routeKIT

Entwurfsdokument

20. Dezember 2013

Kevin Birke
Felix Dörre
Fabian Hafner
Lucas Werkmeister
Dominic Ziegler
Anastasia Zinkina

betreut durch

Julian Arz
G. Veit Batz
Dr. Dennis Luxen
Dennis Schieferdecker

am

Karlsruher Institut für Technologie
Institut für Theoretische Informatik
Algorithmik II
Prof. Dr. Peter Sanders

Inhaltsverzeichnis

1 Einleitung **2**

2 Übersicht **2**

3 Anmerkungen **3**

4 Änderungen gegenüber dem Pflichtenheft **3**

5 Legende **3**
5.1 Paket BeispielPaket 3

6 Pakete und Klassen **4**
6.1 Paket Controllers 4
6.2 Paket Precalculation 8
6.3 Paket RouteCalculator 11
6.4 Paket MapDisplay 13
6.5 Paket Models 16
6.6 Paket Views 18
6.7 Paket Profiles 20
6.8 Paket Map 21
6.9 Paket Exporter 27
6.10 Paket History 28
6.11 Paket Util 29

7 Sequenzdiagramme **31**
7.1 Programmstart 31
7.2 Rendern 32
7.3 Routenberechnung 32
7.4 Profilverwaltung 34
7.5 Vorberechnung 35

8 Glossar **36**

1 Einleitung

Dieses Dokument erläutert den Entwurf der Anwendung *routeKIT*. Es beschreibt ausführlich die verwendeten Pakete, Klassen und Methoden und ihre Beziehungen untereinander (wobei die Beziehungen im Klassendiagramm deutlicher ersichtlich sind).

routeKIT ist eine Anwendung zur Routenplanung; durch Verwendung von ↗Profilen kann sie dem Benutzer die optimalen ↗Routen für sein spezielles Fahrzeug angeben. Um die Routenberechnung zu beschleunigen, wird pro ↗Profil und ↗Karte eine zeitaufwändige ↗Vorberechnung durchgeführt.

2 Übersicht

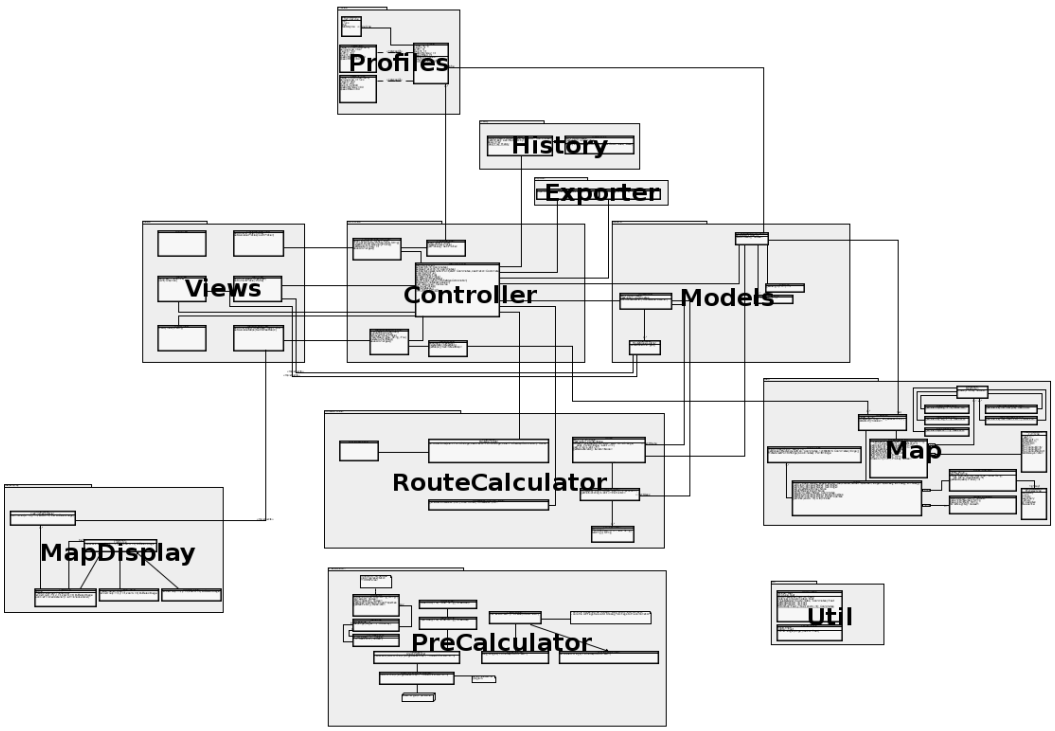

Abbildung 1: Paketdiagramm

Beim Entwurf der Anwendung *routeKIT* wurden ↗Entwurfsmuster eingesetzt. Der grobe Entwurf der Anwendung wird durch das Muster ↗Model View Controller (MVC) vorgegeben, welches in den Paketen ↗`Models`, ↗`Views` und ↗`Controllers` umgesetzt ist. Da bei ↗MVC die Verteilung von Zuständigkeiten zwischen den drei Teilen nicht vollständig festgelegt und auch durchaus umstritten ist, sei hier unsere Variante von ↗MVC erläutert:

Controller Der Controller behandelt die Benutzeraktionen, die den Zustand der Anwendung ändern, und sendet entsprechende Befehle an das Model und die View.

Model Das Model benachrichtigt über das Entwurfsmuster ↗Beobachter die View und den Controller bei Änderungen der Daten.

View Die View bekommt Informationen aus dem Model und sendet Benutzeraktionen, die den Zustand der Anwendung ändern, an den Controller. Benutzeraktionen, die den Zustand der Anwendung *nicht* ändern, werden rein in der View behandelt.

Benutzeraktionen, die den Zustand ändern, sind etwa das Öffnen des Profilmanagers (siehe ↗Abbildung 16) oder die Eingabe eines Startpunkts; Benutzeraktionen, die den Zustand *nicht* ändern, sind etwa das Scrollen im Verlauf oder Navigation auf der Karte – wobei letztere das Anfordern neuer Kartenkacheln aus dem Model auslöst.

Die Klassen ↗`MainController`, ↗`ProfileManager`, ↗`MapManager` im Paket ↗`Controllers` basieren auf dem ↗Entwurfsmuster ↗Einzelstück.

Im Paket ↗`MapDisplay` wird das ↗Entwurfsmuster ↗Dekorierer verwendet. Der Cache baut dabei eine Kapselung für einen effizienteren Zugriff um eine Kachelquelle herum. Dadurch wird die Funktionalität, dass ältere Zugriffe zwischengespeichert werden, von der Funktion, die Kacheln zu beschaffen, abgetrennt. Es wird außerdem das ↗Entwurfsmuster ↗Beobachter verwendet, wobei ↗`TileFinishedListener` der Beobachter ist. Die genauere Funktionsweise wird in ↗`MapDisplay` erläutert.

Im Paket ↗`RouteCalculator` wird das ↗Entwurfsmuster „↗Strategie“ eingesetzt. Der Algorithmus zur Berechnung der ↗Route ist in eine eigene Klasse ↗`ArcFlagsDijkstra` gekapselt und somit leicht zu ersetzen. Im Paket ↗`Precalculation` wird ebenfalls dieses Muster verwendet. Somit wird ermöglicht das Programm zum Partitionieren eines Graphen auszutauschen.

3 Anmerkungen

- Für Attribute, die im Klassendiagramm durch Relationen oder explizit angegeben sind, sind implizite Getter und Setter gegeben.
- Die folgenden Typen aus dem Klassendiagramm sind „typedefs“, keine echten Klassen:
  - `Node`: `int`, die ID des Knotens
  - `Edge`: `int`, die ID der Kante
  - `Turn`: `int`, die ID der Kante im kantenbasierten Graphen (↗`EdgeBasedGraph`)
  - `Partition`: `int`, die ID einer Partition

4 Änderungen gegenüber dem Pflichtenheft

- Es findet keine Angabe darüber statt, wie lange eine Vorberechnung voraussichtlich dauern wird, da dies kaum abschätzbar ist; stattdessen wird einfach darauf hingewiesen, dass die Vorberechnung mehrere Stunden dauern kann. (Pflichtenheft Abschnitte 9.2, 9.5)

5 Legende

5.1 Paket BeispielPaket

Dies ist ein Paket. Es enthält eine kurze Beschreibung und dann mehrere Klassen.

`BeispielKlasse` Dies ist eine Klasse. Sie kann Attribute und Methoden enthalten.

Attribute

`beispielAttribut` Dies ist ein Attribut. Typ: `Typ`

Methoden

`beispielMethode` Dies ist eine Methode. Sie kann mehrere Parameter enthalten und einen Wert eines bestimmten Typs zurückgeben.
Parameter:

`beispielParameter` Dies ist ein Parameter der Methode. Typ: `Typ`

Rückgabetyp: `Rückgabetyp`

Bezeichner und weiterer Code sind in `dicktengleicher Schrift` gesetzt.

6 Pakete und Klassen

6.1 Paket Controllers

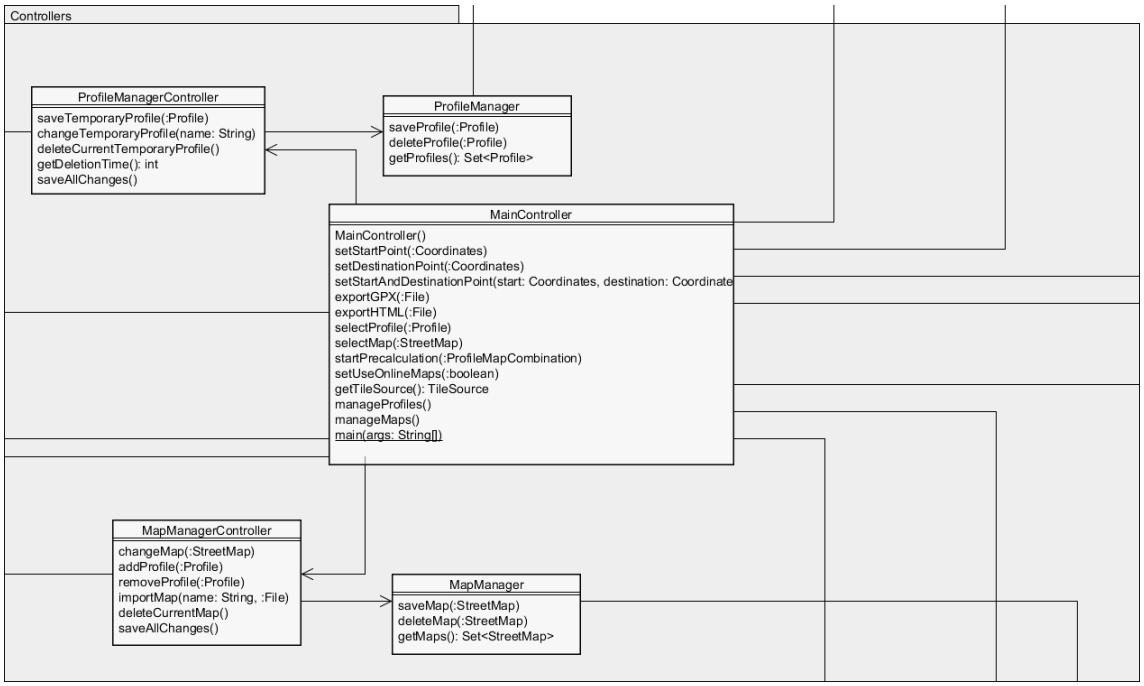

Abbildung 2: Das Paket Controllers

Dieses Paket enthält die Controller aus der ↗MVC-Architektur. Die Controller übernehmen die Steuerung des Programms und regeln die Kommunikation unter den einzelnen Komponenten. Der ↗`MainController` nimmt Benutzeraktionen von der ↗`MainView` und ↗`MapView` entgegen und reagiert darauf.

Der ↗`MainController` erstellt den ↗`ProfileManagerController` bzw. den ↗`MapManagerController`, wenn der Benutzer das Fenster der ↗Profil- bzw. Kartenverwaltung öffnet. Diese sind für ↗`ProfileManagerView` bzw. die ↗`MapManagerView` verantwortlich.

Der ↗`ProfileManager` und der ↗`MapManager` sind für das Speichern und Laden der ↗Profil- oder Kartendaten zuständig.

`MainController` Der Haupt-Controller von *routeKIT*. Er wird beim Programmstart erstellt und erstellt dabei die ↗`MainView`. Er verwaltet den gesamtem Programmablauf und bleibt so lange bestehen, bis *routeKIT* beendet wird.

Methoden

`MainController` Konstruktor: Erstellt den Controller, lädt ↗Profile, die Namen der ↗Karten und die aktuelle ↗Karte vollständig und erstellt dann die ↗`MainView`. Für den genauen Ablauf siehe ↗Abbildung 13.

