Funktionale und objektorientierte Programmierkonzepte Projektaufgabe

Prof. Karsten Weihe

Wintersemester 22/23	v1.0
Themen:	Alle Konzepte der FOP
Relevante Foliensätze:	01-15
Abgabe der Hausübung:	17.03.2023 bis 23:50 Uhr

FOP Projekt — Gesamt: 200 Punkte

Simulation eines Lieferservice

Beachten Sie die Informationen zum Bearbeiten des Projekts im Moodle-Kurs im Abschnitt *Projekt*.

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

(a) Sie dürfen, wenn nicht anders in der Aufgabe angegeben, alle in der Vorlesung vorgestellten Konstrukte verwenden.

(b) Sie dürfen weitere Methoden und Klassen hinzufügen, aber nicht die vorhandenen Konstrukte, im Besonderen die gegebenen Interfaces, nicht verändern, oder bei aufeinander aufbauenden Aufgaben davon ausgehen, dass Ihre eigenen Änderungen der vorherigen Aufgaben vorhanden sind.

(c) Schreiben Sie Ihre Implementationen im Verzeichnis src/main/java/projekt. Die Aufgabe H12 wird im Verzeichnis `src/test/java/projekt` umgesetzt. Alle Aufgaben bis auf die H11 wird dabei im Subproject `Domain` im Verzeichnis `domain/src` umgesetzt. Aufgabe H11 wird im Subproject `Infrastructure` umgesetzt.

Einleitung

Sie wurden von einem bekannten Essenslieferdienst dazu aufgetragen ein Programm zu entwickeln, welches es einem ermöglicht den Ablauf eines Lieferdiensts zu simulieren und am Ende zu bewerten. Vervollständigen Sie dafür anhand der folgenden Aufgaben die Vorlage, welche wir Ihnen zur 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 H11 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 mittels eines Graphen aus Knoten (siehe H3) und Kanten (siehe H4) den Aufbau des Liefergebietes. Die nächste Schicht wird von dem Interface `VehicleManager` (siehe H6) umgesetzt und ist dafür zuständig die verschiedenen Fahrzeuge (Siehe H5) 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. Jede dieser Schichten besitzt dabei einen Verweis auf die unter ihr liegende Schicht.

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 nach unten die `tick(long)` Methode der einzelnen Schichten aufgerufen, welche daraufhin alle in den nächsten Zustand übergehen. Jeder Simulationsschritt, der dabei ausgeführt wird, erzeugt ein `Event`. Eine Liste aller erzeugten `Events` wird am Ende von den `tick` Methoden zurückgegeben. Diese Events werden am Ende der Simulation u.a. 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.
`tickInterval`	`TickInterval`	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

Für die Erzeugung dieser Bestellungen ist das Interface `OrderGenerator` (siehe H7) verantwortlich.

Welche Probleme simuliert werden, wird von Implementierungen des Interface `ProblemArchetype` beschrieben. Klassen, die dieses Interface implementieren, bestehen aus jeweils vier Komponenten, die über die im Interface deklarierten Methoden abgerufen werden können.

- **Vehicle Manager**: Beschreibt den Aufbau der zugrundeliegenden Region und die verfügbaren Fahrzeuge.
- **Order Generator Factory**: Beschreibt die Bestellungen, welche bei dem Lieferservice eingehen (siehe H7).
- **Rater Factory Map**: Beschreibt, welche `Rater` zur Bewertung der jeweiligen Kriterien verwendet wird.
- **Simulation Length**: 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`, die Anzahl an auszuführenden Simulation, sowie eine `deliveryServiceFactory` bekommt. Diese ist dafür zuständig für jedes Objekt von `ProblemArchetype` aus der `ProblemGroup` eine Simulation basierend auf dem erstellten `DeliveryService` zu erstellen, diese so oft durchzuführen, wie angegeben, und am Ende für jedes Bewertungskriterium die durchschnittliche Punktzahl zu berechnen und zurückzugeben.

H1: Location — 6 Punkte

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

> **Hinweis:**
>
> Implementieren Sie alle Aufgaben außer der H11 im Verzeichnis Domain.

