `Rater` aus H8 bewertet werden. Genauere Informationen dazu werden Sie in den Public Tests finden.

> **Hinweis:**
>
> Die Bewertung wird in den Public Tests anhand Ihrer eigenen Implementation ermittelt. Stellen Sie also zuerst sicher, dass alle anderen Aufgaben korrekt funktionieren, bevor Sie sich die Ergebnisse für diese Aufgabe anschauen.

H10: Lauf Simulation, lauf!

Nun steht die grundsätzliche Simulation, aber es gibt noch keine einfache Möglichkeit Ihre Implementationen aus H9 einfach zu simulieren und anhand den Bewertungskriterien aus H8 zu bewerten. Dafür werden Sie nun das Interface `Runner` implementieren, dessen `run` Methode genau diese Funktionalität umsetzt. Die Methode `run` hat dabei drei Parameter. Der erste Parameter ist vom Typ `ProblemGroup`. Ein Objekt vom Typ `ProblemGroup` besteht aus einer `Map<RatingCriteria, Rater.Factory>`, welche beschreibt, für welche Bewertungskriterien welche `Rater` benutzt werden soll, und eine `List<ProblemArchetype>`, welche die Probleme beschreibt, aus denen die Gruppe besteht. Jedes dieser Probleme, also ein Objekt vom Typ `ProblemArchetype`, besteht aus einer `OrderGenerator.Factory`, welche die eingehenden Bestellungen erzeugt, einem `VehicleManager`, welche die Fahrzeuge und zugrundelegende Region verwaltet, und der Länge der Simulation.

H10.1: BasicRunner

Implementieren Sie nun die Methode `run` der Klasse `BasicRunner` im Package `projekt.delivery.Runner`, welche wie folgt vorgeht. Zuerst wird für jedes Problem der `ProblemGroup` eine eigene Simulation vom Typ `BasicDeliverySimulation` erstellt. Als Parameter für den Konstruktor übergeben Sie die Werte, welche in der `ProblemGroup` und dem `ProblemArchetype`, für welches die Simulation erstellt wird, enthalten sind. Als `DeliveryService` erstellen Sie ein neues Objekt vom Typ `ProblemSolverDeliveryService`, welches Sie mit dem momentan `ProblemArchetype` initialisieren. Danach werden diese Simulationen so oft ausgeführt, wie im dritten Parameter angegeben und die Bewertung für jedes Bewertungskriterium einzeln gespeichert. Die einzelnen Bewertungen können Sie über die Methode `Simulation#getRatingForCriterion(RatingCriteria)` abfragen. Zum Schluss soll eine `Map<RatingCriteria, Double>` zurückgegeben werden, welche für jedes Bewertungskriterium die durchschnittliche Bewertung der einzelnen Durchläufe der Simulation enthält.

> **Verbindliche Anforderung:**
>
> Verwenden Sie zum Erstellen der Simulationen die bereits definierte Methode `createSimulations(ProblemGroup, SimulationConfig)`.

H11: Die GUI

Dokumentation:

Listen Sie alle der obigen Funktionalitäten, die Sie realisiert haben, in einer PDF-Datei auf. Beschreiben Sie bitte zu jeder dieser Funktionalitäten, 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 ausgeben, wenn momentan keine da ist), werden nicht angeboten, also entweder gar nicht oder nur ausgegraut (und ohne Reaktion) angezeigt.

H11.1: Startmenü

Implementieren Sie ein Startmenü, welches beim Programmstart automatisch ausgeführt. Dieses soll folgende Funktionalitäten beinhalten:

1. Eine Liste mit allen momentan `ProblemArchetype` Instanzen, welche benutzt werden wird.
2. Für die Liste soll es die Möglichkeit geben Elemente zu entfernen und hinzuzufügen. Beim Hinzufügen soll sich ein Fenster zum Bearbeiten einer Probleminstanz mit den folgenden Einstellungen und einem Bestätigungsknopf öffnen.
   a) Eingabe für den Namen
   b) Auswahl der Länge der Simulation
   c) Einstellung der einzelnen Parameter eines `FridayOrderGenerator`.
   d) Das Öffnen eines Fenster in der eine interaktive Karte angezeigt wird, zu der man Knoten und Kanten einer `Region` hinzufügen kann. Mit einem Button kann man diese `Region` zu der Erstellung der Probleminstanz hinzufügen.
3. Auswahl, ob die Simulation in einem Interaktiven Modus, entsprechend der H11.2.
4. Button zum Starten einer Simulation mit den ausgewählten `ProblemArchetypes`. Erstellen Sie dafür eine `ProblemGroup` und führen Sie diese mit einer passenden Subklasse des Interfaces `Runner` aus.

