Entwurf

Praxis der Softwareentwicklung 13
Entwicklung eines Fahrradroutenplaners
Team 16

Sven Esser, Manuel Fink, Thomas Keh,
Tilman Väth, Lukas Vojković, Fabian Winnen

WS 2011/2012

Inhaltsverzeichnis

1 Einleitung

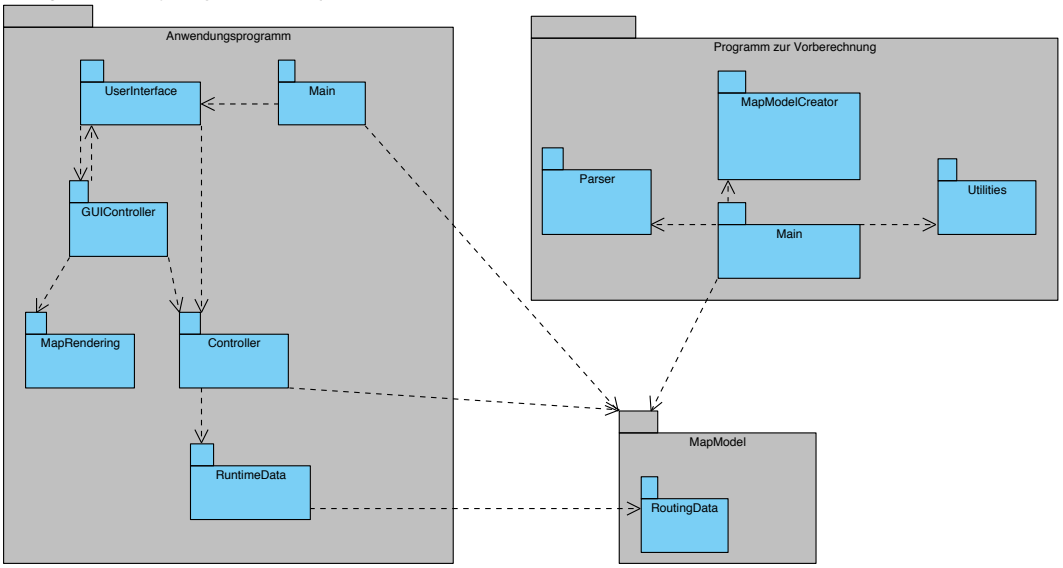
Bei diesem Projekt wird ein Fahrradroutenplaner für die Umgebung Karlsruhe entwickelt. Mit ihm soll es möglich sein, eine geeignete Route zwischen vom Benutzer ausgewählten Start- und Zielpunkten zu berechnen und mit zugehörigen Informationen anzeigen zu lassen.

Als Grundlage für das Straßennetz dienen die Kartendaten des OpenStreetMap-Projektes.
Neben dem Anwendungsprogramm wird zusätzlich ein, nur für die Entwickler zugängliches, Programm entwickelt, welches verschiedene Aufgaben und Berechnungen einmalig vor der Auslieferung des Produkts durchführt.

In diesem Dokument soll der Entwurf der Software dokumentiert und erläutert werden. Dies beinhaltet die Erläuterung der Architektur, die Beschreibung aller verwendeten Klassen, sowie den Ablauf von beispielhaften Programmsequenzen.

2 Architektur

Das Projekt BiKeIT ist in zwei Programme aufgeteilt. Ein Teil ist das Programm zur Vorberechnung, welches unter anderem die Daten des OpenStreetMap-Projekts verarbeitet und der zweite Teil ist das Anwendungsprogramm. Sie teilen sich allerdings eine gemeinsame Datenstruktur - das MapModel. Die folgende Grafik stellt die grobe Aufteilung des gesamten Projekts dar.



Beim Programm zur Vorberechnung gibt es zunächst ein Parser-Paket welches dafür zuständig ist, die für die Kartendarstellung und Routenberechnung relevanten Informationen aus den OSM-Daten auszulesen. Über den MainController können diese Informationen dann durch das MapModelCreator-Paket verarbeitet werden, welches eine geeignete Datenstruktur für alle anzuzeigenden Kartendaten erstellt. Dabei werden StreetData und TerrainData einmal für jedes Zoomlevel erzeugt. Das ermöglicht performantes Rendern auch bei kleinen Zoomstufen. Außerdem berechnet das MapModelCreator-Paket die Arc-Flags für jede Kante, um später im Anwenderprogramm eine schnellere Routenberechnung zu ermöglichen. Hierfür wird ein Rückwärtsdijkstra verwendet, welcher im Utilities-Paket zur Verfügung steht. Dieses besitzt auch einen Exporter, der nun die erstellte Kartendatenstruktur in ein geeignetes Dateiformat schreibt, welches später vom Anwendungsprogramm genutzt wird, um diese wieder aufzubauen.

Das Anwendungsprogramm besitzt im Main-Paket einen Initialisierer, welcher beim Programmstart die Kartendatenstruktur aus der zuvor erstellten Datei wiederherstellt. Zudem wird die Benutzeroberfläche instanziiert. Im Paket UserInterface finden sich sämtliche Klassen, welche anzeigbare Elemente repräsentieren, wie etwa die Benutzeroberfläche, ein Höhenverlaufdiagramm, der momentan anzuzeigende Kartenausschnitt und ein Fenster mit einer detaillierten Routenbeschreibung. Alle Aktionen, die der Benutzer über die Oberfläche tätigt, werden über das GUIController-Paket verwaltet und an den jeweils zuständigen Controller im Controllers-Paket weitergeleitet. Für das Erzeugen der Karte steht ein eigenes MapRendering-Paket zur Verfügung, welches vom MapController gesteuert wird.
Zur Darstellung der Karte wird diese in Kacheln aufgeteilt. Die Aufteilung in Kacheln erfolgt gemäß der Konventionen des OpenStreetMap-Projektes zur Namensgebung der Karten-Kacheln. Hierbei ist jede Kachel 256*256 Pixel groß und die gesamte Weltkarte besteht abhängig von der Zoomstufe aus $2^{Zoomstufe} * 2^{Zoomstufe}$ Kacheln.
Jede Kachel zeichnet sich somit durch ihre Zoomstufe und ihre horizontale und vertikale Position im Kachelgitter (im Folgenden als indexX und indexY bezeichnet) aus.

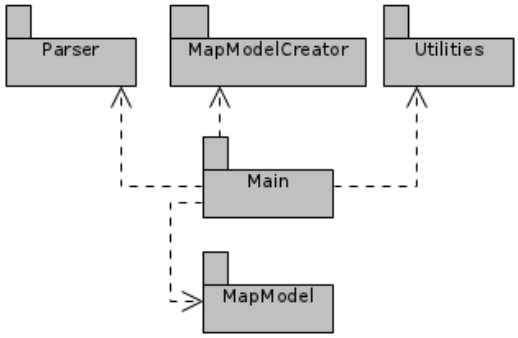
Das MapModel-Paket enthält die verarbeiteten und strukturierten Kartendaten. Die vom Benutzer zur Laufzeit geplante Route, sowie die dazu berechnete geeignete Route werden hingegen im RunTimeData-Paket verwaltet. Weiterhin gibt es ein Paket DataTypes, welches Klassen enthält, die einen bestimmten Datentyp repräsentieren und das Utility-Paket, welches unterstützende Funktionalität für diverse Klassen bereit stellt.

Im Folgenden werden alle Pakete sowie Klassen aufgeführt und genauer beschrieben.