H1.1: `compareTo` — 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: `hashCode` — 2 Punkte

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 $x, y \in \{n \in \mathbb{Z} : -1024 \leq n \leq 1023\}$ unterschiedliche Werte zurückliefern. Bei Ihrer Implementierung ist Ihnen freigestellt, wie Sie diese Vorgabe umsetzen.

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

H1.3: `equals` 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.

H1.4: `toString` 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* 13 Punkte

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. Restaurantknoten stellen die einzelnen Restaurants dar bei denen Bestellungen aufgegeben werden, 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.

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.

> **Hinweis:**
>
> Beachten Sie, dass die Aufgaben H2 bis H4 stark voneinander abhängen. Lesen Sie sich also zunächst alle drei Aufgaben durch, bevor Sie anfangen, diese zu implementieren.

> **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: Wo Noed? — 1 Punkt

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: Da Noed! — 2 Punkte

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 (`this`) 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. Wenn eine Exception geworfen wird, soll der Knoten nicht in die Map eingefügt werden.

H2.3: `getEdge` — 3 Punkte

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, wie im letzten Absatz der Einleitung der H2 erklärt, übergeben werden können und die Funktion trotzdem die korrekte Kante liefern muss.

H2.4: `putEdge` — 4 Punkte

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 Kante ersetzt werden soll. Wenn `edge.getNodeA()` oder `edge.getNodeB()` `null` zurückgibt, soll eine `IllegalArgumentException` mit der Botschaft `"Node{A,B} <location> is not part of the region"` geworfen werden, wobei `"<location>"` mit String Repräsentation der entsprechenden `Location`. Wenn mehrere dieser Fälle zutrifft, ist es Ihnen überlassen, welche der passenden Botschaften gewählt wird. Wenn eine Exception geworfen wird, soll die Kante nicht in die Map, bzw. die Liste, eingefügt werden.

> **Hinweis:**
>
> Achten Sie beim Einfügen einer Kante darauf, dass die im letzten Absatz der Einleitung der H2 Sortierung von `allEdges` erhalten bleibt.
> Im Konstruktor von `EdgeImpl` wird dabei bereits darauf geachtet, dass gilt `nodeA.getLocation().compareTo(nodeB.getLocation()) <= 0`.

H2.5: Wo Noeds? — 1 Punkt

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

H2.6: `equals` — 1 Punkt

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` (oder einem Subtyp) 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: `hashCode` — 1 Punkt

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* — 9 Punkte

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: `getEdge` — 1 Punkt

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.

H3.2: `getAdjacentNodes` — 2 Punkte

Implementieren Sie die Funktion `public Set<Region.Node> getAdjacentNodes()`, welche alle Knoten zurückliefern soll, die mit dem (aktuellen) Knoten verbunden sind. Wenn ein Knoten eine Kante zu sich selber besitzt, soll der Knoten selber ebenfalls zurückgegeben werden.

H3.3: `getAdjacentEdges` — 2 Punkte

Implementieren Sie die Funktion `public Set<Region.Edge> getAdjacentEdges()`, welche alle mit dem (aktuellen) Knoten verknüpfte Kanten zurückgibt. Wenn ein Knoten eine Kante zu sich selber besitzt, soll diese ebenfalls zurückgegeben werden.

H3.4: `compareTo` — 1 Punkt

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.

H3.5: `equals` — 1 Punkt

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` (oder einem Subtyp) 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.

H3.6: `hashCode` — 1 Punkt

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.

H3.7: `toString` 1 Punkt

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* 6 Punkte

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: `getNode{A,B}` 1 Punkt

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.

H4.2: `compareTo` 2 Punkte

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.

H4.3: `equals` 1 Punkt

Implementieren Sie die Funktion `public boolean equals(Object o)`. Sie soll prüfen, ob das übergebene Objekt diesem Objekt entspricht. Bei Gleichheit nach Gleichheitsoperator (==) soll `true` zurückgegeben werden, falls das übergebene Objekt `null` ist oder einen anderen Datentyp als `EdgeImpl` (oder Subtyp) hat, soll `false` zurückgegeben werden. Anderenfalls soll `Objects.equals` mit `this.name`, `o.name` und auf `,Bthis.locationA`, `,Bo.locationA` sowie auf `this.duration`, `o.duration` aufgerufen und per && verknüpft zurückgegeben werden.