H11.2: Grundfunktionalitäten der GUI

Schreiben Sie für das Programm eine grafische Benutzeroberfläche, die es erlaubt, interaktiv den Verlauf einer Simulation anzuzeigen. Sie soll mindestens folgende Funktionalitäten bieten:

1. Die Geschwindigkeit, mit der ein Tick ausgeführt wird, soll mittels eines Sliders auswählbar sein.
2. Es soll eine Karte anzeigt werden, 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 auf 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. Es gibt ein Button, welcher ein Dialog anzeigt, in dem 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.

H11.3: Ergebnisübersicht

Zum Abschluss einer Simulation soll sich ein neues Fenster öffnen, bei dem man mittels einem Balkendiagramm die Scores der einzelnen Bewertungskriterien sieht. Desweiteren existiert ein Knopf, mit welchem man zurück zum Startmenü kommt.

H11.4: Erweiterungsmöglichkeiten für die GUI

Darüber hinaus bringen folgende Funktionalitäten weitere Punkte:

1. Eine Erweiterung der Karte, die Fahrzeugpositionen in Echtzeit anzeigt:
   a) Fahrzeuge, die gerade eine Lieferung ausführen, sollen auf dem 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) Ein existierender Knoten kann mit Drag & Drop verschoben werden. Dabei bleiben alle Daten des Zielortes außer den Koordinaten gleich.
   b) Die Möglichkeit Änderung der Daten eines existierenden Knotens über ein Popup-Fenster vorzunehmen.
   c) Eine Option um ein Zielort aus der Tour eines Fahrzeugs in die Tour eines anderen Fahrzeugs zu verschieben.
3. *„Intelligentes" Editieren*: **???Bei der Erstellung oder Verschiebung eines Knotens wird ein Fahrzeug und ein geeignetes Zeitfenster angeboten.???**
4. Die Ansicht der Karte ist veränderbar:
   a) Es ist möglich mittels $+/-$ Buttons Hinein- und Herauszoomen.
   b) Den im Fenster gezeigte Kartenausschnitt (vor allem im hineingezoomten Zustand) ist mittels Mausaktionen verschiebbar.
   c) Die Ergebnisse mehrerer **???Algorithmen???** können parallel neben- oder übereinander anzeigt werden.

H12: Unit Tests

In dieser Aufgabe geht es darum, die in den anderen Aufgaben vorgenommenen Implementierungen, auf Korrektheit zu testen. Dazu nutzen Sie wie üblich JUnit, beziehungsweise die `Assertions` Klasse von JUnit.

[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.

Verbindliche Anforderung:

Alle geforderten Überprüfungen müssen mittels der Klasse `org.junit.jupiter.api.Assertions` erfolgen.

H12.1: Object und Comparable Tests

Implementieren Sie die folgenden fehlenden Methoden der Klasse `ObjectUnitTests`, welche automatisiert `equals`, `hashCode`, sowie `toString`, eines Objektes vom Typ `Object`, auf Korrektheit prüft. Im bereits implementierten Konstruktor der Klasse wird eine Factory `testObjectFactory` übergeben, mit dem Vertrag `testObjectFactory.apply(i).equals(testObjectFactory.apply(j))`, falls `i == j` und `!testObjectFactory.apply(i).equals(testObjectFactory.apply(j))`, falls `i != j`. Zusätzlich wird eine `Function<T, String> toString` übergeben, die für ein beliebiges Objekt `o` festen Typs, den korrekten Wert für `o.toString()` zurückliefert. Also `toString.apply(o).equals(o.toString())`.

1. `initialize(int testObjectCount)`: Initialisieren Sie `testObjects`, `testObjectsReferenceEquality` und `testObjectsContentEquality` jeweils mit neuen Arrays der Länge `testObjectCount`. Befüllen Sie die Arrays `testObjects` und `testObjectsContentEquality` so, dass sich an Index `i` jeweils unterschiedliche Objekte befinden, die aber beide durch die `testObjectFactory` mit `i` als Parameter erzeugt wurden. Lassen Sie `testObjectsReferenceEquality` an Index `i` auf `testObjects` an Stelle `i` verweisen.
2. `testEquals()`: Schreiben Sie **genau drei** Equals-Assertions, sodass für alle validen Indizes `i` **genau einmal** getestet wird, ob `testObjects`, `testObjectsReferenceEquality` und `testObjectsContentEquality` an Stelle `i` äquivalent sind. Schreiben Sie zusätzlich **genau eine** NotEquals-Assertion, die `testObjects` für alle validen Indizes `i`, `j` mit `i != j` **genau einmal** darauf hin prüft, dass die Objekte an Stellen `i` und `j` ungleich sind.
3. `testHashCode()`: Implementieren Sie `testHashCode` analog zu `testEquals`, mit dem einzigen Unterschied, dass statt `equals` die Methode `hashCode` geprüft wird.
4. `testToString()`: Schreiben Sie **genau eine** Equals-Assertion, die für alle in `testObject` enthaltenen Objekte `o` **genau einmal** testet, ob der Aufruf von `o.toString()` das gleiche Ergebnis liefert, wie das Attribut `Function<Object, String> toString` mit `o` als Parameter.