3 Klassen des Programms zur Vorberechnung

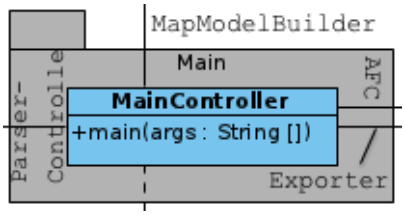
Das Programm zur Vorberechnung liest im Paket Parser vorhandene und benötigte Kartenrohdaten ein. Im Paket MapModelCalculator wird aus diesen Daten nun eine Datenstruktur aufgebaut, wie sie in Abschnitt 3.4.2 näher beschrieben ist. Außerdem werden Arc-Flags für die Kanten berechnet. Die fertige Struktur wird dann in ein geeignetes Dateiformat exportiert, sodass dieses beim Programmstart wieder geladen werden kann.

Das Programm enthält folgende Pakete:



Die einzelnen hierfür benötigten Klassen sollen im folgenden erläutert werden.

3.1 Paket Main



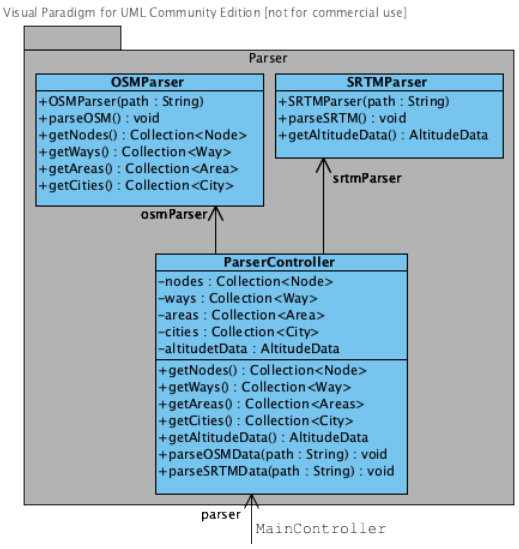
3.1.1 MainController

Der MainController ist der Kern des Programms zur Vorberechnung. Er bietet eine Shell an, mit der das Einlesen und Verarbeiten der OSM-Daten vom Benutzer gesteuert und angepasst werden kann.

Benötigt: ParserController, MapModelBuilder, ArcFlagCalculator, Exporter

3.2 Paket Parser

Das Paket Parser enthält alle nötigen Klassen, um OSM-Straßennetzdaten sowie SRTM-Höhendaten auszulesen und die darin enthaltenen Daten in einer für das MapModelCreator-Paket nutzbaren Struktur abzubilden.



3.2.1 ParserController

Der ParserController ermöglicht das Parsen von OSM- und SRTM-Daten, indem er zwei jeweils dafür vorgesehene Parser erzeugt und steuert. Das Parsen wird ausgelöst, indem man dem ParserController den Pfad zu einer Datei übergibt, welche OSM- oder SRTM-Daten enthält. Nach dessen Beendigung kann er das Produkt der Parser zurückgeben.

Benötigt: OSMParser, SRTMParser

3.2.2 OSMParser

Der OSMParser liest eine OSM-Datei ein und überprüft anhand der Tags, welche der Knoten und Wege für den Routenplaner relevant sind. Aus diesen beiden Informationsquellen erzeugt er Knoten-, Weg-, Flächen- und Stadtobjekte, welche er auf Anfrage in zusammengefasster Form zurückgibt.

Erzeugt: Node, Way, Area, City

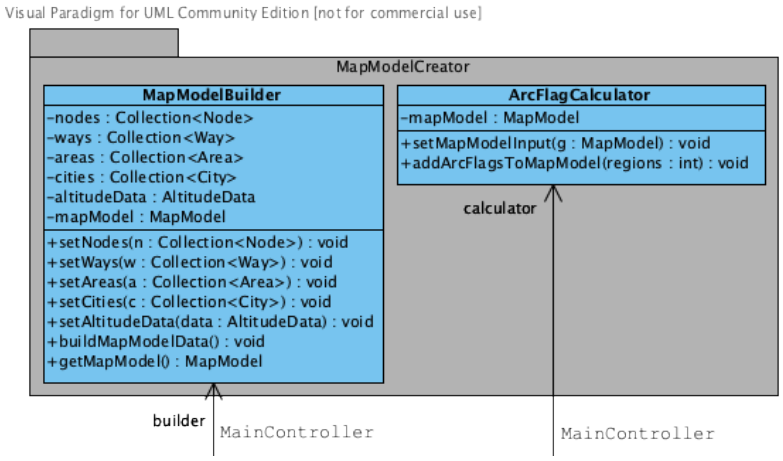
3.2.3 SRTMParser

Der SRTMParser liest eine SRTM-Datei ein und erzeugt aus den darin enthaltenen Höhendaten ein AltitudeData-Objekt (vgl. Abschnitt 4.0.15), welches er auf Anfrage zurückgibt.

Erzeugt: AltitudeData

3.3 Paket MapModelCreator

Das Paket MapModelCreator enthält Klassen, welche aus den vom Parser-Paket gelieferten Daten eine im MapModel-Paket (siehe Abschnitt 4) beschriebene Struktur erzeugen, die das Anwenderprogramm in dieser Form benötigt. Außerdem werden ArcFlag-Daten berechnet, mit denen die Struktur ergänzt wird, um später die Routenberechnung im Anwenderprogramm beschleunigen zu können.



3.3.1 MapModelBuilder

Der MapModelBuilder erzeugt aus den übergebenen Wegen, Flächen, Knoten, Städten und gegebenenfalls Höhendaten die für das Anwendungsprogramm benötigt. Zuerst erzeugt er aus den Wegen die Kanten und daraus dann die StreetData. Dazu erstellt er noch die TerrainData, CityData und NodeData. Daraus dann die ZoomDependentData und das MapModel.

Erzeugt: MapModel, ZoomLevelDependentData, NodeData, StreetData, TerrainData, CityData, Edge

3.3.2 ArcFlagCalculator

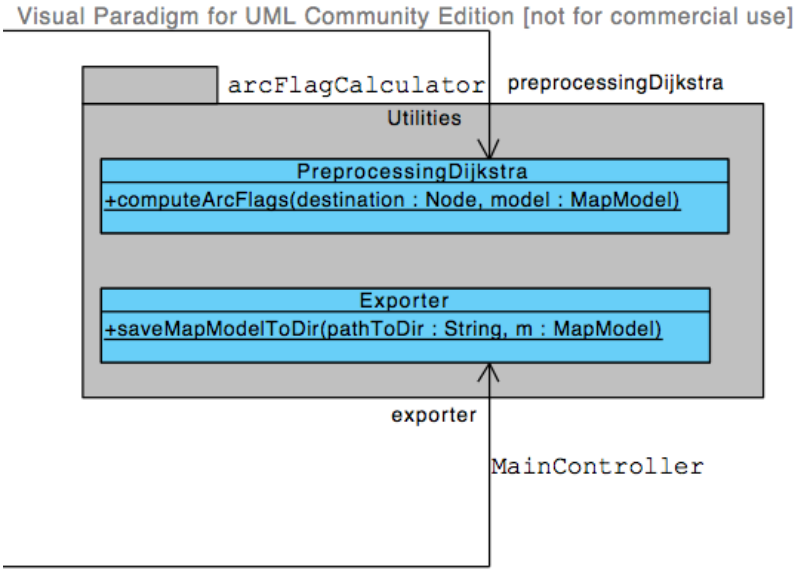
Der ArcFlagCalculator erweitert die Kanten eines MapModels um ArcFlag-Daten für eine maximale Anzahl von 128 Regionen. Zur Berechnung ist er auf die Klasse PreprocessingDijkstra angewiesen.
Im ersten Schritt werden die Knoten der MapModel-Struktur durch Setzen des „region“-Attributes in verschiedene Regionen aufgeteilt. Anschließend wird für die Knoten, die am Rand einer Region liegen, ein Rückwärts-Dijkstra ausgeführt, welcher einen Kürzesten-

Wege-Baum berechnet. Bei den Kanten, die in diesem Baum enthalten sind, wird die entsprechende Flag gesetzt.