H4.4: `hashCode` 1 Punkt

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.

H4.5: `toString` 1 Punkt

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... 16 Punkte

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.

In dieser und der nächsten Aufgabe wird dabei die Klasse `AbstractOccupied` und deren Subklassen verwendet 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.

H5.1: Das Zünglein an der Wage 1 Punkt

Implementieren Sie die Methode `public double getCurrentWeight()` im Interface `Vehicle`, die das aktuelle Gewicht der Ladung des Fahrzeugs zurückgibt, indem das Gewicht der einzelnen geladenen Bestellungen 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 3 Punkte

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 6 Punkte

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 — 6 Punkte

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 einer Kante befindet, soll sich das Fahrzeug zunächst bis zum nächsten Knoten bewegen, wofür ein Pfad zum ersten Knoten aus der alten `moveQueue` wieder der `moveQueue` hinzugefügt wird. Danach wird wie zuvor `moveQueued()` aufgerufen.

H6: Wo ist eigentlich mein Auto? — 12 Punkte

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.

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

H6.1: `toOccupiedNodes` — 4 Punkte

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: `getAllOccupied` — 1 Punkt

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

H6.3: `getOccupied` — 4 Punkte

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 des Typs `Region.Node` noch um eines des Typs `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, wird 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: `getOccupiedNeighborhood` — 3 Punkte

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 `"Node 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 `"Node <node> is not a neighborhood"` geworfen werden, wobei `<node>` mit der String-Repräsentation des aktualen Parameters ersetzt wird. Ansonsten soll einfach der in der Map zugeordnete Wert zurückgegeben werden.

Vervollständigen Sie mit derselben Logik – natürlich angepasst - die Methode `getOccupiedRestaurant`.

H7: Was gibt es heute zu Essen? — 9 Punkte

Als Nächstes wollen wir die Möglichkeit haben, automatisch zufällige Bestellungen zu generieren. Dazu finden Sie im Package `projekt.delivery.generator` 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 — 9 Punkte

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 immer dieselben Bestellungen zurückgeben, wenn derselbe 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 `TickInterval`, welches als Startpunkt den Tick, zu dem die Order ausgeliefert wird, hat und als Endpunkt den Startpunkt plus die Tickanzahl, welche im Konstruktorparameter `deliveryInterval` 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 (inklusive) und dem Wert des Konstruktorparameters `maxWeight` (exklusive).

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? 13 Punkte

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? 3 Punkte

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 `double`, 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:

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

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? 6 Punkte

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 fünf Ticks zu spät ausgeliefert wurde, als pünktlich und eine Bestellung, die sechs Ticks zu spät ausgeliefert wurde als einen Tick zu spät.
`maxTicksOff` gibt an, was die maximale Tickanzahl ist, die bei Verspätungen berücksichtigt wird. Bei einem Wert von `maxTicksOff = 25` und `ignoredTicksOff = 5`, zählt eine Bestellung, die 40 oder auch 1000 Ticks zu spät ausgeliefert wurde, als 25 Ticks zu spät. Eine Bestellung, die 28 Ticks zu spät ist, zählt allerdings aufgrund von `ignoredTicksOff` weiterhin als 23 Ticks zu spät.

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

$$\text{score} = \begin{cases} 1 - \frac{\text{actualTotalTicksOff}}{\text{maxTotalTicksOff}} & \text{maxTicksOff} \neq 0 \\ 0 & \text{sonst} \end{cases}$$

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.

> **Hinweis:**
>
> Bestellungen, welche zu früh ausgeliefert werden, sollen genauso in die Bewertung einbezogen werden, wie Bestellungen, welche zu spät ausgeliefert wurden.
> Wann eine Bestellung ausgeliefert werden soll, können Sie über `order.getDeliveryInterval().getStart()`, bzw. `order.getDeliveryInterval().getEnd()` abfragen.

Sie können dafür in der `onTick` Methode dieselben Events betrachten, wie in H8.1.