`setStartPoint` Wird aufgerufen, wenn sich der Startpunkt ändert (z. B. durch eine Eingabe des Benutzers). Setzt den neuen Startpunkt in der ↗`MapView`. Falls bereits ein Zielpunkt ausgewählt ist, wird außerdem ein neuer Eintrag zum ↗Verlauf hinzugefügt (siehe ↗`History.addEntry`) und die Routenberechnung gestartet.
Parameter:

`start` Die Koordinaten des neuen Startpunkts. Typ: ↗`Coordinates`

`setDestinationPoint` Wird aufgerufen, wenn sich der Zielpunkt ändert (z. B. durch eine Eingabe des Benutzers). Setzt den neuen Zielpunkt in der ↗`MapView`. Falls bereits ein Startpunkt ausgewählt ist, wird außerdem ein neuer Eintrag zum ↗Verlauf hinzugefügt (siehe ↗`History.addEntry`) und die Routenberechnung gestartet.
Parameter:

`destination` Die Koordinaten des neuen Zielpunkts. Typ: ↗`Coordinates`

`setStartAndDestinationPoint` Wird aufgerufen, wenn sich der Start- und Zielpunkt ändern (z. B. durch die Auswahl eines Eintrags aus dem ↗Verlauf). Die gleichen Aktionen wie für ↗`setStartPoint` und ↗`setDestinationPoint` werden ausgeführt, nur nicht doppelt.
Parameter:

`start` Die Koordinaten des neuen Startpunkts. Typ: ↗`Coordinates`

`destination` Die Koordinaten des neuen Zielpunkts. Typ: ↗`Coordinates`

`exportGPX` Speichert die aktuelle ↗Route im ↗GPS Exchange Format (GPX)-Format in die angegebene Datei. Ist keine aktuelle ↗Route verfügbar (z. B. da noch keine ↗Vorberechnung vorliegt), so wird eine `IllegalStateException` geworfen.
Parameter:

`target` Die Datei, in die die ↗Route gespeichert werden soll. Typ: `File`

`exportHTML` Speichert die ↗Wegbeschreibung der aktuellen ↗Route im HTML-Format in die angegeben Datei. Ist keine aktuelle ↗Route verfügbar (z. B. da noch keine ↗Vorberechnung vorliegt), so wird eine `IllegalStateException` geworfen.
Parameter:

`target` Die Datei, in die die ↗Wegbeschreibung gespeichert werden soll. Typ: `File`

`selectProfile` Wählt das angegebene ↗Profil aus.
Parameter:

`profile` Das aktuelle ↗Profil. Typ: ↗`Profile`

`selectMap` Wählt die angegebene ↗Karte aus.
Parameter:

`map` Die aktuelle ↗Karte. Typ: ↗`StreetMap`

`startPrecalculation` Ruft in einem neuen WorkerThread ↗`PreCalculator.doPrecalculation` auf, falls keine ↗Vorberechnung für

diese Kombination aus ↗Profil und ↗Karte existiert. Sperrt währenddessen die ↗`MainView`.
Parameter:
`combination` Eine nicht vorberechnete Kombination aus ↗Profil und ↗Karte. Typ: ↗`ProfileMapCombination`

`setUseOnlineMaps` Legt fest, ob ↗OSM-Kacheln oder selbst gerenderte Kacheln verwendet werden sollen. Für ↗OSM-Kacheln wird der ↗`OSMRenderer` verwendet, für die eigenen Kacheln der ↗`TileRenderer`.
Parameter:
`useOnlineMaps` `true`, um ↗OSM-Kacheln zu verwenden, `false`, um selbst gerenderte Kacheln zu verwenden. Typ: `boolean`

`getTileSource` Gibt eine ↗`TileSource` zurück, die zum ↗Rendern der ↗Karten verwendet werden soll. Rückgabetyp: ↗`TileSource`

`manageProfiles` Startet einen neuen ↗`ProfileManagerController` und öffnet so den Dialog zur Profilverwaltung.

`manageMaps` Startet einen neuen ↗`MapManagerController` und öffnet so den Dialog zur Kartenverwaltung.

`main` (statisch) Hauptmethode des Programms. Erzeugt einen ↗`MainController`.
Parameter:
`args` Kommandozeilen-Argumente. Typ: `String[]`

`ProfileManagerController` Der Controller für die ↗`ProfileManagerView`.

Ein Beispiel für die Kommunikation zwischen den beiden Klassen ist in ↗Abbildung 16 zu sehen.

Methoden

`saveTemporaryProfile` Speichert die Werte des temporären ↗Profils. Wird üblicherweise direkt vor ↗`changeTemporaryProfile` aufgerufen.
Parameter:
`profile` Das temporäre ↗Profil mit den aktuell eingegebenen Werten. Typ: ↗`Profile`

`changeTemporaryProfile` Wechselt zu dem temporären ↗Profil mit dem angegebenen Namen. Falls noch kein ↗Profil mit diesem Namen existiert, wird es als Kopie des aktuellen ↗Profils erstellt.
Die Änderung wird der View über ↗`ProfileManagerView.setCurrentProfile` mitgeteilt.
Parameter:
`name` Der Name des neuen ↗Profils. Typ: `String`

`deleteCurrentTemporaryProfile` Markiert das aktuell ausgewählte ↗Profil zur Löschung und entfernt es aus der Auswahlliste.
Beachte: Das ↗Profil wird erst in ↗`saveAllChanges` tatsächlich gelöscht.
Handelt es sich bei dem aktuell ausgewählten ↗Profil um ein Standardprofil, so wird eine `IllegalStateException` geworfen.

`getDeletionTime` Gibt zurück, wie viel Zeit die ↗Vorberechnungen benötigt haben, die durch die aktuell erfassten Änderungen gelöscht werden. Die Dauer wird in Millisekunden zurückgegeben. Rückgabetyp: `int`

`saveAllChanges` Führt alle vom Benutzer vorgenommenen Änderungen aus. Dazu gehören das Hinzufügen, Ändern und Löschen von ↗Profilen. Für geänderte ↗Profile werden alle ↗Vorberechnungen gelöscht.

`MapManagerController` Der Controller für die ↗`MapManagerView`.

Methoden

`changeMap` Wird aufgerufen, wenn in der ↗`MapManagerView` eine andere ↗Karte ausgewählt wird. Speichert die Liste der ↗Profile für diese ↗Karte und setzt sie auf die Liste der neuen ↗Karte (ggf. die bereits gespeicherte Liste, falls die ↗Karte schon zuvor einmal ausgewählt war). Aktiviert/Deaktiviert den Löschen-Button, je nachdem, ob die neue ↗Karte eine Standardkarte ist oder nicht.
Parameter:

`map` Die neue ↗Karte. Typ: ↗`StreetMap`

`addProfile` Fügt das angegebene ↗Profil zur ausgewählten ↗Karte hinzu.
Parameter:

`profile` Das neue ↗Profil. Typ: ↗`Profile`

`removeProfile` Entfernt das angegebene ↗Profil von der ausgewählten ↗Karte.
Parameter:

`profile` Das ↗Profil, das entfernt werden soll. Typ: ↗`Profile`

`importMap` Fügt eine neue ↗Karte mit dem angegebenen Namen hinzu (oder ersetzt eine bestehende mit diesem Namen) und wählt sie aus.
Beachte: Die ↗Graphical User Interface (GUI)-Aktionen „Importieren“ und „Aktualisieren“ werden beide durch diese Methode implementiert; bei „Importieren“ stellt die ↗GUI sicher, dass kein bereits existierender Name gewählt wird, bei „Aktualisieren“ verwendet sie den Namen der existierenden ↗Karte.
Parameter:

`name` Der Name der neuen ↗Karte. Typ: `String`

`file` Die Datei aus der sie geladen werden soll. Typ: `File`

`deleteCurrentMap` Markiert die aktuell ausgewählte ↗Karte zur Löschung und entfernt sie aus der Auswahlliste.
Beachte: Die ↗Karte wird erst in ↗`saveAllChanges` tatsächlich gelöscht.
Handelt es sich bei der aktuell ausgewählten ↗Karte um eine Standardkarte, so wird eine `IllegalStateException` geworfen.

`saveAllChanges` Führt alle vom Benutzer vorgenommenen Änderungen aus. Dazu gehören das Importieren und Löschen von ↗Karten sowie das Hinzufügen oder Löschen von ↗Profilen je ↗Karte (Löscht also ↗Vorberechnungen oder erzeugt neue).

`ProfileManager` Verwaltet die ↗Profile. Hat intern eine Menge von vorhandenen ↗Profilen.

Methoden

`saveProfile` Speichert das ausgewählte ↗Profil in der internen Liste und auf der Festplatte.
(Der Speicherort wird vom Manager ↗deckend verwaltet.)
Parameter:

`profile` Das ↗Profil, das gespeichert werden soll. Typ: ↗`Profile`

`deleteProfile` Löscht das ausgewählte ↗Profil aus der internen Liste und von der Festplatte.
Parameter:

`profile` Das ↗Profil, das gelöscht werden soll. Typ: ↗`Profile`

`getProfiles` Gibt alle ↗Profile in der internen Liste zurück. Rückgabetyp: `Set<Profile>`

`MapManager` Verwaltet die Kartendaten. Hat intern eine Menge von vorhandenen Kartendaten.

Methoden

`saveMap` Speichert die ausgewählte ↗Karte in der internen Liste und auf der Festplatte.
(Der Speicherort wird vom Manager ↗deckend verwaltet.)
Parameter:

`map` Die ↗Karte, die gespeichert werden soll. Typ: ↗`StreetMap`

`deleteMap` Löscht die ausgewählte ↗Karte aus der internen Liste und von der Festplatte (die zwei Graphen, Daten über Beschränkungen und alle ↗Vorberechnungen).
Parameter:

`map` Die ↗Karte, die gelöscht werden soll. Typ: ↗`StreetMap`

`getMaps` Gibt alle ↗Karten in der internen Liste zurück. Rückgabetyp: `Set<StreetMap>`

6.2 Paket Precalculation

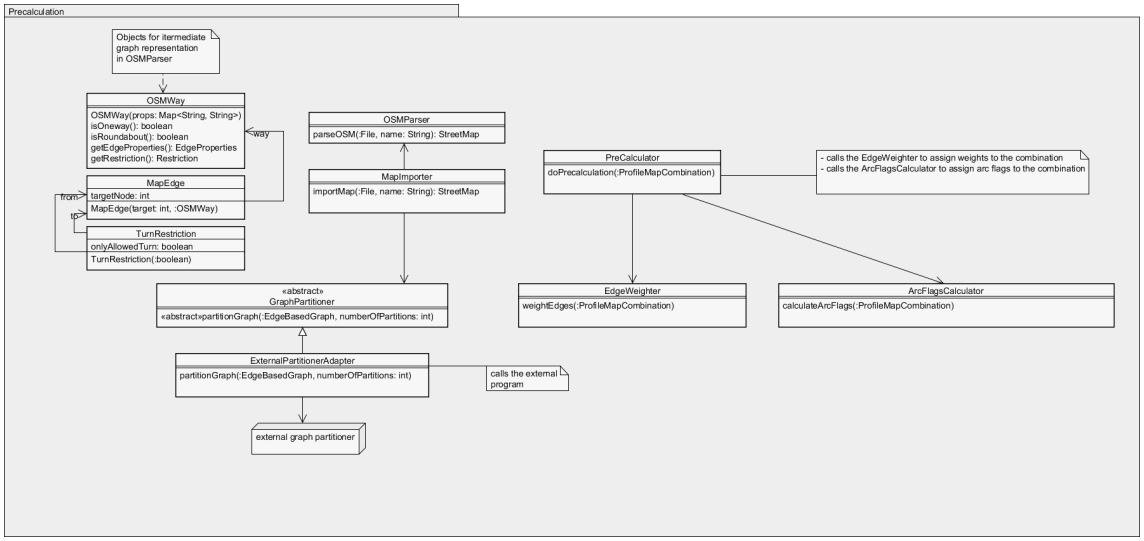

Abbildung 3: Das Paket Precalculation

Dieses Paket enthält alle Klassen, die an der ↗Vorberechnung beteiligt sind oder für den Import einer ↗Karte benötigt werden. Beim Importieren einer ↗Karte werden ein knotenbasierter und ein kantenbasierter Graph aufgebaut und der kantenbasierte Graph partitioniert. Bei der ↗Vorberechnung wird der kantenbasierte Graph vom ↗`EdgeWeighter` gewichtet und danach vom ↗`ArcFlagsCalculator` mit ↗Arc-Flags versehen.

Ein beispielhafter Ablauf der Vorberechnung ist in ↗Abbildung 17 zu sehen.

`MapImporter` Stellt die Funktionalität zum Importieren einer neuen ↗Karte bereit.

Methoden

`importMap` Importiert eine neue ↗Karte aus der angegebenen ↗OpenStreetMap (OSM)-Datei. Die vom ↗`OSMParser` aufgebaute Graphdatenstruktur wird dabei vom ↗`GraphPartitioner` partitioniert und zurückgegeben.
Parameter:
`file` Die ↗OSM-Datei, aus der die Kartendaten importiert werden sollen. Typ: `File`
`name` Der Name der neuen ↗`StreetMap`. Typ: `String`
Rückgabetyp: ↗`StreetMap`

`OSMParser` Stellt die Funktionalität zum Parsen einer ↗OSM-Datei bereit. Dafür werden zur temporären Repräsentation des Graphen im Speicher die Klassen ↗`OSMWay`, ↗`MapEdge` und ↗`TurnRestriction` verwendet.

Methoden

`parseOSM` Liest eine ↗OSM-Datei ein und erzeugt daraus einen ↗`Graph` sowie den zugehörigen (unpartitionierten) ↗`EdgeBasedGraph` und gibt diese als ↗`StreetMap` zurück.
Parameter:
`file` Die ↗OSM-Datei, die eingelesen werden soll. Typ: `File`
`name` Der Name der neuen ↗`StreetMap`. Typ: `String`
Rückgabetyp: ↗`StreetMap`

`OSMWay` Stellt einen Weg aus der OSM-Datei dar. Dies ist nur eine vom ↗`OSMParser` verwendete Hilfsklasse.

Methoden

`OSMWay` Konstruktor: Erzeugt ein neues Objekt aus den angegebenen OSM-Tags.
Parameter:
`props` Eine Liste von OSM-Tags. Typ: `Map<String, String>`
`isOneway` Bestimmt, ob es sich um eine Einbahnstraße handelt. Rückgabetyp: `boolean`
`isRoundabout` Bestimmt, ob es sich um einen Kreisverkehr handelt. Rückgabetyp: `boolean`
`getEdgeProperties` Liefert ein ↗`EdgeProperties`-Objekt mit den Eigenschaften des Wegs. Rückgabetyp: ↗`EdgeProperties`
`getRestriction` Liefert ein ↗`Restriction`-Objekt mit der/den Beschränkung(en) des Wegs oder `null`, falls nicht vorhanden. Rückgabetyp: ↗`Restriction`

`MapEdge` Stellt eine Kante im Straßengraphen dar. Dies ist nur eine vom ↗`OSMParser` verwendete Zwischendarstellung.