Vervollständigen Sie nun die Methoden der Klasse `ComparableUnitTests`, welche `compareTo`, des Interfaces `Comparable`, auf Korrektheit prüft. Im bereits implementierten Konstruktor der Klasse wird eine Factory `testObjectFactory` übergeben, mit dem Vertrag `testObjectFactory.apply(i).compareTo(testObjectFactory.apply(j)) <op> 0`, falls `i <op> j` für alle `<op>` $\in$ {`==`, `<`, `>`}.

1. `initialize(int testObjectCount)`: Initialisieren Sie `testObjects` mit einem neuem Array der Länge `testObjectCount` und befüllen Sie dieses so, dass sich an Index `i` ein Objekt befindet, dass durch die `testObjectFactory` mit `i` als Parameter erzeugt wurde.
2. `testBiggerThen()`: Schreiben Sie **genau eine** Assertion, die für jede Kombination an validen Indizes `i`, `j` mit `i > j` **genau einmal** mittels `compareTo` prüft ob das Objekt an an Stelle `i` von `testObjects`, größer als das Objekt an Stelle `j` ist.
3. `testAsBigAs()`: Mittels **genau einer** Equals-Assertion soll hier für jedes Objekt aus `testObjects` **genau einmal** geprüft werden, ob ein Vergleich über `compareTo`, mit sich selbst, 0 zurückgibt.
4. `testLessThen()`: Überprüfen Sie analog zu `testBiggerThen()`, ob das Objekt an Index `i` kleiner als das Objekt an Index `j` ist, wenn `i < j`.

Hinweis:

Wenn die Anforderung ist, dass man nur eine bestimmte Anzahl an Assertions verwenden darf, bezieht sich dies auf die Anzahl an Vorkommen im Quellcode. D.h. auch in Schleifen zählt ein Aufruf nur als einmal verwendet.

Der Titel `<T>` jeder folgenden Unteraufgabe gibt die dort zu testende Klasse an. Die Tests erfolgen in einer dazugehörigen, bereits in der Vorlage vorhandenen, Unit-Test-Klasse mit den Namen `<T>UnitTests`.

Beispiel: Titel = Location; zu testende Klasse = `Location`; Unit-Test-Klasse = `LocationUnitTests`.

H12.2: Location

Vervollständigen Sie die Methode `initialize()` von `LocationUnitTests`, indem Sie eine lokale Variable `testObjectFactory`, unter Einhaltung des Vertrags des Konstruktors von `ObjectUnitTests` und `ComparableUnitTests`, initialisieren. Dabei soll die Factory für alle Eingabewerte `i >= 0` funktionieren und garantieren, dass für Eingabewerte `i, j < 10` mindestens eine `Location`-Kombination erzeugt wird, bei der sowohl x-, als auch y-Koordinate verschieden sind. Initialisieren Sie anschließend `comparableUnitTests` und `objectUnitTests` mit `testObjectFactory` als ersten Parameter und mit einer den Methodenvertrag erfüllenden toString-Funktion, bei `comparableUnitTests`, als zweiten Parameter. Rufen Sie dann `initialize(int)` von `objectUnitTests` und `comparableUnitTests` so auf, dass die Tests mit 10 Testobjekten initialisiert werden. Delegieren Sie in den Testmethoden `testAsBigAs`, `testHashCode`, ... der Klasse `LocationUnitTests` die Tests von `compareTo`, `hashCode`, ... an die dazugehörigen Methoden von `objectUnitTests` bzw. `comparableUnitTests` weiter. Überprüfen Sie abschließend anhand der Tests, ob Ihre Implementierung korrekt ist und passen sie diese andernfalls an.