H8.3: Wie viel bin ich gefahren? 4 Punkte

Vervollständigen Sie die Klasse `TravelDistanceRater`, welche eine Simulation anhand des Bewertungskriteriums `TRAVEL_DISTANCE`, also wie weit die einzelnen Fahrzeuge gefahren sind, bewertet. Sie besitzt das Attribut `factor` vom Typ `double`, 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:

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

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 `onTick` 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. Beachten Sie dabei, dass in der Rückgabe der Startknoten nicht enthalten ist. 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! 28 Punkte

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.service` 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 8 Punkte

Implementieren Sie zunächst in der Klasse `BasicDeliveryService`, 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.getDeliveryInterval().getStart()` abfragen. Zum Schluss wird auf jedes Fahrzeug, welches sich in einem Restaurant befindet, so viele Bestellungen aus der Liste `pendingOrders` aufgeladen, wie es seine Kapazität zulässt. Dabei kann ein Fahrzeug allerdings nur Bestellungen von dem Restaurant aufnehmen, auf dem es sich momentan befindet. Mittels `vehicleManager.getOccupiedRestaurants()` können Sie abfragen, welche Restaurants existieren, und welche Fahrzeuge sich auf diesen befinden. 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 `moveQueued()` aus dem Interface `Vehicle` können Sie ein Fahrzeug dazu auffordern zu einer bestimmten Position zu fahren. Wenn das Fahrzeug dabei an einem Lieferort ankommt, soll es über die arrivalAction der Methode `moveQueued()` alle passenden Bestellungen ausliefern. Bestellungen können über die Methode `deliverOrder()` der Klasse `OccupiedNeighborhood` ausgeliefert werden. Auf eine `OccupiedNeighborhood` können Sie über die Methode `getOccupiedNeighborhood(regionn.Node)` des `vehicleManagers` zugreifen. Wenn auf ein Fahrzeug mehrere Bestellungen geladen werden, welche zur selben Position geliefert werden sollen, soll die `moveQueued` Methode nur einmal für diese Position aufgerufen werden. Wenn ein Fahrzeug alle Bestellungen ausgeliefert hat, soll es wieder zum Restaurant vom Anfang fahren.

H9.2: Ihr eigener DeliveryService 20 Punkte

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:

Wenn Sie Attribute zu der Klasse hinzufügen müssen Sie diese in der `reset()` Methode zurücksetzten. Achten SIe beim Überschreiben der Methode darauf, dass weiterhin die `rest()` Methode der Superklasse aufgerufen wird.

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! 5 Punkte

Nun steht die grundsätzliche Simulation, aber es gibt noch keine einfache Möglichkeit Ihre Implementationen aus H9 einfach zu simulieren und anhand der Bewertungskriterien aus H8 zu bewerten. Dafür werden Sie nun das Interface `Runner` im Package `projekt.delivery.runner` implementieren, dessen `run` Methode genau diese Funktionalität umsetzt. Die Methode `run` hat dabei vier Parameter. Der erste Parameter ist vom Typ `ProblemGroup`. Ein Objekt vom Typ `ProblemGroup` besteht aus einer `List<RatingCriteria>`, welche beschreibt, anhand welcher Kriterien die Simulation bewertet 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`, welcher die Fahrzeuge und zugrundelegende Region verwaltet, einer `Map<RatingCriteria, Rater.Factory>`, die beschreibt anhand welches Raters ein Kriterium bewertet werden soll, der Länge der Simulation, sowie einem Namen. Der zweite Parameter ist vom Typ `SimulationConfig`, welche grundlegende Einstellungen für die Simulation beinhaltet und der dritte Parameter gibt an, wie oft die einzelnen Simulationen durchgeführt werden sollen. Als Letztes wird ein Objekt vom Typ `DeliveryService.Factory` übergeben, mit welchen die simulierten `DeliveryService` Instanzen erstellt werden.