Attribute

`targetNode` Die ID des Zielknotens in der OSM-Datei. Typ: `int`

Methoden

`MapEdge` Konstruktor: Erzeugt ein neues Objekt mit den angegebenen Attributen.
Parameter:
`target` Die ID des Zielknotens. Typ: `int`
`way` Der zur Kante gehörige OSM-Weg. Typ: ↗`OSMWay`

`TurnRestriction` Stellt eine Abbiegebeschränkung dar. Dies ist nur eine vom ↗`OSMParser` verwendete Zwischendarstellung.

Attribute

`onlyAllowedTurn` Gibt an, ob es sich um die ausschließlich erlaubte Abbiegemöglichkeit handelt (`true`) oder um ein Abbiegeverbot (`false`). Typ: `boolean`

Methoden

`TurnRestriction` Konstruktor: Erzeugt ein neues Objekt mit dem angegebenen Attribut.
Parameter:

`onlyAllowedTurn` Der Wert des gleichnamigen Attributs. Typ: `boolean`

`GraphPartitioner` Partitioniert einen gegebenen Graphen.

Methoden

`partitionGraph` Teilt den Graphen in die gewünschte Anzahl an Partitionen. Die ermittelten Partitionen werden über ↗`EdgeBasedGraph.setPartitions` direkt gesetzt.
Parameter:

`graph` Der zu partitionierende Graph. Typ: ↗`EdgeBasedGraph`

`numberOfPartitions` Die gewünschte Anzahl an Partitionen. Typ: `int`

`ExternalPartitionerAdapter` Leitet die Partitionierungsanfrage an ein externes Partitionierungsprogramm weiter.

Methoden

`partitionGraph` Lässt den Graphen durch das externe Programm in die gewünschte Anzahl an Partitionen teilen.
Parameter:

`graph` Der zu partitionierende Graph. Typ: ↗`EdgeBasedGraph`

`numberOfPartitions` Die gewünschte Anzahl an Partitionen. Typ: `int`

`PreCalculator` Führt die ↗Vorberechnung für eine Kombination aus ↗`Profil` und ↗`Karte` durch.

Methoden

`doPrecalculation` Führt die ↗Vorberechnung für die gegebene Kombination aus ↗`Profil` und ↗`Karte` durch. Dabei werden ein ↗`EdgeWeighter` und ein ↗`ArcFlagsCalculator` aufgerufen. Die benötigte Zeit wird in ↗`ProfileMapCombination.calculationTime` gespeichert.
Parameter:

`comb` Die Kombination aus ↗`Profil` und ↗`Karte`, für die die ↗Vorberechnung durchgeführt werden soll. Typ: ↗`ProfileMapCombination`

`EdgeWeighter` Versieht den kantenbasierten Graphen (↗`EdgeBasedGraph`) mit Kantengewichten.

Die Kantengewichte des kantenbasierten Graphen sind profilabhängig und geben an, wie „teuer“ ein bestimmter Abbiegevorgang ist – Kanten mit niedrigerem Gewicht werden bei der Routenberechnung bevorzugt gewählt. Abbiegebeschränkungen werden durch maximale Kantengewichte umgesetzt.

Methoden

`weightEdges` Berechnet die Kantengewichte für die angegebene Kombination und setzt die ↗`Weights` von ↗`combination` entsprechend.
Parameter:

`combination` Die zu gewichtende Kombination aus ↗Profil und ↗Karte. Typ: ↗`ProfileMapCombination`

`ArcFlagsCalculator` Berechnet ↗Arc-Flags für einen partitionierten, gewichteten Graphen.

Methoden

`calculateArcFlags` Berechnet die ↗Arc-Flags für die angegebene Kombination und setzt die ↗`ArcFlags` von ↗`combination` entsprechend.
Parameter:

`combination` Die Kombination aus ↗Profil und ↗Karte. Typ: ↗`ProfileMapCombination`

6.3 Paket RouteCalculator

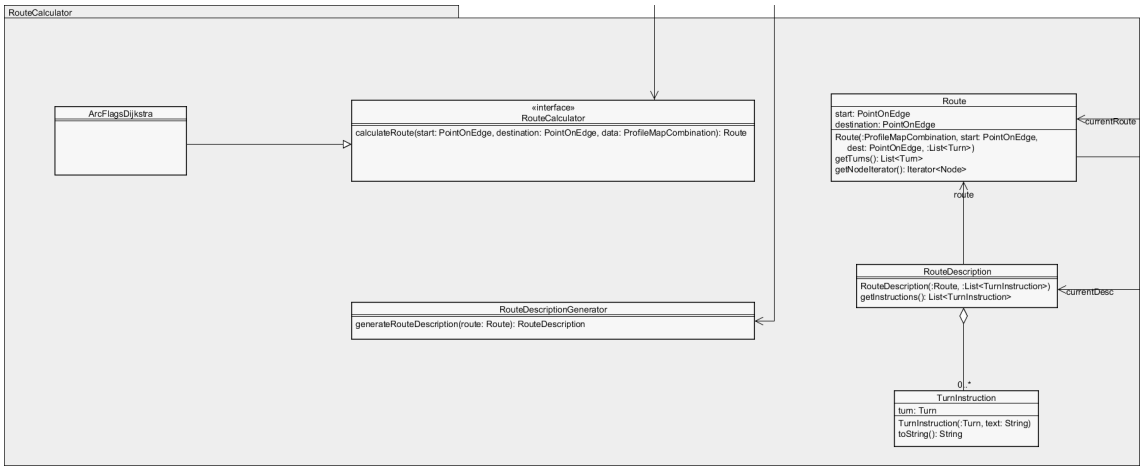

Abbildung 4: Das Paket RouteCalculator

Dieses Paket enthält alle Klassen, die zur Routenberechnung unter Berücksichtigung von ↗Arc-Flags und Erzeugung der ↗Wegbeschreibung benötigt werden. Die Klasse ↗`ArcFlagsDijkstra`, die das Interface ↗`RouteCalculator` implementiert, erstellt eine ↗`Route`. Zu dieser ↗`Route` erstellt ↗`RouteDescriptionGenerator` eine ↗`RouteDescription`.
Ein beispielhafter Ablauf der Routenberechnung ist in ↗Abbildung 15 zu sehen.

`RouteCalculator` Stellt ein Interface für einen Algorithmus zur Routenberechnung bereit.

Methoden

`calculateRoute` Berechnet einen Weg vom Startpunkt zum Zielpunkt auf dem gegebenen Graphen.
Parameter:

`start` Der Startpunkt für die Routenberechnung. Typ: ↗`PointOnEdge`

`destination` Der Zielpunkt für die Routenberechnung. Typ: ↗`PointOnEdge`

`data` Der vorberechnete Graph auf dem die Routenberechnung durchgeführt wird. Typ: ↗`ProfileMapCombination`

Rückgabetyp: ↗`Route`

`ArcFlagsDijkstra` Verwendet ↗Dijkstra's Algorithmus, um die schnellste ↗Route zwischen Start- und Zielpunkt für die aktuelle Kombination aus ↗Karte und ↗Profil zu berechnen. Durch ↗Arc-Flags wird die Berechnung beschleunigt.

`Route` Repräsentiert eine berechnete ↗Route.

Attribute

`start` Der Startpunkt der ↗Route. Typ: ↗`PointOnEdge`

`destination` Der Zielpunkt der ↗Route. Typ: ↗`PointOnEdge`

Methoden

`Route` Konstruktor: Erzeugt ein neues Routen-Objekt mit den angegebenen Attributen.

Parameter:

`data` Die Karte, auf dem die Route berechnet wurde. Typ: ↗`ProfileMapCombination`

`start` Der Startpunkt der Route. Typ: ↗`PointOnEdge`

`destination` Der Zielpunkt der Route. Typ: ↗`PointOnEdge`

`turns` Die Liste der Abbiegevorgänge der Route. Typ: `List<Turn>`

`getTurns` Liefert eine Liste der Abbiegevorgänge, aus denen die ↗Route besteht. Rückgabetyp: `List<Turn>`

`getNodeIterator` Gibt einen Iterator über die Knoten (`Node`) der ↗Route einschließlich Start- und Zielpunkt zurück. Der Iterator ermittelt diese dynamisch aus der Liste der Abbiegevorgänge. Rückgabetyp: `Iterator<Node>`

`RouteDescriptionGenerator` Stellt die Funktionalität zur Erzeugung einer ↗Wegbeschreibung bereit.

Methoden

`generateRouteDescription` Erzeugt die zur ↗`route` gehörende ↗Wegbeschreibung.

Parameter:

`route` Die ↗Route, deren Beschreibung erzeugt werden soll. Typ: ↗`Route`

Rückgabetyp: ↗`RouteDescription`

`RouteDescription` Kapselt die zu einer ↗`Route` gehörende ↗Wegbeschreibung.

Methoden

`RouteDescription` Konstruktor: Erzeugt ein neues Objekt mit den angegebenen Parametern.

Parameter:

`route` Die Route, die die Wegbeschreibung beschreibt. Typ: ↗`Route`

`instructions` Eine Liste von Abbiegeanweisungen. Typ: `List<TurnInstruction>`

`getInstructions` Liefert eine Liste der Abbiegeanweisungen. Rückgabetyp: `List<TurnInstruction>`

`TurnInstruction` Kapselt eine einzelne Abbiegeanweisung, aus welchen eine ↗`RouteDescription` aufgebaut ist.

Attribute

`turn` Der Abbiegevorgang, den die Anweisung beschreibt. Typ: `Turn`

Methoden

`TurnInstruction` Konstruktor: Erzeugt ein neues Objekt mit den angegebenen Attributen.
Parameter:

`turn` Der Abbiegevorgang, den die Anweisung beschreibt. Typ: `Turn`

`text` Der Text der Abbiegeanweisung. Typ: `String`

`toString` Gibt den Text der Abbiegeanweisung zurück. Rückgabetyp: `String`

6.4 Paket MapDisplay

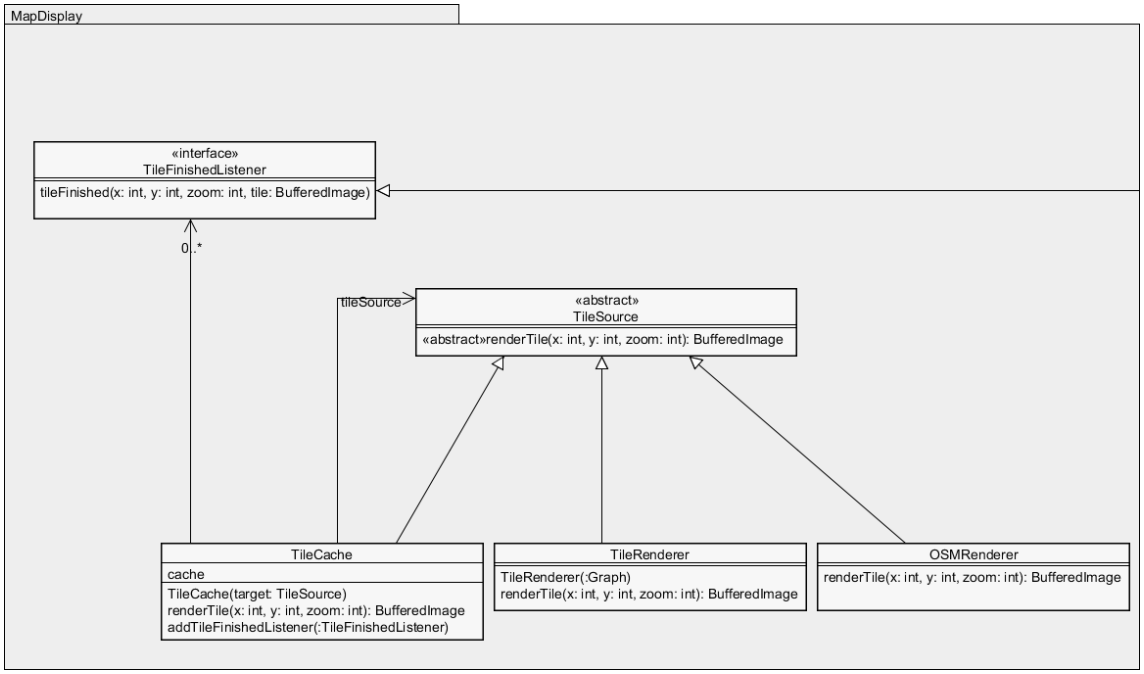

Abbildung 5: Das Paket MapDisplay

Dieses Paket enthält alle Klassen, die an der Darstellung der ↗Karte im ↗GUI beteiligt sind.

Die zentrale Klasse dieses Pakets ist der ↗`TileCache`, der dafür sorgt, dass Kartenkacheln nicht jedes Mal neu berechnet werden müssen. Er berechnet selbst keine Kacheln, sondern erhält diese von einer untergeordneten ↗`TileSource`, welche die Kacheln synchron berechnet. Der Cache stellt diese Kacheln dann asynchron bereit, indem er für nicht gespeicherte Kacheln zunächst eine „Dummy-Kachel“ zurückgibt und bei abgeschlossener Berechnung registrierte ↗`TileFinishedListener` benachrichtigt. Ein beispielhafter Ablauf ist in ↗Abbildung 14 zu sehen.

Um nicht zu viel Arbeitsspeicher zu beanspruchen, verwaltet der Cache die gespeicherten Kacheln so, dass sie bei Speicherknappheit vom Garbage Collector der JVM freigegeben werden können.

Kartenkacheln werden über ↗Slippy Map Tile-Koordinaten, wie sie auch bei anderen ↗OSM-Viewern verwendet werden, adressiert.