Benötigt: PreprocessingDijkstra

3.4 Paket Utilities

Das Utilities-Paket enthält Klassen, die die Funktionalität des Vorberechnungsprogramms ergänzen.



3.4.1 Exporter

Speichert das vom MapModelCreator-Paket erstellte MapModel und alle zugehörigen Daten auf die Festplatte, sodass das Anwenderprogramm diese wieder einlesen und verwenden kann. Dem Exporter wird ein Pfad zu einem Ordner übergeben, in welchen die einzelnen Bestandteile der MapModel-Struktur geeignet in mehrere Dateien gespeichert werden.

Benötigt: MapModel

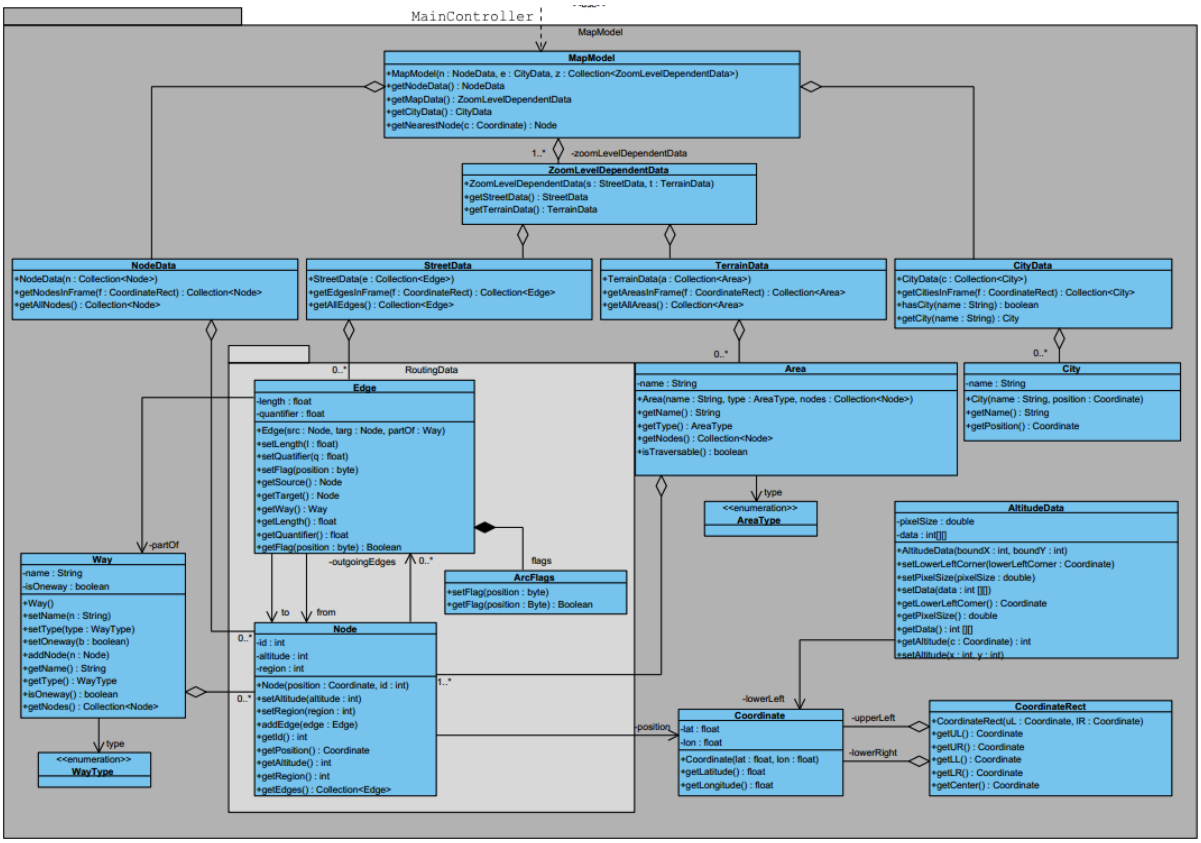
3.4.2 PreprocessingDijkstra

Führt einen Rückwärts-Dijkstra von einem gegebenen Startknoten aus durch. Die Kanten des dabei entstehenden Kürzeste-Wege-Baumes werden mit dem ArcFlag versehen, welches die Region des Startknotens kennzeichnet.

Benötigt: MapModel, Node

4 Klassen der Kartendaten

Im Folgenden sollen die Klassen zur Datenhaltung der Graphstruktur und verwandter Daten erläutert werden. Diese werden sowohl im Programm zur Vorberechnung, als auch im Anwendungsprogramm verwendet. Eine Übersicht gibt der folgende Ausschnitt des Klassendiagramms:



4.0.3 MapModel

Die Klasse MapModel repräsentiert alle Kartendaten. Sie stellt die NodeData, die CityData und insbesondere mehreren Instanzen der Klasse ZoomLevelDependentData bereit. Außerdem enthält sie eine Methode zur Findung des nächstgelegenen Knotens zu gegebener Koordinate.

Besteht aus: NodeData, ZoomlevelData (mehrere Instanzen), CityData

4.0.4 ZoomLevelDependentData

Die Klasse ZoomlevelData repräsentiert diejenigen Kartendaten, die für das jeweilige Zoomlevel relevant sind. Das Objekt mit dem niedrigsten Zoomlevel enthält somit alle vorhandenen Kartendaten - dieser Datensatz wird auch für die Routenberechnung verwendet.

Besteht aus: StreetData, TerrainData

4.0.5 Node

Die Klasse Node repräsentiert einen geographischen Punkt (Knoten), welcher seine eigenen Koordinaten, seine Höhe, alle von ihm ausgehenden Kanten, sowie die Region, in der er sich befindet, als Attribute beinhaltet. Die Klasse Node ist dabei, neben der Klasse Edge, der elementare Baustein aller im Programm aufgebauter Graphen.

Besteht aus: Coordinate
Benötigt: Edge
Benötigt von: NodeData, Edge, PlannedWayPoints, Way,

4.0.6 NodeData

Die Klasse NodeData repräsentiert alle Knoten und bietet eine Möglichkeit herauszufinden welche Knoten sich gerade im sichtbaren Bereich befinden.

Besteht aus: Node
Benötigt von: MapModel

4.0.7 StreetData

Die Klasse StreetData repräsentiert alle Kanten die für das jeweilige Zoomlevel angezeigt werden. Diese werden mithilfe der Klasse 2DDataStructure geeignet gespeichert, um einen schnellen Zugriff auf alle Kanten zu bekommen, die ein gewünschtes Koordinatenrechteck schneiden. Diese Zugriffsmöglichkeit wird von dem Renderer benötigt, um eine Kartenkachel zu zeichnen.

Besteht aus: Edge
Benötigt von: ZoomLevelDependentData

4.0.8 Edge

Die Klasse Edge repräsentiert eine Kante, die aus Start- sowie Zielknoten, einem Bewertungsfaktor, den ArcFlags und dem zugehörigen Weg besteht. Der Bewertungsfaktor hängt von der Beschaffenheit des Wegstücks und der Steigung ab. Außerdem bietet die Klasse Zugriff auf die Länge der Strecke, die zwischen Start- und Zielknoten liegt.