H10.1: Simulationen erstellen 1 Punkt

Vervollständigen Sie zunächst in der Klasse `AbstractRunner` die Methode `createSimulations()`. Diese erstellt für jedes Problem in der übergebenen `ProblemGroup` ein Objekt vom Typ `BasicDeliverySimulation` und gibt alle Simulationen in einer `Map<ProblemArchetype, Simulation` zurück. Die Values der zurückgegebenen Map sind dabei die erzeugten `BasicDeliverySimulations` und die jeweiligen Keys das `ProlbemArchetype`, für welches die SImulation erzeugt wurde. Die Konstruktoren der Simulationen erhalten dabei als Parameter die übergebene `SimulationConfig`, die Eigenschaften der jeweiligen Probleme, und als `DeliveryService` die Rückgabe der `apply` Methode der `deliveryServiceFactory`. Die `apply` Methode kriegt dabei als Parameter den `VehicleManager` des jeweiligen Problems übergeben.

H10.2: BasicRunner — 4 Punkte

Implementieren Sie nun die Methode `run` der Klasse `BasicRunner`, welche wie folgt vorgeht. Zuerst wird für jedes Problem der `ProblemGroup` eine eigene Simulation mittels der in Aufgabe H10.1 implementierten Methode erstellt. Danach werden diese Simulationen so oft mittels der Methode `runSimulation(long)` ausgeführt, wie im dritten Parameter angegeben, und die Bewertung für jedes Bewertungskriterium einzeln gespeichert. Die Methode `runSimulation(long)` von `Simulation` kriegt dabei die Länge der Simulation in Ticks übergeben. 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 Simulationen enthält.

H11: Die GUI — 54 Punkte

Damit haben Sie nun eine vollständig funktionierende Simulation, welche auch ausgeführt werden kann. Als Letztes werden Sie nun noch eine grafische Oberfläche erstellen, mit welcher ein Durchlauf einer Simulation visualisiert werden kann. Wir geben Ihnen bereits einen grundsätzlichen Aufbau der GUI vor. Dieser besteht aus einem Startmenü, von welchem aus die Simulation gestartet werden kann, der Ansicht der momentan laufenden Simulation und einem Endmenü. Diese werden Sie in den folgenden Aufgaben vervollständigen und erweitern. Die Menu Szenen erben dabei alle von der abstrakten Klasse `MenuScene` und funktionieren nach folgendem Prinzip:
Im Konstruktor wird der Superkonstruktor aufgerufen, welche den zugehörigen `SceneController`, den Title der Szene und eine beliebige Anzahl an Stylesheets, welche angewendet werden sollen. Der Superkonstruktor erstellt mit diesen Informationen den grundsätzlichen Aufbau der Szene. Nach dem Erstellen der Szene wird von außerhalb die `init` Methode aufgerufen, welche alle weiteren, notwendigen Informationen übergeben bekommt. Insbesondere bekommt sie dabei immer eine Liste von `ProblemArcheType` übergeben, welche alle Probleme enthält, welche simuliert werden sollen, bzw. simuliert wurden. Diese Liste übergibt die Methode der `init` Methode der Klasse `MenuScene`, welche daraufhin dafür sorgt, dass die beiden abstrakten Methoden `initComponents()` und `initReturnButton()` aufgerufen werden. Diese beiden Methoden werden von den jeweiligen Subklassen implementiert und erstellen die Szenenspezifischen Komponenten und erstellen die Funktionalität des return Knopfes ein. Schauen Sie sich dies auch nochmal beispielhaft in der Klasse `MainMenuScene` an um genau zu verstehen, wie die Klassen aufgebaut sind.

Sämtlicher Code zur GUI befindet sich im Package `projekt.gui`, allerdings im Verzeichnis `infrastructur` und **nicht** im Verzeichnis `domain`. Sie können diesen über die `main()` Methode der dort vorhandenen `Main` Klasse ausführen.

> **Hinweis:**
>
> Der grundsätzliche Aufbau der GUI ist sehr ähnlich zu Übungsblatt 13. Es kann sich also lohnen dieses zunächst zu bearbeiten. Darüber hinaus lohnt es sich vor allem in dieser im Internet nachzuschauen, wie man bestimmte Konstrukte in JavaFX umsetzten, kann, da Sie unter Umständen nicht alles, was Sie verwenden, wollen auf den Folien finden.