`TileSource` Abstrakte Klasse, die ein Interface für das (synchrone) ↗Rendern von Kartenkacheln definiert.

Methoden

`renderTile` Berechnet die angegebene Kachel und gibt sie zurück.

Parameter:

`x` siehe ↗`TileCache.renderTile.x` Typ: `int`

`y` siehe ↗`TileCache.renderTile.y` Typ: `int`

`zoom` siehe ↗`TileCache.renderTile.zoom` Typ: `int`

Rückgabetyp: `BufferedImage`

`TileCache` Verwaltet die Berechnung von Kartenkacheln und ist ein Zwischenspeicher für diese. Kacheln können angefragt werden, und nachdem die (asynchrone) Berechnung abgeschlossen ist, werden registrierte ↗`TileFinishedListener` benachrichtigt.

Intern werden die zwischengespeicherten Kacheln so gehalten, dass der Garbage Collector sie bei Speicherknappheit verwerfen kann (etwa durch `SoftReference`s).

Methoden

`TileCache` Konstruktor: Erstellt einen neuen Cache für die angegebene ↗`TileSource`.

Parameter:

`target` Die `TileSource`, die die tatsächliche Berechnung durchführt und deren Ergebnisse zwischengespeichert werden. Typ: ↗`TileSource`

`renderTile` Ist die angeforderte Kachel bereits im Zwischenspeicher vorhanden, so wird sie direkt zurückgegeben; andernfalls wird eine Dummy-Kachel zurückgegeben und die richtige von ↗`TileCache.target` angefordert, im Zwischenspeicher gespeichert und dann zurückgegeben. Kacheln von tieferer Zoomstufe und der Umgebung einer Kachel werden von ↗`TileCache.target` angefordert und im Zwischenspeicher gespeichert.

Parameter:

`x` Die ↗Slippy Map Tile (SMT)-X-Komponente. Typ: `int`

`y` Die ↗SMT-Y-Komponente. Typ: `int`

`zoom` Die Zoomstufe. Typ: `int`

Rückgabetyp: `BufferedImage`

`addTileFinishedListener` Registriert einen ↗`TileFinishedListener`, der benachrichtigt wird, wenn eine Kachel fertig berechnet ist. Die Kachel ist Teil der Nachricht.

Parameter:

`listener` Der Listener, der hinzugefügt werden soll. Typ: ↗`TileFinishedListener`

`TileFinishedListener` Wird benachrichtigt, wenn die Berechnung einer Kartenkachel abgeschlossen ist.

Methoden

`tileFinished` Wird vom ↗`TileCache` aufgerufen, wenn die Berechnung einer Kachel abgeschlossen ist. Die übliche Aktion ist, ein `repaint` der Kartenansicht im ↗GUI auszulösen.
Parameter:

`x` siehe ↗`TileCache.renderTile.x` Typ: `int`

`y` siehe ↗`TileCache.renderTile.y` Typ: `int`

`zoom` siehe ↗`TileCache.renderTile.zoom` Typ: `int`

`tile` Die berechnete Kachel. Typ: `BufferedImage`

`TileRenderer` Eine ↗`TileSource`, die die Kacheln selbst berechnet.

Methoden

`TileRenderer` Konstruktor: Erzeugt einen neuen `TileRenderer`.
Parameter:

`graph` Ein Adjazenzfeld. Typ: ↗`Graph`

`renderTile` Berechnet die angegebene Kachel und gibt sie zurück.
Parameter:

`x` siehe ↗`TileCache.renderTile.x` Typ: `int`

`y` siehe ↗`TileCache.renderTile.y` Typ: `int`

`zoom` siehe ↗`TileCache.renderTile.zoom` Typ: `int`

Rückgabetyp: `BufferedImage`

`OSMRenderer` Eine ↗`TileSource`, die die ↗OSM-Kacheln vom OpenStreetMap-Server herunterlädt.

Methoden

`renderTile` Lädt die angegebene Kachel herunter und gibt sie zurück.
Parameter:

`x` siehe ↗`TileCache.renderTile.x` Typ: `int`

`y` siehe ↗`TileCache.renderTile.y` Typ: `int`

`zoom` siehe ↗`TileCache.renderTile.zoom` Typ: `int`

Rückgabetyp: `BufferedImage`

6.5 Paket Models

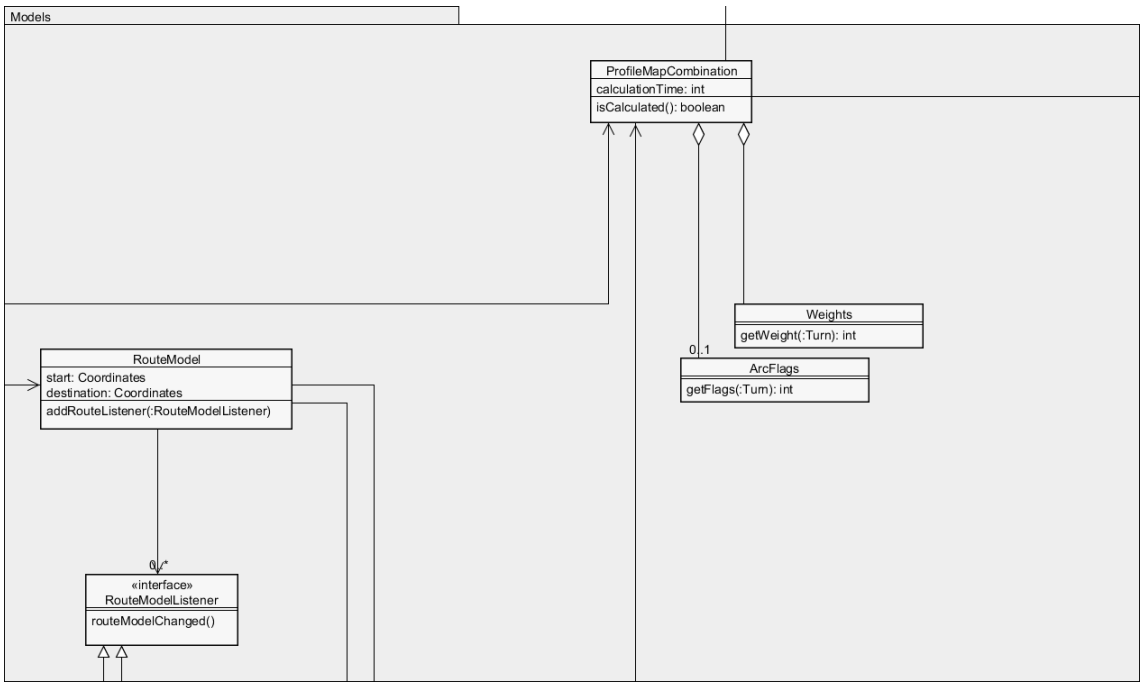

Abbildung 6: Das Paket Models

Dieses Paket enthält die Models aus der ↗MVC-Architektur.

`ProfileMapCombination` Eine Kombination aus einem ↗`Profil` und einer ↗`Karte`.

Attribute

`profile` Das ↗Profil. Typ: ↗`Profile`

`map` Die ↗Karte. Typ: ↗`StreetMap`

`calculationTime` Die Zeit, die für die Vorberechnung dieser Kombination benötigt wurde, in Millisekunden.
Für Kombinationen ohne ↗Vorberechnung ist dieser Wert `0`. Typ: `int`

Methoden

`isCalculated` Gibt `true` zurück, wenn für eine Kombination aus ↗Profil und ↗Karte eine ↗Vorberechnung der Gewichte und der ↗Arc-Flags existiert. Rückgabetyp: `boolean`

`ArcFlags` Enthält die ↗Arc-Flags für den vorberechneten Graphen.

Methoden

`getFlags` Gibt die zum Abbiegevorgang gehörenden ↗Arc-Flags zurück.
Parameter:
`turn` Die Nummer eines Abbiegevorgang. Typ: `Turn`
Rückgabetyp: `int`

`Weights` Enthält die Kantengewichte für den vorberechneten Graphen.

Methoden

`getWeight` Gibt das zum Abbiegevorgang gehörende Gewicht zurück.
Parameter:
`turn` Die Nummer eines Abbiegevorgang. Typ: `Turn`
Rückgabetyp: `int`

`RouteModel` Stellt die aktuellen Start- und Zielpunkte, sowie die aktuell berechnete ↗Route dar. Die Getter liefern dabei immer den aktuellen Zustand (auch `null` möglich). Die Setter ändern den Wert und informieren eventuelle `RouteListener`.

Methoden

`addRouteListener` Fügt einen RouteListener dem Modell hinzu, damit er über Änderungen an ↗Route, Start oder Ziel informiert wird
Parameter:
`listener` Der neue Listener, der über Änderungen informiert werden will. Typ: ↗`RouteModelListener`

`RouteModelListener` Wird bei Änderungen am ↗`RouteModel` informiert.

Methoden

`routeModelChanged` Wird bei jeder Änderung am ↗`RouteModel` aufgerufen.

6.6 Paket Views

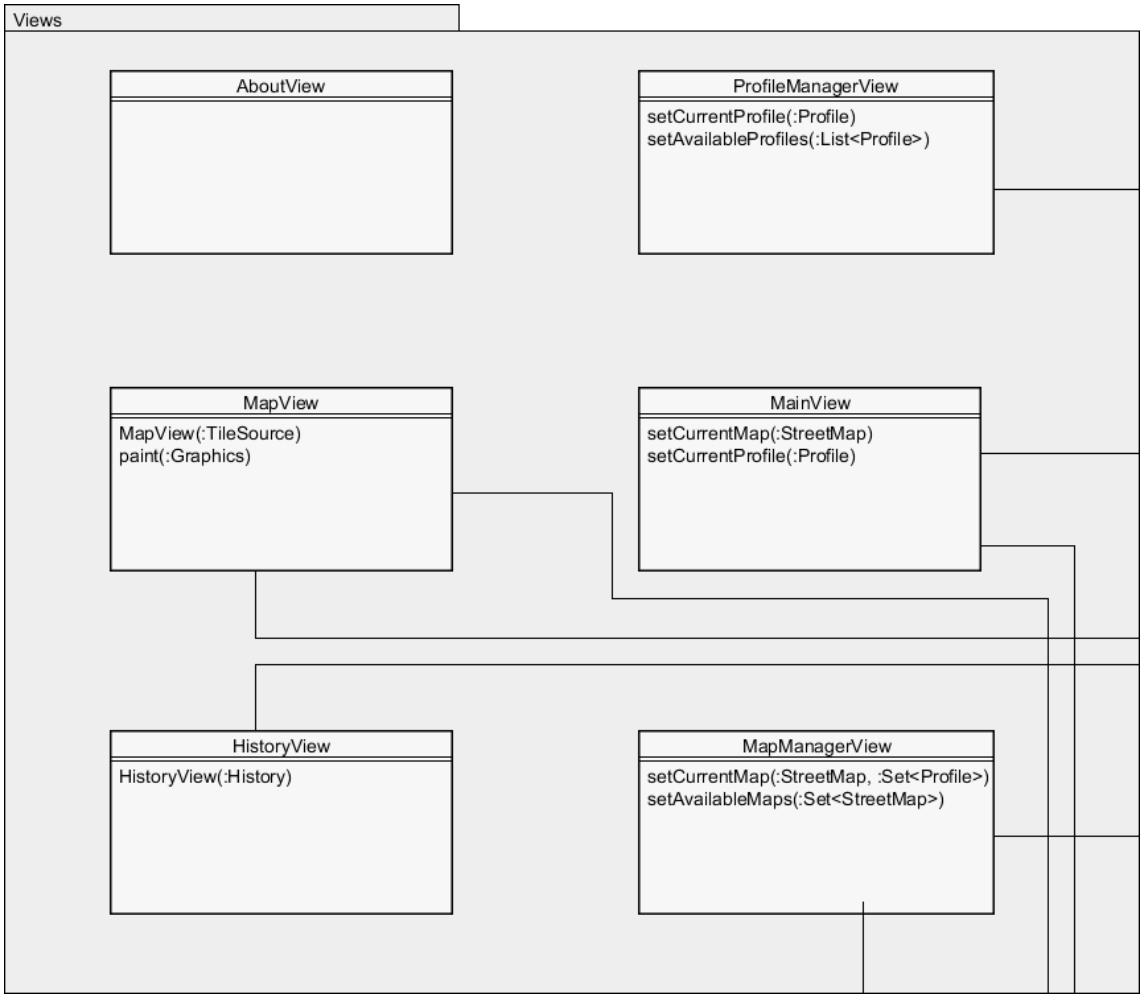

Abbildung 7: Das Paket Views

Dieses Paket enthält die Views nach der ↗MVC-Architektur. Alle Views im Klassendiagramm sind Interfaces, die von genau einer Klasse realisiert werden.

`MainView` Zeigt das Hauptfenster auf dem Bildschirm an.

Methoden

`setCurrentMap` Aktualisiert die Anzeige der aktuellen ↗Karte.
Parameter:
`map` Die neue ↗Karte. Typ: ↗`StreetMap`

`setCurrentProfile` Aktualisiert die Anzeige des aktuellen ↗Profils.
Parameter:
`profile` Das neue ↗Profil. Typ: ↗`Profile`

`MapView` Zeigt einen Kartenausschnitt auf dem Bildschirm an.

Als Kartenprojektion wird die ↗Mercator-Projektion verwendet.

Methoden

`MapView` Konstruktor: Erzeugt eine neue ↗`MapView`. Die angegebene ↗`TileSource` wird zum ↗Rendern verwendet.
Da die Kacheln bei jedem ↗`paint` synchron angefragt werden, sollte ↗`source` ein ↗`TileCache` sein.
Parameter:

`source` Ein Objekt, das die Kartenkacheln liefert, die dann angezeigt werden. Typ: ↗`TileSource`

`paint` Zeichnet den aktuell sichtbaren Kartenausschnitt. Alle sichtbaren Kacheln werden von ↗`MapView.source` synchron angefordert.
Parameter:

`graphics` Die Java `Graphics`, auf welche die ↗Karte gezeichnet wird. Typ: `Graphics`

`ProfileManagerView` Zeigt das Fenster der Profilverwaltung auf dem Bildschirm an.