Die Klasse Edge ist im ganzen System das entscheidende Objekt, wenn es um Zeichenarbeiten geht.

Besteht aus: Node, ArcFlag, Way
Benötigt von: StreetData

4.0.9 ArcFlags

Die Klasse ArcFlags repräsentiert die ArcFlag-Information einer Kante, welche durch k Bits dargestellt wird. Dabei gilt k = Anzahl der Regionen in welches das gesamte Wegenetzwerk unterteilt ist. Das i-te Bit wird für eine Kante auf 1 gesetzt, falls ein kürzester Weg über diese Kante in die i-te Region führt. Die einzelnen Flags können gezielt gesetzt und abgefragt werden.

Enthalten in: Edge

4.0.10 Way

Die Klasse Way repräsentiert einen Weg, welcher aus mehreren, miteinander verbundenen Knoten besteht. Er besitzt einen Namen, wird in anhand der Fahrrad-Tauglichkeit unterschiedliche Typen (z.B. Radweg und Straße mit Radweg oder Kreis- und Gemeindestraßen ohne Radweg) aufgeteilt. Zudem wird festgehalten ob die Straße/der Weg für Radfahrer nur in eine Richtung befahrbar ist. Ein Way wird leer erzeugt und dann nach und nach mit den nötigen Informationen befüllt.

Besteht aus: Node, Waytype
Benötigt von: Edge

4.0.11 TerrainData

Diese Klasse TerrainData repräsentiert alle Flächen, welche für das jeweilige Zoomlevel angezeigt werden. Die Areas werden dazu in Bereiche aufgeteilt.

Besteht aus: Area
Benötigt von: ZoomLevelDependentData

4.0.12 Area

Die Klasse Area repräsentiert eine Fläche (z.B. Wälder, Seen, Plätze). Diese Fläche hat einen Namen, einen Typ (z.B. Wald, Stadt, See) und eine Sammlung der Knoten die ihren Rand markieren.

Besteht aus: Node, AreaType
Benötigt von: TerrainData

4.0.13 CityData

Die Klasse CityData repräsentiert alle Städte, welche für die Funktion, den Kartenausschnitt zu einer Stadt zu verschieben, bereitgestellt werden sollen. Sie bietet außerdem einen einfachen Zugriff auf alle Städte, deren gespeicherte Koordinate sich innerhalb eines gegebenen Koordinatenrechtecks befindet. Sucht der Anwender eine Stadt wird der eingegebene Name an diese Klasse weitergeleitet - ist eine solche vorhanden, wird sie als City-Objekt zurückgegeben.

Besteht aus: City (mehrere Instanzen)
Enthalten in: CityData
Benötigt von: MainGUIController

4.0.14 City

Die Klasse City repräsentiert eine Stadt und speichert hierfür zu dem Stadtnamen eine geeignete Koordinate.

Enthalten in: CityData

4.0.15 AltitudeData

Die Klasse AltitudeData speichert Höhendaten und liefert die Höhen für übergebene Koordinaten zurück. Dazu besitzt es die Seitenlänge eines quadratischen Pixels, die Koordinate an welcher der untere linke Pixel beginnt. Die klasse bietet die Möglichkeit gezielt ein Pixel mit einem Höhenwert zu versehen.

Besteht aus: Coordinate (unten links)
Benötigt von: MapModelBuilder

4.0.16 CoordinateRect

Diese Klasse definiert einen durch zwei Koordinaten definierten rechteckigen Bereich. Sie bietet mithilfe geeigneter Umrechnung Zugriff auf alle vier Eckkoordinaten und den Mittelpunkt des Rechtecks.

Besteht aus: Coordinate (oben links, unten rechts)

4.0.17 Coordinate

Die Klasse Coordinate repräsentiert eine geographische Koordinate aus Längengrad und Breitengrad.

Benötigt von: CoordinateRect, City, Node, MapController, AltitudeData

4.0.18 WayType

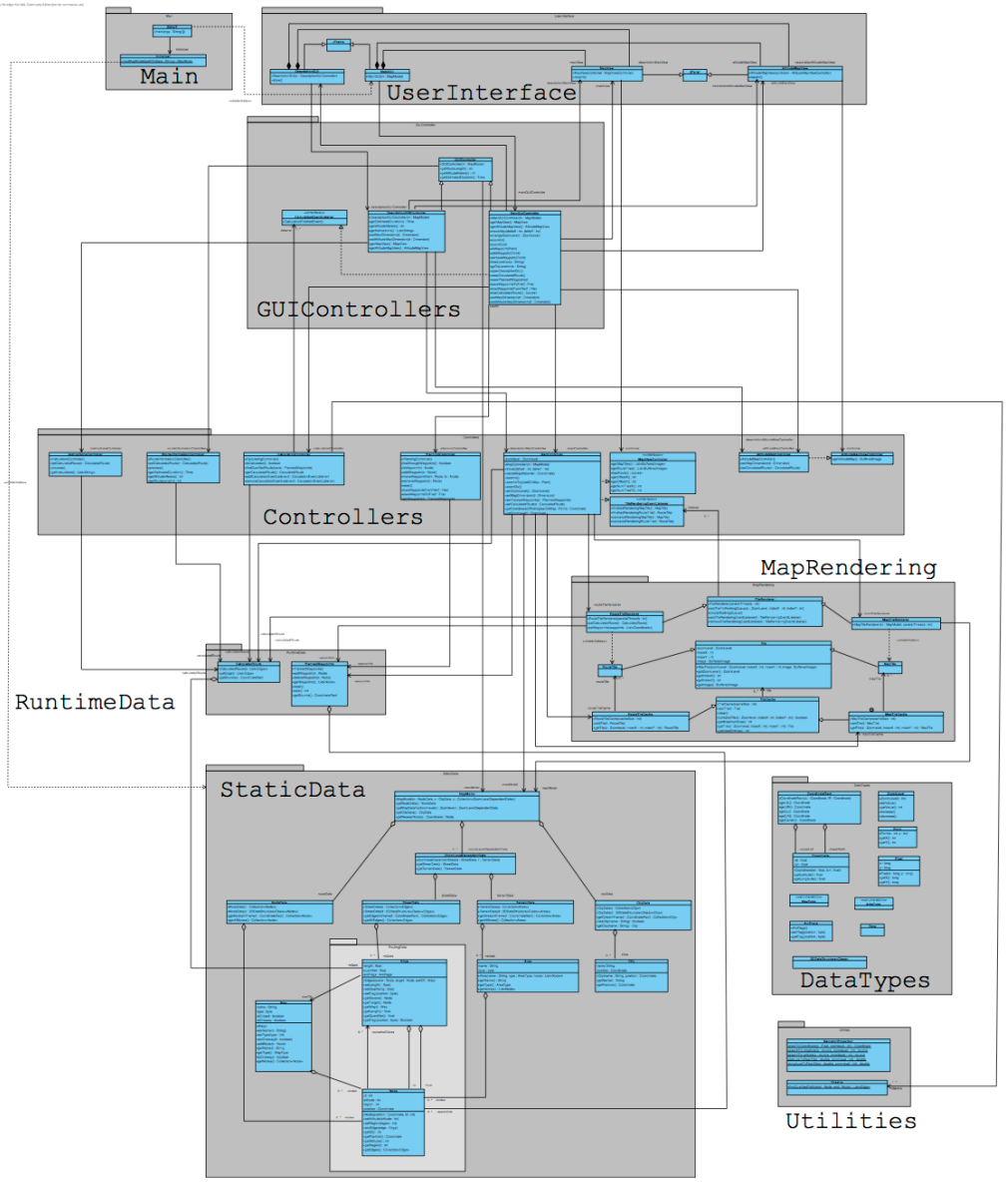
Die Enumeration WayType beschreibt den Typ eines Wegs - wie z.B. Landstraße. Es gibt dabei eine vordefinierte Menge an möglichen Wegtypen.