H12.3: RegionImpl

Implementieren Sie die Methode `initialize()`, sowie `equalsTests()` und `hashCodeTests()` analog zu `LocationUnitTests`, mit dem Unterschied, dass als Parameterwert für `toString` `null` übergeben wird, der Teil für `comparableUnitTests` ignoriert wird und keine Einschränkungen für die `testObjectFactory`, außer natürlich dem Methodenvertrag beachtet werden müssen. Ebenso soll selbstverständlich `RegionImpl` statt `LocationImpl` getestet werden. Implementieren Sie zusätzlich folgende Methoden:

1. `testNodes()`: Erzeugen Sie zunächst ein Objekt von `RegionImpl` und fügen Sie mit `region.putNode(NodeImpl)` drei Knoten A, B, C zu `region` hinzu und überprüfen Sie mittels `region.getNodes()` und `genau drei` passender Assertions, dass alle Knoten vorhanden sind. Wiederholen Sie die Prüfung mittels `getNode(Location)` für alle drei Knoten mit **jeweils** einer weiteren Assertion (insgesamt also drei weitere Assertions). Prüfen Sie abschließend, dass `getNode(Location)` für eine Position, an der in der Region kein Knoten vorhandenen ist, `null` zurück gibt. Fügen Sie außerdem einen Knoten hinzu, der einer anderen Region angehörig ist, dies sollte zu einer `IllegalArgumentException` führen, was wie der vorheriger Fall mittels Assertion überprüft wird. Überprüfen Sie ebenfalls, ob die Botschaft der Exception korrekt gesetzt wurde.
2. `testEdges()`: Erzeugen Sie zunächst ein Objekt von `RegionImpl` mit den gleichen Knoten, wie zuvor in `testNodes()`. Durch `region.putEdge(EdgeImpl)` sollen Kanten in `region` generiert werden, sodass Knoten A auf sich selbst und Knoten B verweist. Knoten B verweist auf Knoten C und Konten C verweist auf keinen Knoten. Stellen Sie mit **genau vier** Assertions sicher, dass `region.getEdge(Location, Location)` eben jene Kanten zurückliefert und durch vier weitere Assertion, dass das auch für `region.getEdges()` der Fall ist. Befindet sich ein Knoten der Kante, oder die Kante selbst nicht in der Region, muss `putEdge(EdgeImpl)` eine `IllegalArgumentException` werfen. Stellen Sie dies mittels **drei** weiterer Assertions sicher, welche jeweils testen, dass genau eine der drei Komponenten nicht zu der selben region gehört.

H12.4: NodeImpl

Ergänzen Sie die Methoden aus `NodeImplUnitTests` analog zu `LocationUnitTests`, mit dem Unterschied, dass in `initialize()` keine Einschränkungen für die `testObjectFactory`, außer natürlich dem Methodenvertrag beachtet werden müssen und, dass selbstverständlich `NodeImpl` statt `Location` getestet wird. Erzeugen sie in der `initialize()` Methode zusätzlich eine Region, die analog zu der Region in `testNodes()` von `RegionImplUnitTests` aufgebaut ist, und speichern Sie diese im Attribut `region`, sowie die Knoten in den Attributen `node{A,B,C}`. Schreiben Sie dann zwei Testmethoden:

1. `testGetEdge()`: Testen Sie mittels **genau drei** Assertions, dass die Methode `nodeA.getEdge(Region.Node)` für jeden der drei Knoten als Input, die passende Kante, bzw. `null`, zurückgibt.
2. `testAdjacent()`: Überprüfen sie mithilfe von Assertions, dass `getAdjacentNodes()` und `getAdjacentEdges()` für die drei verschiedenen Kanten die korrekten angrenzenden Kanten und Knoten zurückliefert.

H12.5: EdgeImpl

Ergänzen Sie die Methoden aus `EdgeImplUnitTests` analog zu `LocationUnitTests`, mit dem Unterschied, dass in `initialize` keine Einschränkungen für die `testObjectFactory`, außer natürlich dem Methodenvertrag beachtet werden müssen und das selbstverständlich `EdgeImpl` statt `Location` getestet wird. Erzeugen sie in der `initialize()` Methode zusätzlich eine Region, die analog zu der Region in `testNodes()` von `RegionImplUnitTests` aufgebaut ist, und speichern Sie diese im Attribut `region`, sowie die Kanten in den Attributen `edge{AA,AB,BC}`. Schreiben Sie dann folgende Testmethode:

1. `testGetNode()`: Testen Sie mittels **jeweils drei** Assertions, dass die Methoden `getNodeA()` und `getNodeB()` aufgerufen auf jeden der drei Kanten, den passenden Knoten, zurückgibt. Erstellen Sie zusätzlich zwei Objekte der Klasse `EdgeImpl`, bei denen einmal `locationA` und einmal `locationB` nicht in der Region enthalten ist und überprüfen Sie mit **jeweils einer** Assertion, dass die entsprechende `getNode` Methode `null` zurückgibt.