Methoden

`setCurrentProfile` Setzt das aktuelle ↗Profil auf das angegebene ↗Profil, lädt seine Werte in die Eingabefelder und aktiviert/deaktiviert die Eingabeelemente, je nachdem, ob es sich um ein Standardprofil handelt oder nicht.
Parameter:

`profile` Das neue ↗Profil. Typ: ↗`Profile`

`setAvailableProfiles` Setzt die ↗Profile, die aktuell ausgewählt werden können.
Parameter:

`profiles` Die verfügbaren ↗Profile. Typ: `List<Profile>`

`MapManagerView` Zeigt das Fenster der Kartenverwaltung auf dem Bildschirm an.

Methoden

`setCurrentMap` Setzt die aktuelle ↗Karte auf die angegebene ↗Karte, aktualisiert die Liste der ↗Profile für die ausgewählte ↗Karte und aktiviert/deaktiviert die „Import“- und „Löschen“-Buttons, je nachdem, ob es sich um eine Standardkarte handelt oder nicht.
Parameter:

`map` Die neue ↗Karte. Typ: ↗`StreetMap`

`profiles` Die ↗Profile für die neue ↗Karte. Typ: `Set<Profile>`

`setAvailableMaps` Setzt die ↗Karten, die aktuell ausgewählt werden können.
Parameter:

`maps` Die verfügbaren ↗Karten. Typ: `Set<StreetMap>`

`HistoryView` Zeigt das Fenster mit dem ↗Verlauf auf dem Bildschirm an.

Methoden

`HistoryView` Konstruktor: Erstellt eine HistoryView für den angegebenen ↗Verlauf. (Der ↗Verlauf kann später nicht mehr geändert werden.)
Parameter:

`history` Der ↗Verlauf, der angezeigt wird. Typ: ↗`History`

`AboutView` Zeigt das Fenster mit den Informationen über *routeKIT* auf dem Bildschirm an.

6.7 Paket Profiles

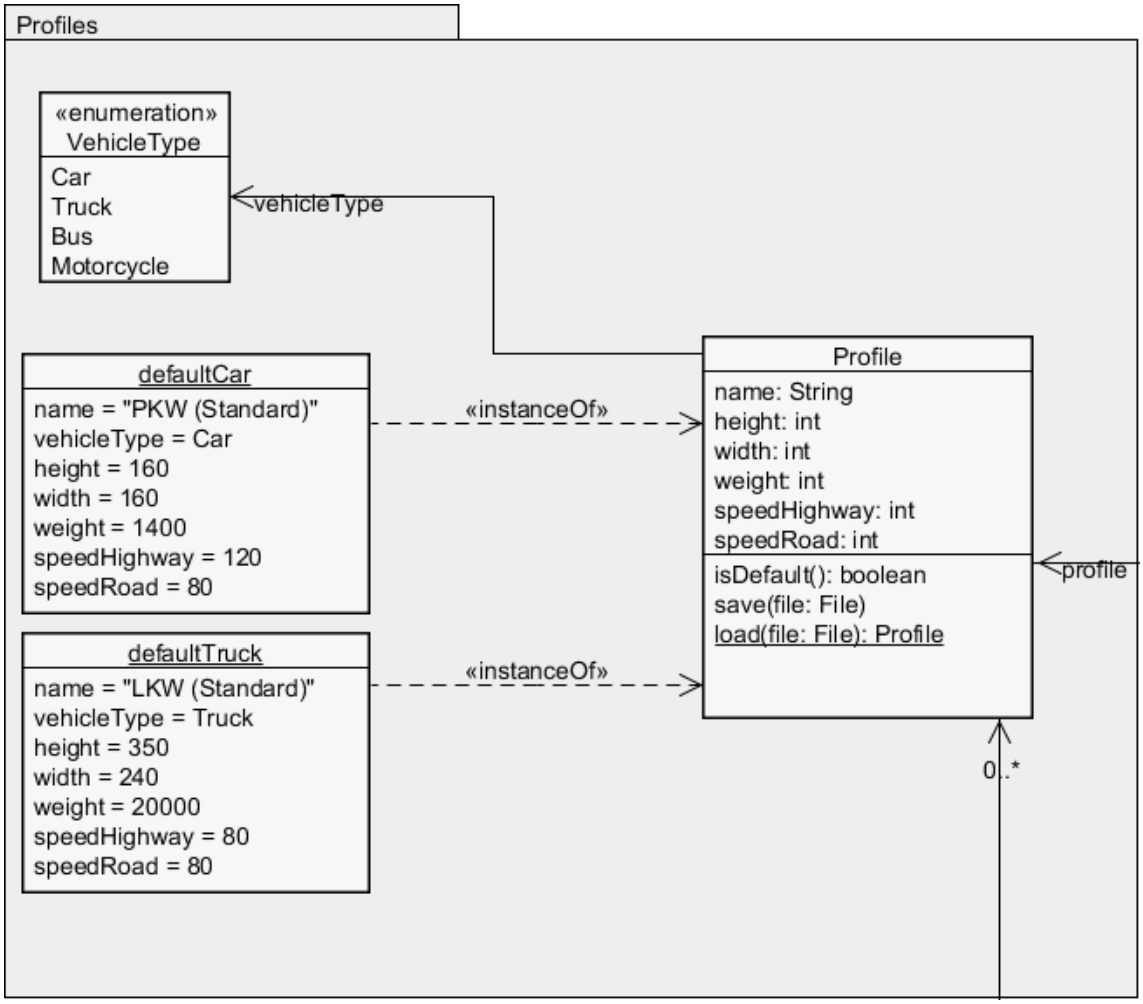

Abbildung 8: Das Paket Profiles

Dieses Paket enthält alle Klassen zu ↗Profilen.

`Profile` Ein ↗Fahrzeugprofil.

Attribute

`name` Der Name des ↗Profils. Typ: `String`

`vehicleType` Der Fahrzeugtyp. Typ: ↗`VehicleType`

`height` Die Höhe des Fahrzeugs, in Zentimetern. Typ: `int`

`width` Die Breite des Fahrzeugs, in Zentimetern. Typ: `int`

`weight` Das Gewicht des Fahrzeugs, in Kilogramm. Typ: `int`

`speedHighway` Die Durchschnittsgeschwindigkeit des Fahrzeugs auf der Autobahn, in Kilometern pro Stunde. Typ: `int`

`speedRoad` Die Durchschnittsgeschwindigkeit des Fahrzeugs auf der Landstraße, in Kilometern pro Stunde. Typ: `int`

Methoden

`isDefault` Gibt an, ob es sich um ein Standardprofil handelt oder nicht. Rückgabetyp: `boolean`

`save` Speichert das ↗Profil in die angegebene Datei.
Parameter:

`file` Die Datei, in die das ↗Profil gespeichert wird. Typ: `File`

`load` (statisch) Lädt ein ↗Profil aus der angegebenen Datei und gibt es zurück.
Parameter:

`file` Die Datei, aus der das ↗Profil geladen wird. Typ: `File`

Rückgabetyp: ↗`Profile`

`VehicleType` Ein Fahrzeugtyp. Instanzen:

`Car` Ein ↗Personenkraftwagen (PKW).

`Truck` Ein ↗Lastkraftwagen (LKW).

`Bus` Ein Omnibus.

`Motorcycle` Ein Motorrad.

6.8 Paket Map

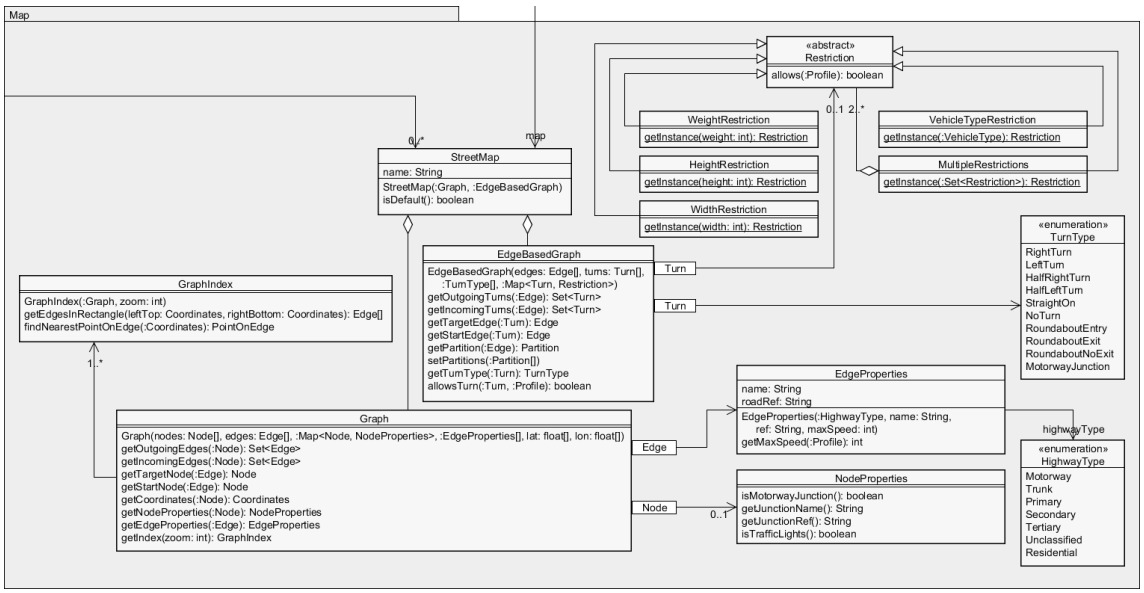

Abbildung 9: Das Paket Map

Dieses Paket enthält alle Klassen zu Karten und Kartengraphen.
routeKIT verwendet zwei Datenstrukturen für den Straßengraphen:

↗`Graph` ist ein regulärer, knotenbasierter Graph, gespeichert als Adjazenzfeld. Aus ihm werden geometrische Datenstrukturen ↗`GraphIndex` für verschiedene Zoomstufen erzeugt. Er wird zum ↗Rendern der ↗Karte verwendet.

↗`EdgeBasedGraph` ist ein kantenbasierter Graph (ebenfalls gespeichert als Adjazenzfeld). Er wird zur Routenberechnung verwendet.

Zusammen mit einem Namen ergeben sie ↗`StreetMap`.

`StreetMap` Eine ↗Karte.

Attribute

`name` Der Name der ↗Karte. Typ: `String`

`graph` Der Graph der ↗Karte, d. h. das Straßennetz. Typ: ↗`Graph`

`edgeBasedGraph` Die kantenbasierte Version des Straßennetzes, die zur Routenberechnung verwendet wird. Typ: ↗`EdgeBasedGraph`

Methoden

`StreetMap` Konstruktor: Erzeugt ein neues Objekt aus den gegebenen Graphen.
Parameter:

`graph` Der Kartengraph. Typ: ↗`Graph`

`edgeBasedGraph` Der kantenbasierte Graph. Typ: ↗`EdgeBasedGraph`

`isDefault` Gibt zurück, ob es sich um eine Standardkarte handelt. Rückgabetyp: `boolean`

`Graph` Ein Kartengraph/Straßennetz. Beachte: Dieser Graph ist nicht das Ergebnis einer ↗Vorberechnung für ein ↗Profil und eine ↗Karte, sondern nur für eine ↗Karte.

Attribute

`highwayType` Der Straßentyp. Typ: ↗`HighwayType`

Methoden

`Graph` Konstruktor: Erzeugt ein neues Graph-Objekt aus dem gegebenen Adjazenzfeld.
Parameter:

`nodes` Der Knoten-Bestandteil des Adjazenzfeldes. Typ: `Node[]`

`edges` Der Kanten-Bestandteil des Adjazenzfeldes. Typ: `Edge[]`

`nodeProps` Die `NodeProperties` der Knoten des Graphen. Es wird eine `Map` anstelle eines Arrays verwendet, da die meisten Knoten keine besonderen Eigenschaften haben und daher das Array zum großen Teil leer wäre. Typ: `Map<Node, NodeProperties>`

`edgeProps` Die `EdgeProperties` der Kanten des Graphen. Hier wird ein Array verwendet, da jede Kante einen Namen und damit ein `EdgeProperties`-Objekt hat. Typ: `EdgeProperties[]`

`lat` Die geographischen Breiten der Knoten des Graphen. Typ: `float[]`

`lon` Die geographischen Längen der Knoten des Graphen. Typ: `float[]`

`getOutgoingEdges` Gibt alle ausgehenden Kanten des angegebenen Knotens zurück.
Parameter:

`node` Der Knoten, dessen ausgehende Kanten gesucht werden. Typ: `Node`

Rückgabetyp: `Set<Edge>`

`getIncomingEdges` Gibt alle in den Knoten eingehende Kanten zurück.
Parameter:

`node` Der Knoten, dessen eingehende Kanten gesucht werden. Typ: `Node`

Rückgabetyp: `Set<Edge>`

`getTargetNode` Gibt den Endknoten der angegebenen Kante zurück.
Parameter:

`edge` Die Kante, dessen Endknoten gesucht wird. Typ: `Edge`

Rückgabetyp: `Node`

`getStartNode` Gibt den Startknoten der angegebenen Kante zurück.

Parameter:

`edge` Die Kante, dessen Startknoten gesucht wird. Typ: `Edge`

Rückgabetyp: `Node`

`getCoordinates` Gibt die Koordinaten des angegebenen Knotens zurück.

Parameter:

`node` Der Knoten, dessen Koordinaten gesucht werden. Typ: `Node`

Rückgabetyp: ↗`Coordinates`

`getNodeProperties` Gibt die `NodeProperties` des angegebenen Knotens zurück.