> **Hinweis zur Bewertung:**
>
> Diese Aufgabe wird nicht anhand genau definierten Testfällen bewertet, sondern wird manuell bewertet. Dabei wird darauf geschaut, ob Sie alle geforderten Funktionalitäten sinnvoll umgesetzt haben. Achten Sie also darauf, dass ihr GUI verständlich und intuitiv aufgebaut ist. Sie dürfen daher auch den bestehenden Code für GUI nach Belieben abändern, solange am Ende ein vernünftiges Programm mit den geforderten Funktionalitäten vorhanden ist.

H11.1: GUIRUnner — 1 Punkt

Bevor Sie anfangen die GUI zu bauen, müssen Sie noch die Klasse `GUIRunner` Implementieren, welche die Simulationen durchführt und für die Kommunikation mit der GUI verantwortlich ist. Die grundsätzliche Implementation der `run()` Methode ist dabei identisch zur Aufgabe H10.2. Der Unterschied ist, dass vor und nach dem Ausführen und ganz am

Ende weiterer Code zur Kommunikation mit der GUI benötigt wird. Wir stellen Ihnen diesen Code bereits in der Vorlage zur Verfügung. Sie müssen diesen also nur in Ihre Implementation aus H10.2 integrieren.

H11.2: Startmenü — 15 Punkte

In der Klasse `MainMenuScene` im Package `projekt.gui.scene` finden Sie die momentane Implementation des Startmenüs. Erweitern Sie diese Szene um eine Auflistung der Namen aller Probleme, welche simuliert werden, also allen Problem welche in der Liste `problems` der Superklasse gespeichert sind. Zusätzlich soll es eine Möglichkeit geben einen konkreten Namen eines Problems auszuwählen, woraufhin eine Übersicht über die Eigenschaften des jeweiligen Problems angezeigt wird. Diese Übersicht soll alle Eigenschaften des Problems anzeigen, wie z.B. welche `Rater` mit welchen Parametern ausgewählt wurde, für welche Bewertungskriterien kein Rater ausgewählt wurde, welche Knoten und Kanten existieren, wo welche Fahrzeuge mit welcher Kapazität starten, etc. Für die Auflistung der Probleme bietet sich zum Beispiel eine `TableView<ProblemArcheType>` an. Implementieren Sie diese Funktionalitäten in der Methode `initComponents()`. Sie dürfen allerdings auch weitere Bestandteile verändern oder hinzufügen.

H11.3: Endmenü — 6 Punkte

Die Klasse `RaterMenuScene` im Package `projekt.gui.scene` stellt die Szene dar, welche angezeigt wird, nachdem eine Simulation erfolgreich durchgelaufen ist. In der `init()` Methode werden die Ergebnisse, also für welches Bewertungskriterium welcher Score erreicht wurde, der zuvor ausgeführten Simulation übergeben und in dem Attribut `result` gespeichert. Erweitern Sie die Klasse in der Methode `initComponents()` um eine grafische Anzeige der Bewertung, z.B. in der Form eines Balkendiagramms. Dafür eignet sich u.a. ein `BarChart<String, Number>`.

H11.4: Anzeige der Simulation — 7 Punkte

In der Klasse `SimulationScene` im Package `projekt.gui.scene` wird die Szene für die Anzeige der Simulation Erweitern Sie diese um eine Anzeige, welche Informationen über die ID, die Position und die geladenen Bestellungen aller Fahrzeuge, welche Sie über die Methode `getVehicles()` der Klasse `VehicleManager` abfragen können, enthält. Damit diese aktuell bleiben, müssen Sie die angezeigten Daten in der `ontick()` Methode jedes Mal aktualisieren. Beachten Sie dabei, dass Sie bei der `onTick()` Methode ihren Code innerhalb von `Platfrom.runLater(() -> ...)` schreiben müssen.