4.0.19 AreaType

Die Enumeration AreaType beschreibt den Typ einer Fläche - wie z.B. Wald. Es gibt dabei eine vordefinierte Menge an möglichen Flächentypen.

5 Klassen des Anwendungsprogramms

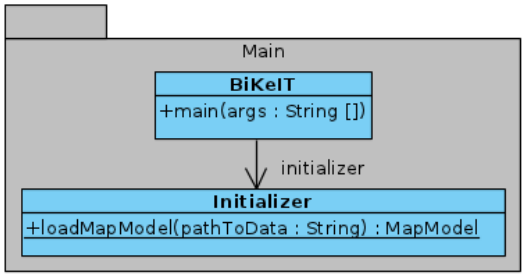
Das Anwendungsprogramm ist derjenige Teil des Projekts, der dem Endanwender ausgeliefert wird. Seine Implementierung ist in folgende Pakete aufgeteilt:



Die in den Paketen enthaltenen Klassen sollen im Folgenden erklärt werden.

5.1 Paket Main

Im Paket Main findet die Initialisierung des Systems statt.



5.1.1 BiKeIT

Hier befindet sich die Funktion main(), die den Start des Programms markiert.

Benötigt: Initializer, MainGUI

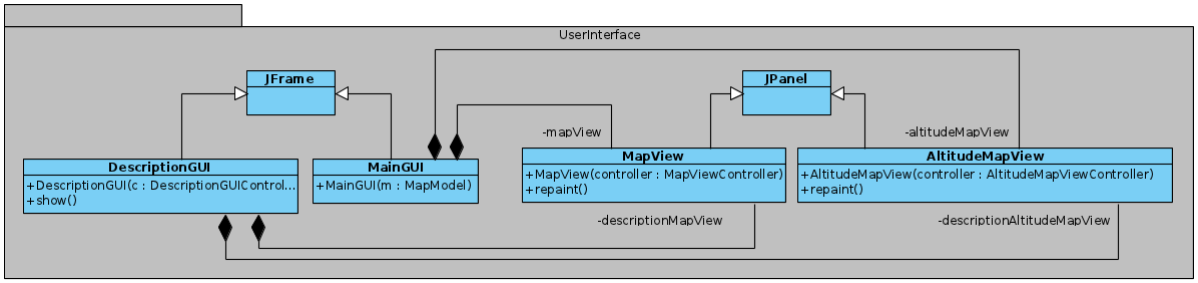
5.1.2 Initializer

Diese Klasse lädt die Kartendaten, die vom Vorberechnungsprogramm erstellt wurden. Ihr wird der Pfad zu dem Ordner übergeben, in dem die Dateien liegen, die die Kartendaten in einfacher Form enthalten. Diese werden in die später benötigte Datenstruktur umgewandelt und als MapModel-Objekt bereitgestellt.

Verwendet von: Main
Benötigt: Alle Klassen aus dem Paket MapModel

5.2 Paket UserInterface

Das Paket Userinterface enthält die gesamte Benutzerschnittstelle. Die Klassen in diesem Paket kümmern sich nur um die Anzeige auf Controllerebene vorgefertigter Informationen und bietet dem Nutzer die Möglichkeit Änderungen am internen Zustand, wie beispielsweise der Position der Karte, hervorzurufen.



5.2.1 MainGUI

Die Hauptoberfläche, die nach Programmstart sichtbar ist. Sie zeigt Karte, Höhenverlaufsdiagramm, Informationen und Knöpfe zur Benutzerinteraktion an. Sie reagiert auf Benutzerinteraktionen und leitet entsprechende Ereignisse an den MainGUIController weiter.

Erbt von: JFrame
Benötigt: MainGUIController
Besteht aus: MapView, AltitudeMapView

5.2.2 DescriptionGUI

Die Oberfläche, die angezeigt wird, wenn der Nutzer die Routenbeschreibung öffnet. Sie zeigt den Kartenausschnitt der berechneten Route und dazugehörige Informationen an. Sie reagiert auf Benutzerinteraktionen und leitet entsprechende Ereignisse an den DescriptionGUIController weiter.

Erbt von: JFrame
Benötigt: DescriptionGUIController
Besteht aus: MapView, AltitudeMapView

5.2.3 MapView

Oberflächenkomponente, die die Karte anzeigt. Sie besorgt sich die gerenderte Karte in Form von kleineren Teilkarten als BufferedImage. Diese, sowie weitere Informationen, wie ein Offset zur positionsgenauen Darstellung der Kartenteile, werden vom MapViewController bereitgestellt.

Teil von: MainGUI, DescriptionGUI (verschiedene Instanzen)
Benötigt: MapViewController
Erstellt von: MainGUIController, DescriptionGUIController

5.2.4 AltitudeMapView

Oberflächenkomponente, die bei berechneter Route ein Höhenverlaufsdiagramm derselben anzeigt. Sie besorgt sich das fertig gerenderte Diagramm vom AltitudeMapViewController als BufferedImage.

Teil von: MainGUI, DescriptionGUI (verschiedene Instanzen)
Benötigt: AltitudeMapViewController
Erstellt von: MainGUIController, DescriptionGUIController

5.2.5 JFrame

Java-eigene Swing-Klasse zur Darstellung eines Fensters.

Vererbt an: MainGUI, DescriptionGUI

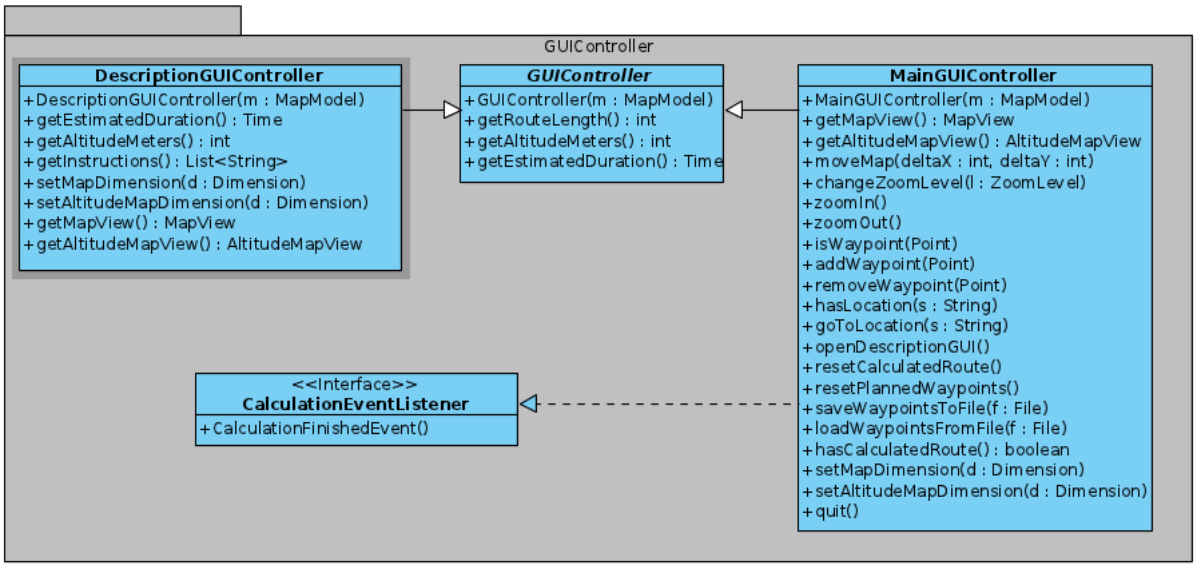
5.2.6 JPanel

Java-eigene Swing-Klasse zur Darstellung eines rechteckigen Elements im Fenster. In diesem Element kann bei entsprechender Überschreibung einiger Methoden gezeichnet werden.