Parameter:

`node` Der Knoten, dessen ↗`NodeProperties` gesucht werden. Typ: `Node`

Rückgabetyp: ↗`NodeProperties`

`getEdgeProperties` Gibt die `EdgeProperties` der angegebenen Kante zurück.

Parameter:

`edge` Die Kante, deren ↗`EdgeProperties` gesucht werden. Typ: `Edge`

Rückgabetyp: ↗`EdgeProperties`

`getIndex` Gibt eine geometrische Datenstruktur zur angegebenen Zoomstufe zurück.

Parameter:

`zoom` Die Zoomstufe. Typ: `int`

Rückgabetyp: ↗`GraphIndex`

`GraphIndex` Eine geometrische Datenstruktur zum schnellen Auffinden von Kanten innerhalb eines Kartenausschnitts.

Methoden

`GraphIndex` Konstruktor: Erzeugt die Datenstruktur für den gegebenen Graph und die angegebene Zoomstufe.

Parameter:

`graph` Ein Graph. Typ: ↗`Graph`

`zoom` Die Zoomstufe. Typ: `int`

`getEdgesInRectangle` Bestimmt alle Kanten innerhalb eines rechteckigen Kartenausschnitts, der durch ↗`leftTop` und ↗`rightBottom` festgelegt ist.

Parameter:

`leftTop` Die Koordinaten der linken oberen Ecke des Ausschnitts. Typ: ↗`Coordinates`

`rightBottom` Die Koordinaten der rechten unteren Ecke des Ausschnitts. Typ: ↗`Coordinates`

Rückgabetyp: `Edge[]`

`findNearestPointOnEdge` Sucht zu gegebenen Koordinaten den nächsten Punkt auf einer Kante.

Parameter:

`coords` Die Koordinaten eines Punktes. Typ: ↗`Coordinates`

Rückgabetyp: ↗`PointOnEdge`

`NodeProperties` Kapselt die Eigenschaften eines Knotens.

Methoden

`isMotorwayJunction` Bestimmt, ob der Knoten eine Schnellstraßen- oder Autobahnanschlussstelle ist. Rückgabetyp: `boolean`

`getJunctionName` Gibt den Namen der Anschlussstelle zurück oder `null`, falls es sich nicht um eine Anschlussstelle handelt. Rückgabetyp: `String`

`getJunctionRef` Gibt die Nummer der Anschlussstelle zurück oder `null`, falls es sich nicht um eine Anschlussstelle handelt. Rückgabetyp: `String`

`isTrafficLights` Bestimmt, ob sich an dem Knoten eine Ampel befindet. Rückgabetyp: `boolean`

`EdgeProperties` Kapselt die Eigenschaften einer Kante.

Attribute

`name` Der Name der Straße oder `null`, falls nicht vorhanden. Typ: `String`

`roadRef` Die Nummer der Straße oder `null`, falls nicht vorhanden. Typ: `String`

Methoden

`EdgeProperties` Konstruktor: Erzeugt ein neues Objekt mit den angegebenen Eigenschaften.
Parameter:

`type` Der Straßentyp. Typ: ↗`HighwayType`

`name` Der Wert für ↗`name`. Typ: `String`

`roadRef` Der Wert für ↗`roadRef`. Typ: `String`

`maxSpeed` Die zulässige Höchstgeschwindigkeit für diese Kante oder `0`, falls nicht festgelegt. Typ: `int`

`getMaxSpeed` Bestimmt die zulässige Höchstgeschwindigkeit (in Kilometern pro Stunde) auf dieser Kante für das angegebene ↗Profil.
Parameter:

`profile` Das ↗Profil, für das die Höchstgeschwindigkeit auf dieser Kante bestimmt werden soll. Typ: ↗`Profile`

Rückgabetyp: `int`

`HighwayType` Ein Straßentyp. Instanzen:

`Motorway` Autobahn

`Trunk` Schnellstraße

`Primary` Bundesstraße

`Secondary` Landesstraße

`Tertiary` Kreisstraße

`Unclassified` Gemeindeverbindungsstraße

`Residential` Ortsstraße

`EdgeBasedGraph` Enthält das Straßennetz als kantenbasierten Graphen. Die Knoten dieses Graphen entsprechen den Kanten des zugehörigen ↗`Graph`-Objekts und werden daher mit `Edge` bezeichnet. Die Kanten dieses Graphen repräsentieren Abbiegemöglichkeiten und werden mit `Turn` bezeichnet.

Diese Datenstruktur ist unabhängig vom ↗Profil und wird wie ↗`Graph` bei der ↗Vorberechnung für eine ↗Karte erstellt. Erst in Kombination mit den profilspezifischen ↗`Weights` kann sie zur Routenberechnung verwendet werden.

Methoden

`EdgeBasedGraph` Konstruktor: Erzeugt ein neues Objekt aus dem gegebenen Adjazenzfeld.
Parameter:

`edges` Das Knoten-Array (Kanten im Straßengraph) des Adjazenzfelds. Typ: `Edge[]`

`turns` Das Kanten-Array (Abbiegemöglichkeiten) des Adjazenzfelds. Typ: `turn[]`

`turnTypes` Die Typen der Abbiegemöglichkeiten. Typ: `TurnType[]`

`restrictions` Die Beschränkungen der Abbiegemöglichkeiten. Typ: `Map<Turn, Restriction>`

`getOutgoingTurns` Gibt alle Abbiegemöglichkeiten *von* der angegebenen Kante zurück.
Parameter:

`edge` Die Kante, deren ausgehende Abbiegemöglichkeiten gesucht werden. Typ: `Edge`

Rückgabetyp: `Set<Turn>`

`getIncomingTurns` Gibt alle Abbiegemöglichkeiten *auf* die angegebene Kante zurück.
Parameter:

`edge` Die Kante, deren eingehende Abbiegemöglichkeiten gesucht werden. Typ: `Edge`

Rückgabetyp: `Set<Turn>`

`getTargetEdge` Gibt die Kante zurück, auf die die angegebene Abbiegemöglichkeit führt.
Parameter:

`turn` Die Abbiegemöglichkeit, deren Endkante gesucht wird. Typ: `Turn`

Rückgabetyp: `Edge`

`getStartEdge` Gibt die Kante zurück, von der die angegebene Abbiegemöglichkeit besteht.
Parameter:

`turn` Die Abbiegemöglichkeit, deren Anfangskante gesucht wird. Typ: `Turn`

Rückgabetyp: `Edge`

`getPartition` Gibt die Partition zurück, in der sich die angegebene Kante (der Knoten des kantenbasierten Graphen) befindet.
Ist noch keine Partitionierung gegeben, so wird immer eine Standard-Partition zurückgegeben.
Parameter:

`edge` Die Kante, deren Partition bestimmt werden soll. Typ: `Edge`

Rückgabetyp: `Partition`

`setPartitions` Setzt die Partitionen des Graphen. Die `Edge`s des Graphen sind die Indizes in ↗`partitions`.
Parameter:

`partitions` Die neuen Partitionen. Typ: `Partition[]`

`getTurnType` Gibt die Art des angegebenen Abbiegevorgangs zurück.
Parameter:

`turn` Der Abbiegevorgang, dessen Art gesucht wird. Typ: `Turn`

Rückgabetyp: ↗`TurnType`

`allowsTurn` Bestimmt, ob der angegebene Abbiegevorgang unter dem angegeben ↗Profil zulässig ist.
Parameter:

`turn` Der zu betrachtende Abbiegevorgang. Typ: `Turn`

`profile` Das verwendete ↗Profil. Typ: ↗`Profile`

Rückgabetyp: `boolean`

`TurnType` Der Typ einer Abbiegemöglichkeit. Instanzen:

`RightTurn` Rechts abbiegen.

`LeftTurn` Links abbiegen.

`HalfRightTurn` Rechts halten.

`HalfLeftTurn` Links halten.

`StraightOn` Geradeaus.

`NoTurn` Keine echte Abbiegemöglichkeit.

`RoundaboutEntry` Einfahrt in einen Kreisverkehr.

`RoundaboutExit` Ausfahrt aus einem Kreisverkehr.

`RoundaboutNoExit` An einer Ausfahrt im Kreisverkehr bleiben.

`MotorwayJunction` Eine Anschlussstelle einer Autobahn oder Schnellstraße.

`Restriction` Abstrakte Klasse zur Repräsentation unterschiedlicher Beschränkungen für Straßen oder Abbiegemöglichkeiten. Die einzelnen Unterklassen sind Multitons, um nicht unnötig Speicherplatz für mehrere gleiche Objekte zu verbrauchen.

Methoden

`allows` Bestimmt, ob es die Beschränkung erlaubt, unter dem angegebenen ↗Profil eine Straße oder Abbiegemöglichkeit zu nutzen.
Parameter:

`profile` Das verwendete ↗Profil. Typ: ↗`Profile`

Rückgabetyp: `boolean`

`VehicleTypeRestriction` Repräsentiert eine Beschränkung der Benutzung durch einen bestimmten Fahrzeugtyp.

Methoden

`getInstance` (statisch) Gibt eine Instanz dieser Klasse für den angegebenen Fahrzeugtyp zurück.
Parameter:

`type` Der Fahrzeugtyp. Typ: ↗`VehicleType`

Rückgabetyp: ↗`Restriction`

`WeightRestriction` Repräsentiert eine Beschränkung des Fahrzeuggewichts.

Methoden

`getInstance` (statisch) Gibt eine Instanz dieser Klasse für den angegebenen Wert zurück.
Parameter:

`weight` Das maximale Fahrzeuggewicht. Typ: `int`

Rückgabetyp: ↗`Restriction`

`WidthRestriction` Repräsentiert eine Beschränkung der Fahrzeugbreite.

Methoden

`getInstance` (statisch) Gibt eine Instanz dieser Klasse für den angegebenen Wert zurück.

Parameter:

`width` Die maximale Fahrzeugbreite. Typ: `int`

Rückgabetyp: ↗`Restriction`

`HeightRestriction` Repräsentiert eine Beschränkung der Fahrzeughöhe.

Methoden

`getInstance` (statisch) Gibt eine Instanz dieser Klasse für den angegebenen Wert zurück.

Parameter:

`height` Die maximale Fahrzeughöhe. Typ: `int`

Rückgabetyp: ↗`Restriction`

`MultipleRestrictions` Repräsentiert eine Kombination aus mehreren Beschränkungen.

Methoden

`getInstance` (statisch) Gibt eine Instanz dieser Klasse für den angegebenen Fahrzeugtyp zurück.

Parameter:

`restrictions` Eine Menge von Beschränkungen. Typ: `Set<Restriction>`

Rückgabetyp: ↗`Restriction`

6.9 Paket Exporter

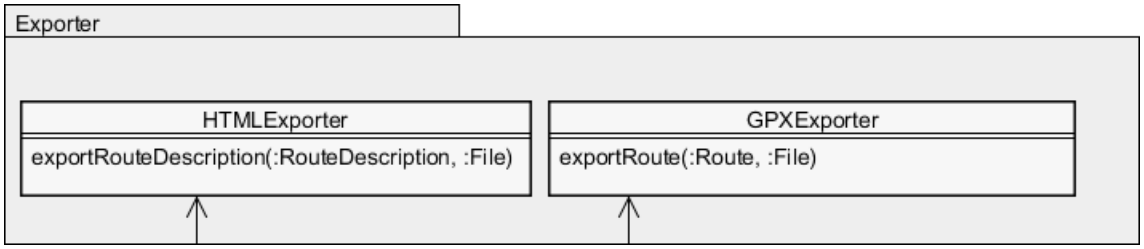

Abbildung 10: Das Paket Exporter

Dieses Paket enthält alle Klassen, die zum Exportieren der ↗Route oder ihrer Beschreibung benötigt werden.

`HTMLExporter` Stellt die Funktionalität zum Export der ↗Wegbeschreibung einer ↗Route im HTML-Format bereit.

Methoden

`exportRouteDescription` Exportiert die ↗`routeDesc` im HTML-Format in die angegebene Datei.
Parameter:

`routeDesc` Die zu exportierende ↗Wegbeschreibung. Typ: ↗`RouteDescription`

`file` Die HTML-Datei, die geschrieben werden soll. Typ: `File`

`GPXExporter` Stellt die Funktionalität zum Export einer ↗Route im ↗GPX-Format bereit.

Methoden

`exportRoute` Exportiert die Wegpunkte der ↗`route` im ↗GPX-Format in die angegebene Datei.
Parameter:

`route` Die zu exportierende ↗Route. Typ: ↗`Route`

`file` Die ↗GPX-Datei, die geschrieben werden soll. Typ: `File`

6.10 Paket History

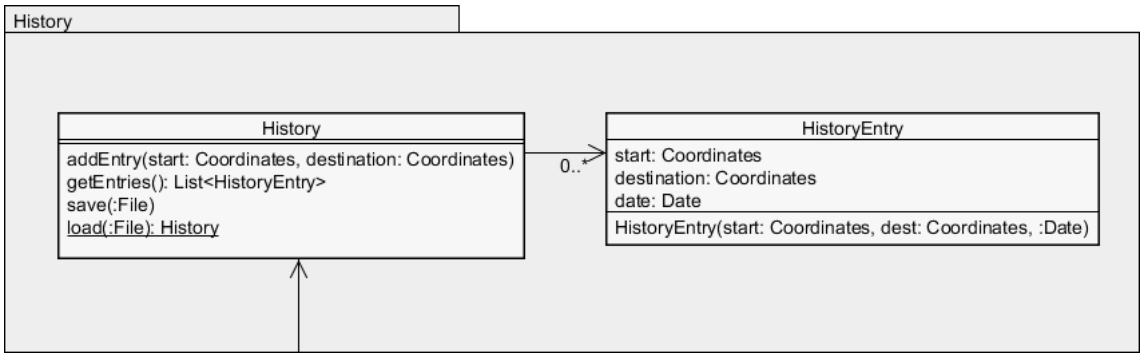

Abbildung 11: Das Paket History

Dieses Paket enthält alle Klassen, die für den ↗Verlauf benötigt werden.