H11.5: Erstellen weiterer Probleme — 25 Punkte

Erweitern Sie als Letztes noch das Startmenü aus H11.2 um die Möglichkeit Probleme aus der Liste zu entfernen und wieder hinzuzufügen. Dabei soll man beim Hinzufügen neuer Probleme auswählen können, ob man ein komplett neues Problem erstellen möchte, oder ein bereits zuvor erstelltes Problem hinzufügen möchte. Bereits zuvor erstellte Probleme können Sie über die Methode `readProblems()` der Klasse `MenuScene` einlesen. Beim Erstellen eines neuen Problems soll es möglich sein sämtliche Eigenschaften eines Problems (siehe Übersicht über ein Problem aus H11.2) einstellen können. Wenn ein neu erstelltes Problem hinzugefügt wird, soll es automatisch mit der Methode `writeProblem(ProblemArcheType)` der Klasse `MenuScene` gespeichert werden. Es soll dabei nur möglich sein, sinnvolle Werte bei der Erzeugung eines neuen Problems anzugeben. Insbesondere soll es nicht möglich sein ein Problem zu erzeugen, welches denselben Namen, wie ein bereits bestehendes Problem, besitzt, oder dessen Name nur aus Whitespace besteht.

Diese Funktionalität soll in mindestens einer neuen Szene umgesetzt werden. Erstellen Sie dafür zunächst eine neue Controller Klasse im Package `projekt.gui.controller`, welche von der Klasse `MenuSceneController` erbt. Danach erstellen Sie eine neue Szenen-Klasse im Package `projekt.gui.scene`, welche von der Klasse `MenuScene` erbt. Die Klasse `MenuScene` wird dabei mit Klasse `Controller` instanziiert. Als Letztes müssen Sie noch im selben Package in der Klasse `SceneSwitcher` die Szene registrieren, indem Sie dem `enum SceneType` analog zu den bereits vorhandenen Konstanten eine weiter passende Konstante hinzufügen. Sie können sich für die Implementation der Methoden `initReturnButton()` und `getController()` beispielhaft die Implementation in der Klasse `RaterScene` anschauen. Nun können Sie in der Szenen Klasse ihre Implementation umsetzten. Implementieren Sie dafür Methode `initComponents()`, welche von der `init()` Methode aufgerufen wird, und die einzelnen angezeigten Komponenten initialisiert. Wenn notwendig, können Sie die `init()` Methode überladen und weitere Parameter, welche bei der Initialisierung benötigt werden, hinzufügen. Vergessen Sie nicht die `init()` Methode der Superklasse in Ihrer `init()` Methode aufzurufen. Die `init()` Methode von `MainMenuScene` kriegt dabei bereits immer eine Liste von

`ProblemArcheTypes` übergeben, um die momentane Auswahl von Problemen zwischen den einzelnen Szenen zu speichern.

Das Erstellen einer neuen Region und eines neuem VehicleManagers soll gemeinsam auf einer Karte visualisiert werden. Auf diese Karte soll man einzelne Knoten, Kanten und Fahrzeuge hinzufügen und wieder entfernen können - wie genau ist wieder Ihnen überlassen. Für die Karte stellen wir Ihnen die Klasse `MapPane` im Package `projekt.gui.pane` zur Verfügung. Schauen Sie sich die Dokumentation der Klasse an um zu verstehen, wie genau Sie mit der Klasse einzelne Knoten und Kanten visualisieren können. Um Restaurants zu erstellen, soll man aus einer Liste von Voreinstellungen Auswählen können. Diese Voreinstellungen werden als Record in `Region.Restaurant.Preset`. Dort finden Sie bereits definierte Voreinstellungen, welche man beim Erstellen auswählen können soll.

Hinweis (unverbindliche Vorschläge)**:**

Um die Erstellung der RaterFactory und OrderGeneratorFactory können Sie Objekte vom Typ `Rater.FactoryBuilder` und `OrderGenerator.FactoryBuilder` verwalten und erst zum Schluss die Methode `build()` auf diesen aufrufen, um die Factories zu erstellen. Um die erstellte Region und VehicleManager zu verwalten können Sie zwei Objekte vom Typ `Region.Builder` und `VehicleManager.Builder` verwalten, welchen sie mit den `add*()`, bzw. `remove*()` Methoden Komponenten hinzufügen und wieder entfernen können, des Weiteren können Sie mit der `check*()` der Builder abfragen, ob Komponenten hinzugefügt werden können. Erstellen Sie dann nach jeder Änderung mit der `build()` Methode der Builder die momentane `Region`, bzw. `VehicleManager`, und fügen Sie deren Komponenten mit den `addAll*()` Methoden der Karte hinzu.

H12: Unit Tests — 29 Punkte

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 Klasse `Assertions` von JUnit.