Vererbt an: MainGUI, DescriptionGUI

5.3 Paket GUIController

Im Paket GUIController befinden sich der MainGUIController und der DescriptionGUIController, sowie verwandte Klassen und Schnittstellen. Sie bieten allesamt den Klassen aus dem UserInterface-Paket eine einfache Schnittstelle, mit allen Interaktionsmöglichkeiten, die sie benötigen. Die Controller in diesem Paket fungieren ausschließlich als Vermittler zwischen GUI und untergeordneten Controllern im Paket Controllers.



5.3.1 GUIController

Der abstrakte GUIController bietet Schnittstellen, die sowohl von der MainGUI, als auch von der DescriptionGUI benötigt werden.

Erzeugt: RouteInformationController
Vererbt an: MainGUIController, DescriptionGUIController

5.3.2 MainGUIController

Der MainGUIController bietet Schnittstellen, die nur von der MainGUI benötigt werden. Er erzeugt und verwaltet alle benötigten Controller, und leitet Funktionsaufrufe an diese weiter.

Die Klasse folgt dem Entwurfsmuster Fassade, da sie die Schnittstellen aus dem Paket Controllers bündelt.

Erzeugt: MapController, PlanningController, CalculatingController, AltitudeMapController, MapView, AltitudeMapView
Benötigt: MercadorProjection, MapModel
Erbt von: GUIController
Implementiert: CalculationEventListener

5.3.3 DescriptionGUIController

Der DescriptionGUIController bietet Schnittstellen, die nur von der DescriptionGUI benötigt werden. Er erzeugt und verwaltet alle benötigten Controller, und leitet Funktionsaufrufe an diese weiter.

Erzeugt: InstructionsController, MapController, AltitudeMapController, MapView, AltitudeMapView
Erbt von: GUIController

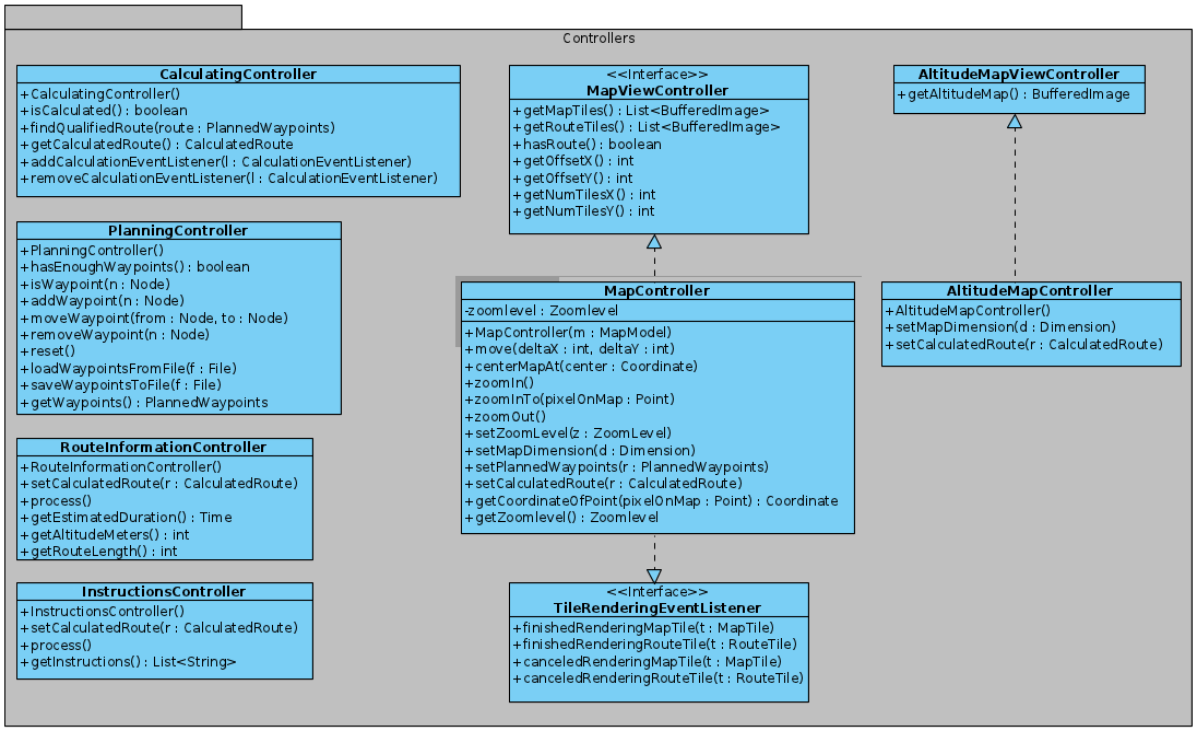
5.3.4 CalculationEventListener

Das Interface CalculationEventListener definiert Methoden, die von einem CalculationController benötigt werden um andere Klassen über den Fortschritt der Routenberechnung zu benachrichtigen.

Implementiert von: MainGUIController
Benötigt von: CalculatingController

5.4 Paket Controllers

Das Paket Controllers enthält nun die eigentlichen Funktionen des Programms. Einfachere Aufgaben werden hierbei selbst erledigt, bei komplexeren Aufgaben, wie dem Rendern von Karte und Route, wird Arbeit an Klassen in anderen Paketen delegiert.



5.4.1 InstructionsController

Der InstructionsController erstellt schrittweise Anweisungen zum Verfolgen der Route in Form einer Liste aus Zeichenketten. Er benötigt hierfür die berechnete Route als CalculatedRoute-Objekt. Auf das Ergebnis greift die DescriptionGUI über den DescriptionGUIController zu.

Erstellt von: DescriptionGUIController
Benötigt: CalculatedRoute

5.4.2 RouteInformationController

Der RouteInformationController berechnet Eigenschaften einer berechneten Route. Dazu zählen die Länge der Route, die geschätzte Fahrtdauer und die bewältigten Höhenmeter - das heißt die Summe der Höhendifferenzen eines jeden Anstiegs.

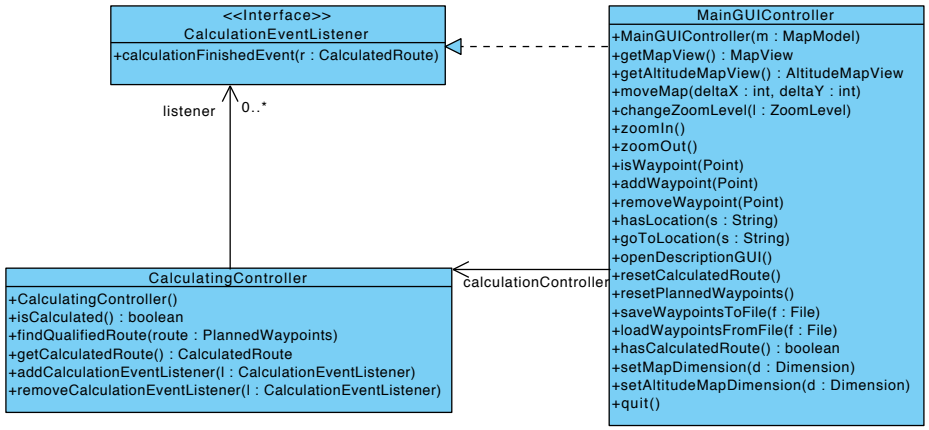
Erstellt von: GUIController

Benötigt: CalculatedRoute

5.4.3 CalculatingController

Der CalculatingController ist dafür zuständig das Berechnen einer geeigneten Route zu veranlassen und den MainGUIController über dessen Beendigung zu informieren. Er führt hierzu im Hintergrund einen Dijkstra-Algorithmus durch, der im Paket Utilities zu finden ist.