`History` Kapselt den ↗Verlauf.

Methoden

`addEntry` Fügt einen Eintrag zum ↗Verlauf hinzu. Als ↗`HistoryEntry.date` des neuen Eintrags wird die aktuelle Zeit verwendet.
Parameter:

`start` Der Startpunkt. Typ: ↗`Coordinates`

`destination` Der Zielpunkt. Typ: ↗`Coordinates`

`getEntries` Gibt alle Einträge des ↗Verlaufs zurück. Rückgabetyp: `List<HistoryEntry>`

`save` Speichert den ↗Verlauf in die angegebene Datei.
Parameter:

`file` Die Datei, in die der ↗Verlauf gespeichert wird. Typ: `File`

`load` (statisch) Lädt einen ↗Verlauf aus der angegebenen Datei und gibt ihn zurück.
Parameter:

`file` Die Datei, aus der der ↗Verlauf geladen wird. Typ: `File`

Rückgabetyp: ↗`History`

`HistoryEntry` Ein Eintrag im ↗Verlauf.

Attribute

`start` Der Startpunkt der Anfrage. Typ: ↗`Coordinates`

`destination` Der Zielpunkt der Anfrage. Typ: ↗`Coordinates`

`date` Der Zeitpunkt der Anfrage. Typ: `Date`

Methoden

`HistoryEntry` Konstruktor: Erzeugt ein neues Objekt mit den angegebenen Attributen.
Parameter:

`start` Der Startpunkt. Typ: ↗`Coordinates`

`dest` Der Zielpunkt. Typ: ↗`Coordinates`

`date` Der Zeitpunkt. Typ: `Date`

6.11 Paket Util

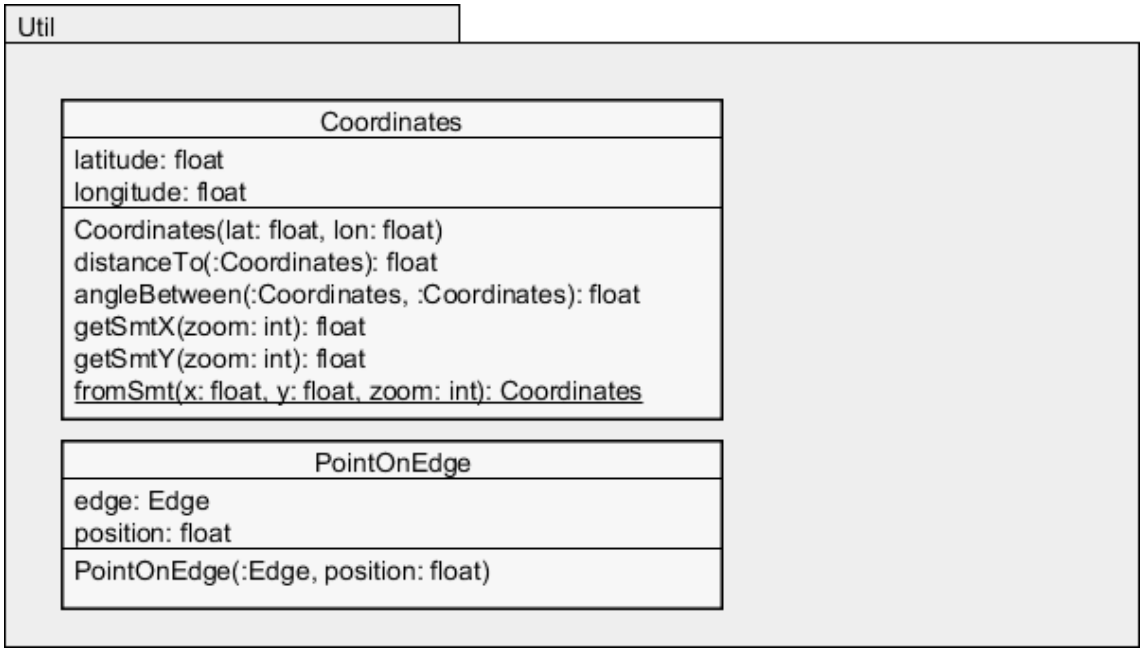

Abbildung 12: Das Paket Util

Dieses Paket enthält verschiedene nützliche Klassen, die sonst nirgendwo hingehören.

`Coordinates` Kapselt ein Paar geographischer Koordinaten.

Attribute

`latitude` Der Breitengrad des Koordinatenpaars. Typ: `float`

`longitude` Der Längengrad des Koordinatenpaars. Typ: `float`

Methoden

`Coordinates` Konstruktor: Erstellt ein neues Objekt aus den gegebenen Koordinaten.
Parameter:

`lat` Der Breitengrad. Typ: `float`

`lon` Der Längengrad. Typ: `float`

`distanceTo` Berechnet die Entfernung (Luftlinie, in Metern) zwischen den zwei Koordinaten.
Parameter:

`other` Die anderen Koordinaten. Typ: ↗`Coordinates`

Rückgabetyp: `float`

`angleBetween` Berechnet den zwischen einer Linie von diesem zum ersten und einer Linie von diesem zum zweiten Punkt eingeschlossenen Winkel.
Parameter:

`coords1` Die Koordinaten des ersten Punkts. Typ: ↗`Coordinates`

`coords2` Die Koordinaten des zweiten Punkts. Typ: ↗`Coordinates`

Rückgabetyp: `float`

`getSmtX` Berechnet die ↗SMT-X-Komponente zu diesen Koordinaten.
Parameter:

`zoom` Die Zoomstufe. Typ: `int`

Rückgabetyp: `float`

`getSmtY` Berechnet die ↗SMT-Y-Komponente zu diesen Koordinaten.
Parameter:

`zoom` Die Zoomstufe. Typ: `int`

Rückgabetyp: `float`

`fromSmt` (statisch) Rechnet ↗SMT-Koordinaten in Koordinaten um.
Parameter:

`x` Die ↗SMT-X-Komponente. Typ: `float`

`y` Die ↗SMT-Y-Komponente. Typ: `float`

`zoom` Die Zoomstufe. Typ: `int`

Rückgabetyp: ↗`Coordinates`

`PointOnEdge` Beschreibt einen Punkt auf der Kante.

Attribute

`edge` Die Kante, auf der sich der Punkt befindet. Typ: `Edge`

`position` Eine Zahl zwischen 0 und 1, die den Anteil der Strecke vom Punkt zum Anfangsknoten an der Gesamtlänge der Kante angibt. Typ: `float`

Methoden

`PointOnEdge` Konstruktor: Erstellt ein neues Objekt aus den gegebenen Attributen.
Parameter:

`edge` Die Kante. Typ: `Edge`

`position` Der Wert für das Attribut ↗`position`. Typ: `float`

7 Sequenzdiagramme

7.1 Programmstart

↗Abbildung 13 zeigt den Programmstart. Zunächst wird der ↗`MainController` erstellt (Konstruktor ↗`MainController.MainController`); dieser erstellt wiederum die restlichen Komponenten des Programms.

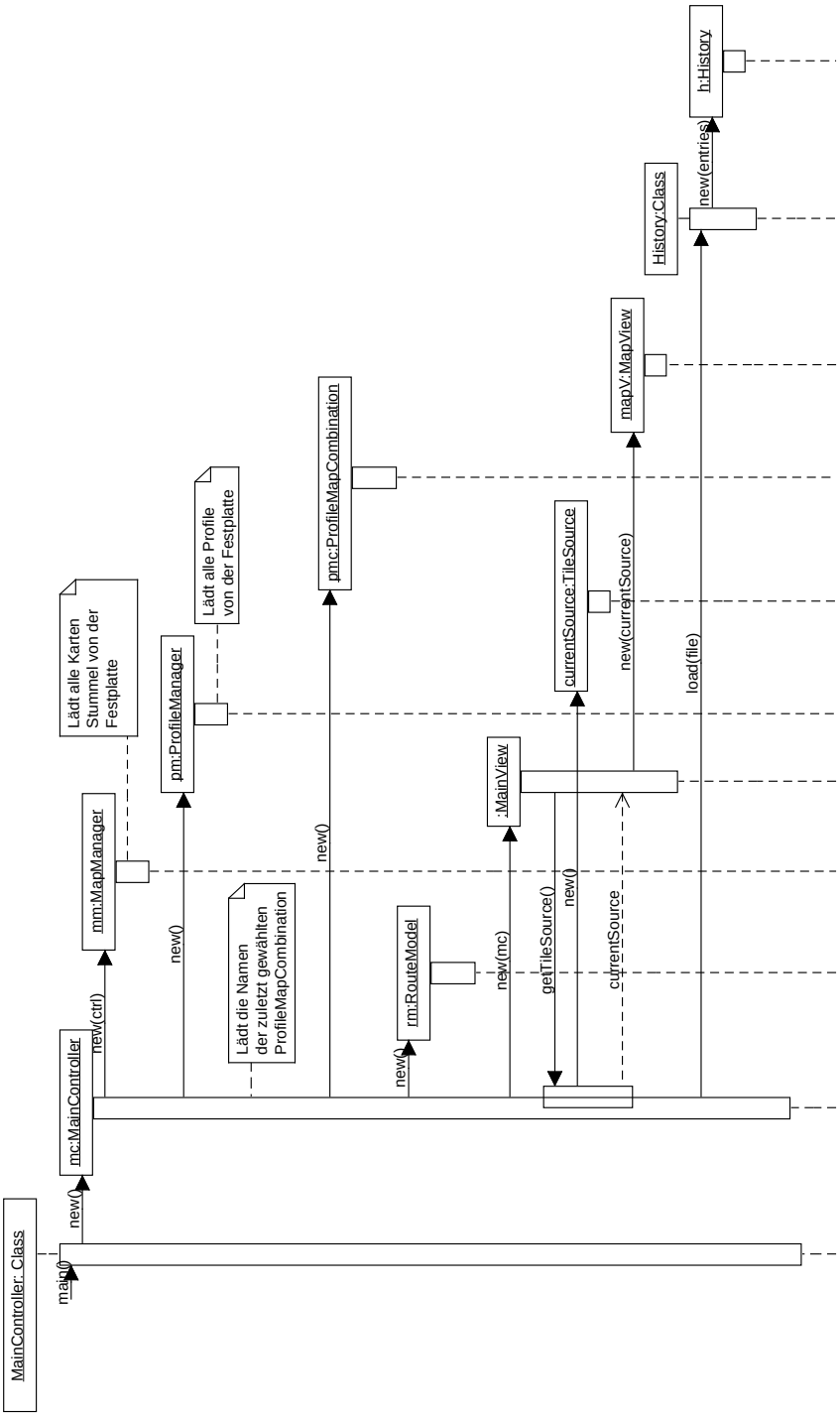

Abbildung 13: Start des Programms

Beteiligte Klassen: `RouteKit`, ↗`MainController`, ↗`MapManager`, ↗`ProfileManager`, ↗`ProfileMapCombination`, ↗`RouteModel`, ↗`MainView`, ↗`TileCache`, ↗`MapView`

7.2 Rendern

↗Abbildung 14 zeigt, wie die ↗`MapView` eine Kartenkachel vom ↗`TileCache` anfordert. Hier ist die Kachel noch nicht zwischengespeichert und wird vom Cache also auf Anfrage generiert.

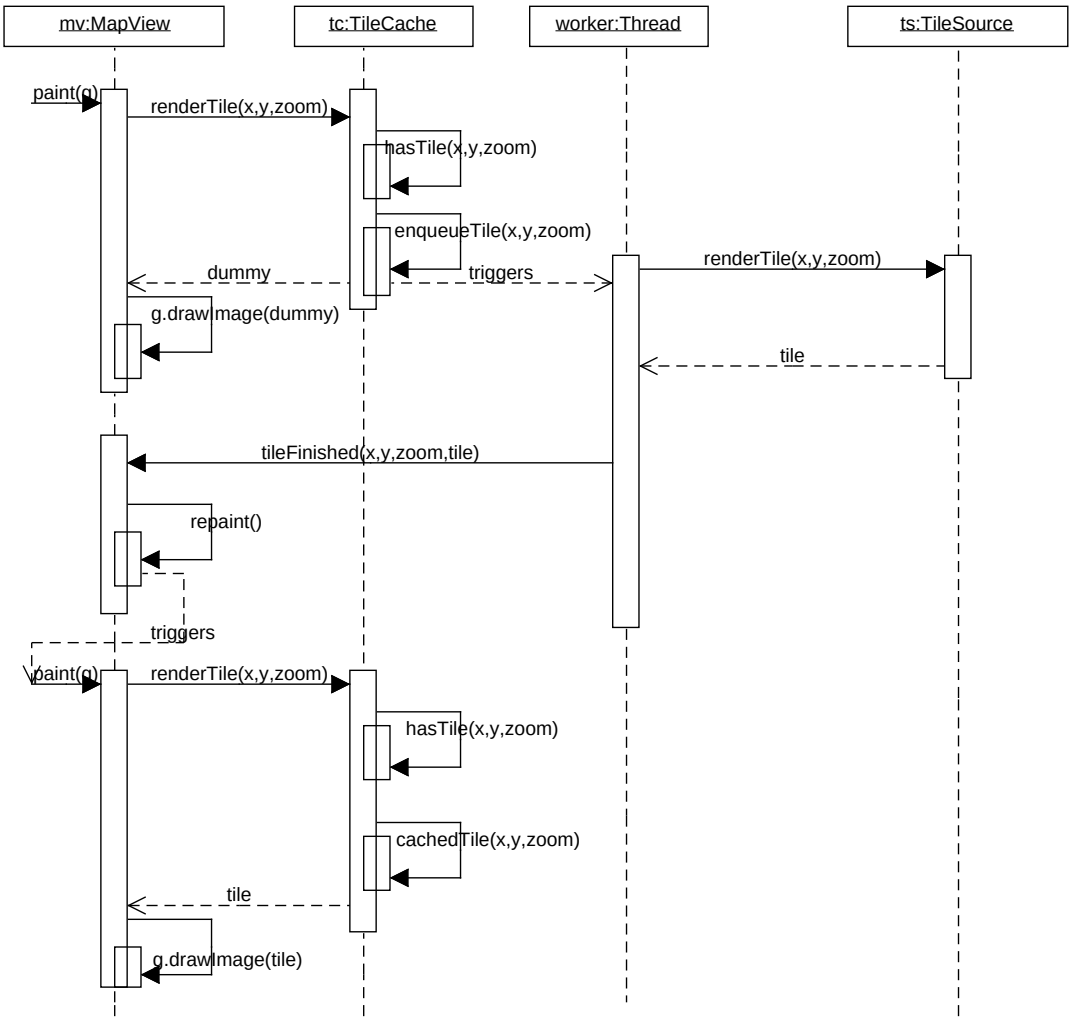

Abbildung 14: Rendern der Karte

Beteiligte Klassen: ↗`MapView`, ↗`TileCache`, ↗`TileSource`

7.3 Routenberechnung

↗Abbildung 15 zeigt den Ablauf der Routenberechnung, wenn noch kein Start- und Zielpunkt gesetzt sind. (Wäre ein Zielpunkt bereits gesetzt, so würde bereits das Setzen des Startpunkts die Routenberechnung auslösen.)