Verbindliche Anforderung:

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

H12.1: Object und Comparable Tests — 10 Punkte

Implementieren Sie die folgenden fehlenden Methoden der Klasse `ObjectUnitTests<T>`, 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.

> **Hinweis:**
>
> Sie können davon ausgehen, dass equals$(a, b) \rightarrow$ equals(b, a) gilt, aber nicht, dass equals$(a, b) \wedge$equals$(b, c) \rightarrow$ equals(a, c) gilt.

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<T extends Comparable<? super T>>`, welche `compareTo`, des Interfaces `Comparable<T>`, 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 neuen 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 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.

> **Hinweis:**
>
> Das Erstellen von generischen Arrays ist direkt nicht möglich. Sie können dies aber wie folgt umgehen:
>
> - `(T[]) new Object[size]`, bzw.
> - `(T[]) new Comparable<?>[size]`

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 — 3 Punkte

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 Testme-

thoden `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 — 6 Punkte

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, dass der Methodenvertrag beachtet werden muss, und, dass die erstellten Regionen mindesten zwei Knoten und eine Kante enthalten. 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` zurückgibt. Fügen Sie außerdem einen Knoten hinzu, der einer anderen Region angehörig ist. Dies sollte zu einer `IllegalArgumentException` führen, was wie der vorherige Fall mittels **einer** Assertion überprüft wird. Überprüfen Sie ebenfalls in derselben Assertion, 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 drei** Assertions sicher, dass `region.getEdge(Location, Location)` eben jene Kanten zurückliefert und durch drei 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, soll `putEdge(EdgeImpl)` eine `IllegalArgumentException` werfen. Stellen Sie dies mittels **drei** weiterer Assertions sicher, welche jeweils testen, dass genau eine der drei Komponenten nicht zu derselben region gehört. Stellen Sie mittels einer weiteren Assertions-Abfrage sicher, dass `getEdge` `null` zurückgibt, wenn zwischen den übergebenen Postionen keine Kante vorhanden ist.

H12.4: NodeImpl — 6 Punkte

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. Achten Sie in der `testObjectFactory` darauf, dass der Knoten der `Region`, der dem Konstruktor der Klasse `NodeImplUnitTests` übergeben wird, hinzugefügt wird und keine weiteren Komponenten enthalten sind. Erzeugen Sie in der `initialize()` Methode zusätzlich eine Region, die analog zu der Region in `testEdges()` von `RegionImplUnitTests` aufgebaut ist, und speichern Sie die Kanten und Knoten in den entsprechenden Attributen. Erzeugen Sie ebenfalls eine `nodeD`, welche keine Verbindungen zu den anderen Knoten besitzt. Schreiben Sie dann zwei Testmethoden:

1. `testGetEdge()`: Testen Sie mittels **genau vier** Assertions, dass die Methode `nodeA.getEdge(Region.Node)` für jeden der vier Knoten als Input, die passende Kante, bzw. `null`, zurückgibt.
2. `testAdjacentNode()`: Überprüfen sie mithilfe von vier Assertions, dass `getAdjacentNodes()` für die vier verschiedenen Knoten die korrekten angrenzenden Knoten zurückliefert.
3. `testAdjacentEdges()`: Diese Methode funktioniert analog zu `testAdjacentNode` mit dem Unterschied, dass die Methode `getAdjacentEdges()` getestet wird.

H12.5: EdgeImpl — 4 Punkte

Ergänzen Sie die Methoden aus `EdgeImplUnitTests` analog zu `NodeImplUnitTests`, mit dem Unterschied, dass `EdgeImpl` statt `NodeImpl` getestet wird. Initialisieren Sie die Kanten und Knoten Attribut analog zu `NodeImplUnitTests`, aber ohne `nodeD`. Beachten Sie weiterhin in der `testObjectFactory`, dass die benutzte `Region` ausschließlich die benutzten Knoten und Kanten enthält. Schreiben Sie dann folgende Testmethode:

1. `testGetNode()`: Testen Sie mittels **insgesamt sechs** Assertions, dass die Methoden `getNodeA()` und `getNodeB()` aufgerufen auf jeden der drei Kanten, den passenden Knoten, zurückgibt.