Die Klasse CalculatingController verfolgt das Entwurfsmuster Beobachter. Sie benachrichtigt alle angemeldeten CalculationEventListener. Der MainGUIController, der diese Schnittstelle implementiert, ist in diesem Fall ein konkreter Beobachter.



Erstellt von: MainGUIController
Benötigt: CalculatedRoute, CalculationEventListener, Dijkstra

5.4.4 PlanningController

Der PlanningController verwaltet die von der Oberfläche über den MainGUIController gesetzten Wegpunkte. Er bekommt Wegpunkte stets als Knoten überreicht, da der MainGUIController die Umsetzung des Punktes auf der Oberfläche zum nächstgelegenen Knoten schon durchgeführt hat. Außerdem bietet der Controller das Speichern und Laden der geplanten Wegpunkte in beziehungsweise aus einer Datei.

Erstellt von: MainGUIController
Benötigt: PlannedWaypoints, CalculationEventListener, Dijkstra

5.4.5 MapController

Der MapController verwaltet die Karte. Er hält ihren Zustand, bestehend aus Zoomlevel, Größe und Position des Kartenausschnittes, sowie der anzuzeigenden Route mit ihren Informationen.

Der MapController bietet Methoden um diesen Zustand zu ändern und sorgt dafür, dass immer die benötigten Kacheln für den aktuellen Zustand bereitstehen. Für diese Aufgabe verwaltet er jeweils einen Cache für Karten- und Routen-Kacheln. Wenn sich die geforderten Kacheln nicht im Cache befinden veranlasst er seinen MapTile- bzw. RouteTileRenderer die benötigte Kachel zu erstellen.
Über sein Interface TileRenderingEventListener wird er über den Fortschritt der von ihm veranlassten Render-Aufträge benachrichtigt.
Über das Interface MapViewController, stellt er alle Informationen zur Verfügung, die zum Darstellen einer Karte nötig sind.

Erstellt von: MainGUIController, DescriptionGUIController
Implementiert: MapViewController, TileRenderingEventListener
Benötigt: RouteTileRenderer, MapTileRenderer, RouteTileCache, MapTileCache
Reicht Referenzen weiter von: MapModel, PlannedWaypoints, CalculatedRoute

5.4.6 MapViewController

Die Schnittstelle MapViewController definiert Methoden, welche von einem MapView benötigt werden um eine Karte mit Route darzustellen.

Implementiert von: MapController
Benötigt von: MapView

5.4.7 TileRenderingEventListener

Die Schnittstelle TileRenderingEventListener definiert Methoden, welche von einem TileRenderer benötigt werden um andere Klassen über seinen Fortschritt bei Render-Vorgängen zu informieren.

Implementiert von: MapController
Benötigt von: TileRenderer

5.4.8 AltitudeMapController

Der AltitudeMapController verwaltet das Höhenverlaufsdiagramm. Er hält seinen Zustand, bestehend aus berechneter Route und der Größe des Diagramms. Der AltitudeMapController bietet Methoden um diesen Zustand zu ändern und sorgt dafür, dass immer ein passendes Diagramm zur Verfügung steht.
Er rendert das Diagramm selbst.
Über sein Interface AltitudeMapViewController stellt er alle Informationen zur Verfügung, die zur Darstellung eines Höhenverlaufdiagrammes nötig sind.

Erstellt von: MainGUIController, DescriptionGUIController
Implementiert: AltitudeMapViewController
Benötigt: CalculatedRoute

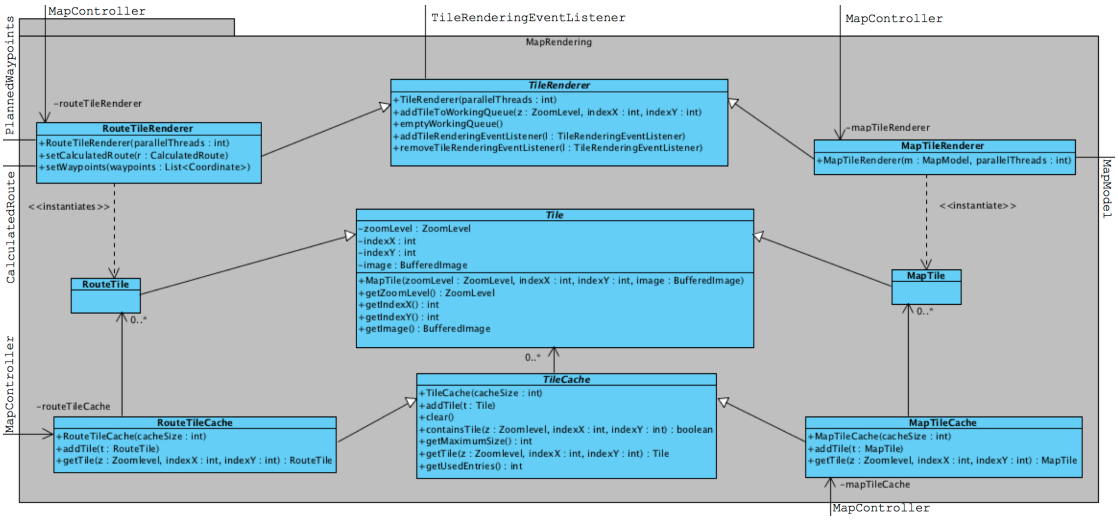
5.4.9 AltitudeMapViewController

Die Schnittstelle AltitudeMapViewController definiert Methoden, welche von einem AltitudeMapView benötigt werden um ein Höhenverlaufsdiagramm darzustellen.

Implementiert von: AltitudeMapController
Benötigt von: AltitudeMapView

5.5 Paket MapRendering

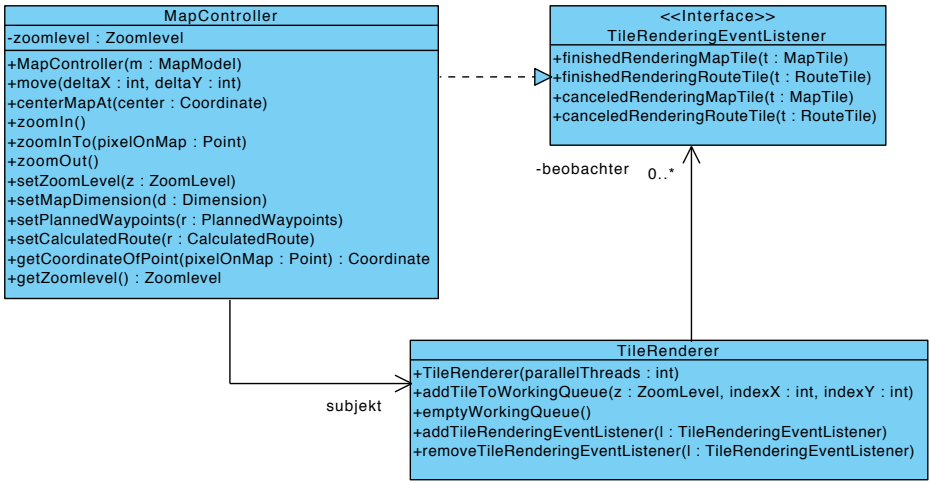
Im Paket MapRendering findet das Erzeugen der Kartenkacheln sowie der Routenkacheln statt. Dies sind Bilder, die von Klassen im Paket UserInterface benötigt werden, um die Karte mitsamt geplanten Wegpunkten und berechneter Route anzeigen zu können.