Abbildung 15: Berechnen der Route

Beteiligte Klassen: ↗`MainView`, ↗`MapView`, ↗`MainController`, ↗`RouteModel`,

↗`RouteCalculator`, ↗`RouteDescription`, ↗`Route`, ↗`RouteDescription`

7.4 Profilverwaltung

↗Abbildung 16 zeigt den Ablauf einer Profilverwaltung, bei der der Benutzer ein neues Profil „Smart“ anlegt, es 500 kg leichter macht und dann ein Profil „Mercedes“ löscht.

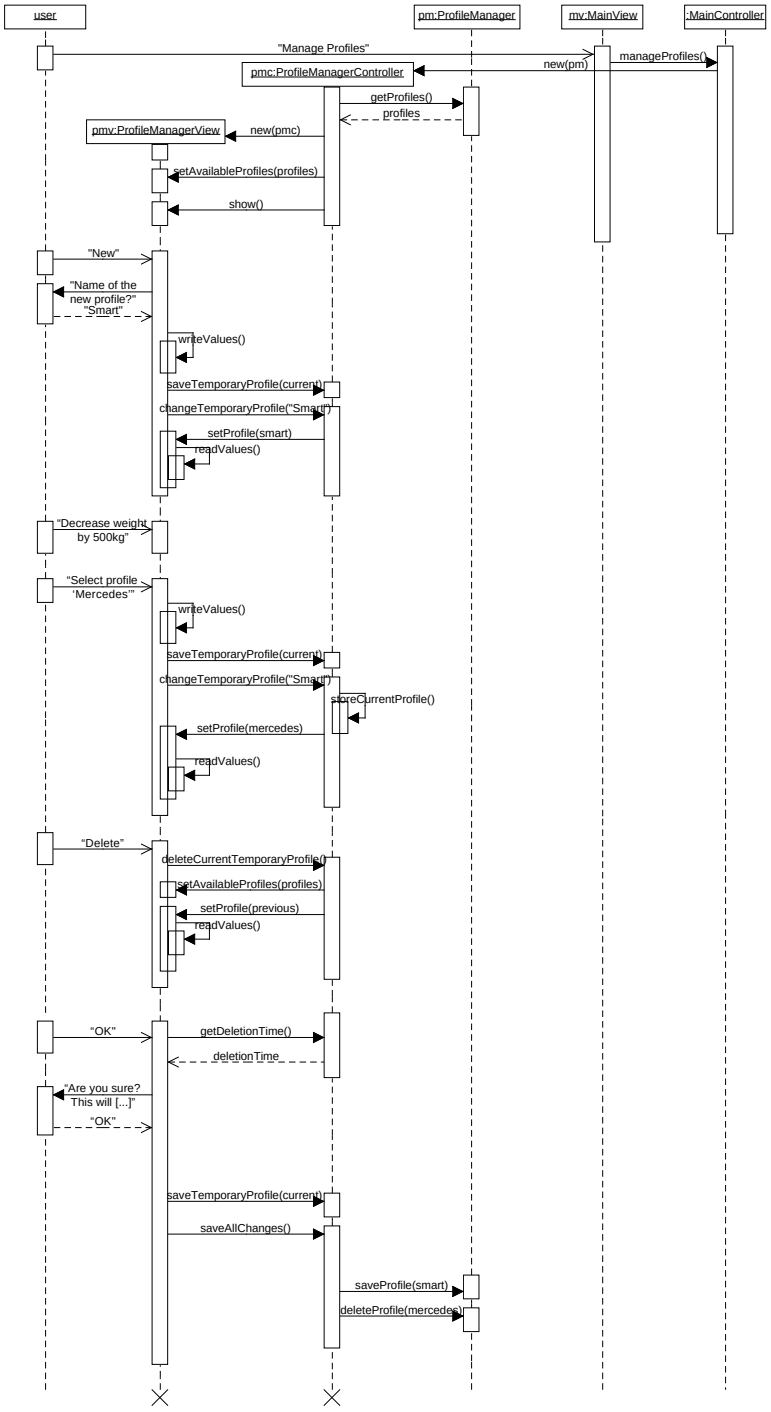

Abbildung 16: Hinzufügen und Entfernen von Profilen

Beteiligte Klassen: ↗`ProfileManagerView`, ↗`ProfileManagerController`, ↗`ProfileManager`, ↗`MainView`

7.5 Vorberechnung

↗Abbildung 17 zeigt den Ablauf der Vorberechnung.

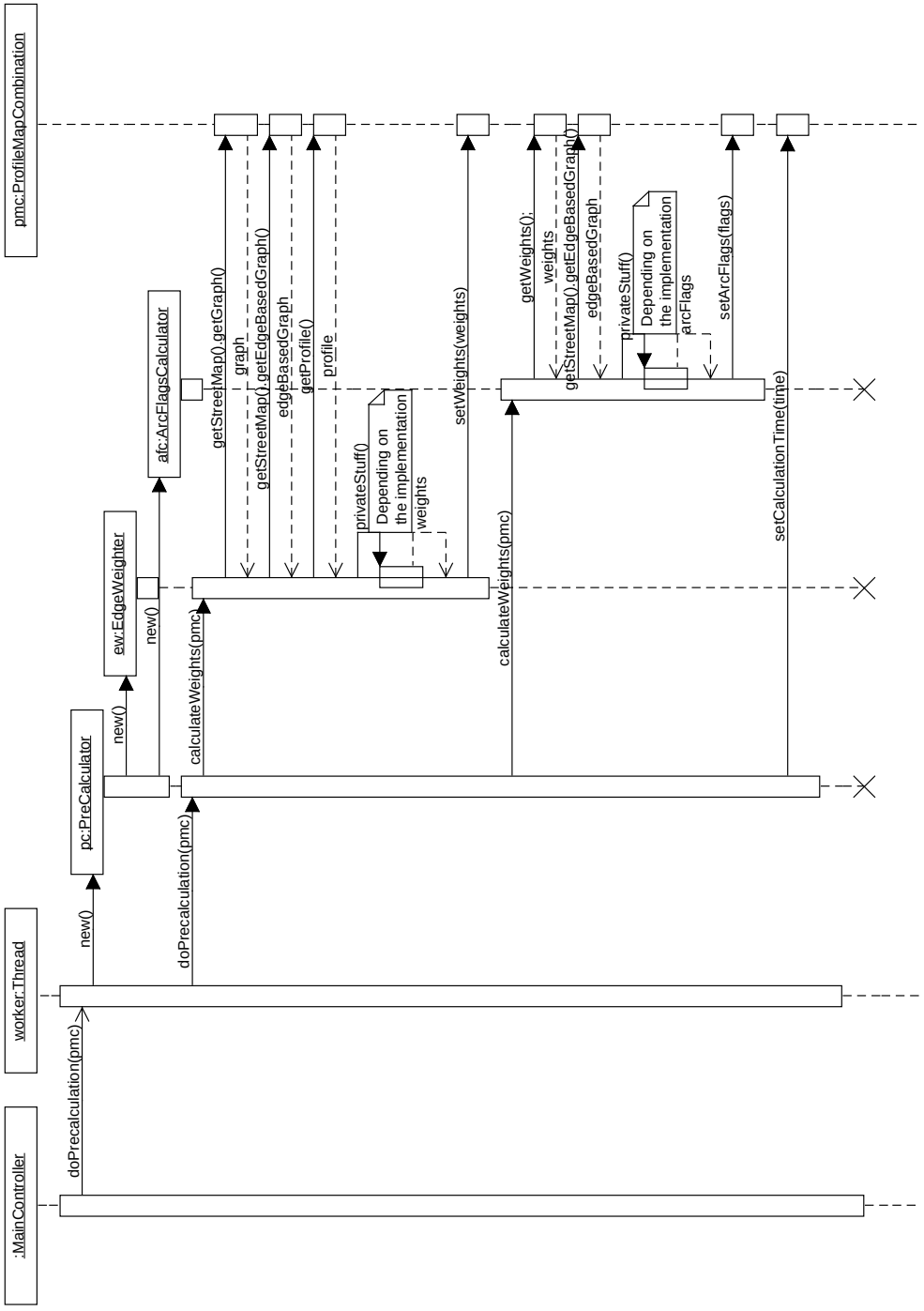

Abbildung 17: Vorberechnung für eine Karte und ein Profil

Beteiligte Klassen: ↗`MainController`, ↗`PreCalculator`, ↗`EdgeWeighter`, ↗`ArcFlagsCalculator`, ↗`ProfileMapCombination`

8 Glossar

Arc-Flags

eine Technik, um eine Routenberechnung zu beschleunigen. Auf Seite 8, 11, 12, 16, 37.

Beobachter

Ein ↗Entwurfsmuster, das bei Änderung des Zustands eines Objektes alle abhängenden Objekte banachrichtigt. Auf Seite 2, 3.

deckend

Die innere Implementierung wird vor anderen Klassen verborgen; Zugriff ist nur über das definierte Interface der Klasse/Methode möglich. (engl. "opaque"). Auf Seite 7, 8.

Dekorierer

Ein ↗Entwurfsmuster, das dynamisch neue Funktionalität zum Objekt hinzufügt. Auf Seite 3.

Dijkstra's Algorithmus

ein Algorithmus, um den optimalen Pfad in einem gerichteten Graphen zu finden. Auf Seite 12.

Einzelstück

Ein ↗Entwurfsmuster, das zusichert, dass eine Klasse genau ein Exemplar besitzt und stellt einen globalen Zugriffspunkt darauf bereit. Auf Seite 3.

Entwurfsmuster

Eine Schablone, die eine Lösung für ein oder mehrere Entwurfsprobleme darstellt. Ihre Verwendung erleichtert Änderungen im Entwurf. Auf Seite 2, 3, 36, 37.

GPX

GPS Exchange Format. Auf Seite 5, 28.

GUI

Graphical User Interface. Auf Seite 7, 13, 15.

Karte

enthält die Daten aus ↗OSM sowie einen Namen. Auf Seite 2, 5–9, 11–13, 16, 18, 19, 21, 22, 24, 37.

LKW

Lastkraftwagen. Auf Seite 21.

Mercator-Projektion

die Zylinderprojektion der Weltkugel. Auf Seite 18, 37.

MVC

Model View Controller. Auf Seite 2, 4, 16, 18.

OSM

OpenStreetMap. Auf Seite 9, 14, 36, 37.

OSM-Kachel

die öffentlichen gerenderten ↗OSM-Kacheln z. B. auf `http://a.tile.openstreetmap.org/`. Auf Seite 6, 15.

PKW

Personenkraftwagen. Auf Seite 21.

Profil

enthält für die Routenplanung relevante Informationen, z. B. die Höhe und das Gewicht des Fahrzeugs. Auf Seite 2, 4–7, 11, 12, 16, 18–22, 24, 26, 37.

rendern

Berechnung einer Bildkachel aus den zu rendernden Daten. Auf Seite 6, 14, 19, 21.

Route

ein Weg von einem Start- zu einem Zielpunkt. Auf Seite 2, 3, 5, 12, 17, 27, 28, 37, 38.

Slippy Map Tile

Koordinaten, durch die Kartenkacheln einer graphischen Darstellung (mit ↗Mercator-Projektion) adressiert werden können. Drei Bestandteile: x, y, zoom. Mit nichtganzzahligen Werten für x und y können nicht nur Kacheln, sondern auch Geokoordinaten adressiert werden. Auf Seite 14.

SMT

Slippy Map Tile. Auf Seite 14, 30, *Glossareintrag:* Slippy Map Tile.

Strategie

Ein ↗Entwurfsmuster, das eine Familie von Algorithmen definiert, sie kapselt und austauschbar macht. Auf Seite 3.

Verlauf

speichert Routenanfragen für spätere Wiederverwendung. Es existiert nur ein Verlauf für die gesamte Anwendung. Auf Seite 5, 19, 28, 29.

Vorberechnung

verbindet die ↗Karte mit den auf einem ↗Profil basierenden Informationen und ermöglicht, unter anderem mittels ↗Arc-Flags, später eine Beschleunigung der Berechnung einer ↗Route; muss für jede neue ↗Karte/↗Profil-Kombination einmal ausgeführt werden. Auf Seite 2, 5–8, 10, 16, 22, 24, *siehe* Arc-Flags.

Wegbeschreibung

Beschreibung einer ↗Route durch eine Liste von Abbiegeanweisungen. Auf Seite 5, 11, 12, 27, 28.