5.5.1 TileRenderer

Die Klasse TileRenderer nimmt Render-Aufträge für einzelne Kacheln entgegen und verteilt sie auf mehrere Threads, die das Rendern jeweils einer Kachel durchführen.

Außerdem werden bei Abschluss des Rendervorgangs bzw. bei dessen Entfernen aus der Warteschlange alle angemeldeten TileRenderingEventListener benachrichtigt. Die Klasse ist somit Teil des Entwurfsmusters Beobachter. Der MapController ist in diesem Fall ein konkreter Beobachter.



Benötigt: TileRenderingEventListener
Vererbt an: MapTileRenderer, RouteTileRenderer

5.5.2 MapTileRenderer

Der MapTileRenderer übernimmt das Rendern der Kartenkacheln. Er benötigt hierfür den Kachelindex der gewünschten Kachel, die Zoomstufe, sowie Zugriff auf das MapModel. Konkret überschreibt er die Renderingmethode der Oberklasse TileRenderer und implementiert somit ausschließlich das Rendern einer einzelnen Kartenkachel.

Benötigt: MapModel
Erbt von: TileRenderer

5.5.3 RouteTileRenderer

Der RouteTileRenderer übernimmt das Rendern der Routenkacheln. Er benötigt zusätzlich zum Kachelindex und der Zoomstufe, einerseits die geplanten Wegpunkte, die dann als Fähnchen oder ähnliche Markierung gezeichnet werden. Andererseits benötigt er auch die berechnete Route, um dieselbe auf die Kachel einzuzeichnen. Konkret überschreibt er die Renderingmethode der Oberklasse TileRenderer und implementiert somit ausschließlich das Rendern einer einzelnen Routenkachel.

Benötigt: PlannedWaypoints, CalculatedRoute
Erbt von: TileRenderer

5.5.4 Tile

Die Klasse Tile repräsentiert eine Kachel. Diese Kachel besteht aus einem 256*256 Pixel großen Bild als BufferedImage, sowie Identifikationsinformationen, also Kachelindex in X- und Y-Richtung und Zoomstufe.

Benötigt von: TileCache, TileRenderingEventListener
Vererbt an: MapTile, RouteTile

5.5.5 MapTile

MapTile ist eine Kachel, die einen reinen Kartenausschnitt darstellt - ohne vom Benutzer veränderliche Informationen.

Benötigt von: MapTileCache, MapController
Erbt von: Tile

5.5.6 RouteTile

RouteTile ist eine Kachel, die als teilweise transparenter Overlay über der Karte fungiert und darauf - bei Vorhandensein - die geplanten Wegpunkte, sowie die berechnete Route darstellt.

Benötigt von: RouteTileCache, MapController
Erbt von: Tile

5.5.7 TileCache

Der TileCache ist eine Datenablage für fertig gerenderte Tiles. Ist ein neues Tile fertig, wird es vom MapController in den TileCache eingefügt. Zum Schutz vor Speicherüberlauf werden lange nicht benötigte Tiles wieder aus dem Cache entfernt.

Vererbt an: MapTileCache, RouteTileCache

5.5.8 MapTileCache

Dies ist der TileCache für Kartenkacheln.

Benötigt von: MapController
Erbt von: TileCache

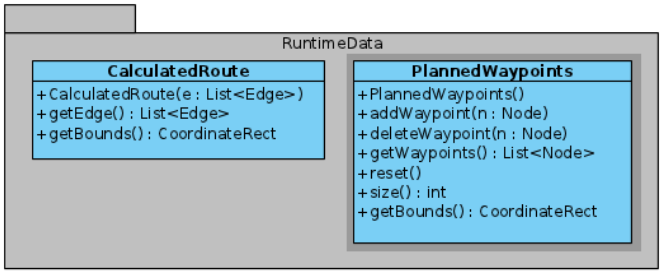
5.5.9 RouteTileCache

Dies ist der TileCache für Routenkacheln.

Benötigt von: MapController
Erbt von: TileCache

5.6 Paket RuntimeData

RunTimeData enthält Klassen, die zur Laufzeit entstandene Daten beschreiben.



5.6.1 CalculatedRoute

Diese Klasse enthält die berechnete Route in Form einer Liste von Kanten, sodass sie leicht gezeichnet werden kann.
Außerdem enthält diese Klasse eine Hilfsmethode getBounds(), womit das kleinste Rechteck ermittelt wird, in das alle Kanten hineinpassen. Dies kann verwendet werden, um die Route in voller Größe in der Karte anzuzeigen.

Besteht aus: Edge (mehrere Instanzen)
Benötigt von: RouteTileRenderer, AltitudeMapController, MapController, CalculatingController, RouteInformationController

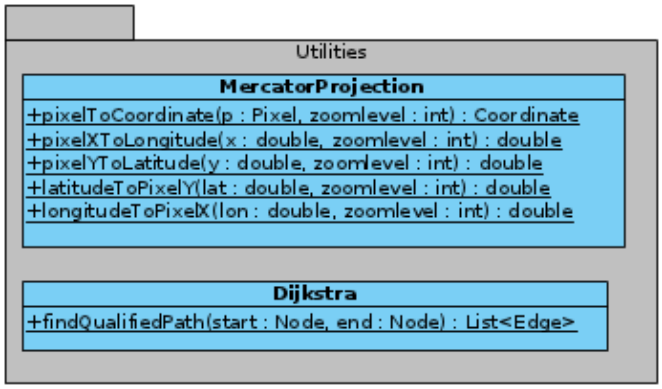
5.6.2 PlannedWaypoints

Diese Klasse enthält alle geplanten Wegpunkte in Form einer Liste von Knoten. Wegpunkte sind Knoten, die der Nutzer auf der Karte als Start-, Ziel-, oder Zwischenzielknoten auserwählt hat. Nur die Reihenfolge bestimmt hierbei die Funktion als Start- bzw. Zielknoten.
Außerdem enthält diese Klasse eine Hilfsmethode getBounds(), womit das kleinste Rechteck ermittelt wird, in das alle Wegpunkte hineinpassen. Dies kann verwendet werden, um die alle geplanten Wegpunkte gemeinsam in der Karte anzuzeigen.

Besteht aus: Node (mehrere Instanzen)
Benötigt von: PlanningController, RouteTileRenderer, MapController

5.7 Paket Utilities

Utilities enthält Klassen, die von mehreren anderen Klassen für allgemeine Aufgaben benötigt werden.



5.7.1 Dijkstra

Die Klasse Dijkstra implementiert den Algorithmus zur Findung einer geeigneten Route. Sie bekommt hierzu den Start- und Zielknoten. Da in jedem Knoten die ausgehenden Kanten referenziert sind, reicht dies als Ausgangsdatensatz aus. Der Algorithmus berücksichtigt auch den in den Kanten gespeicherten Qualifier um so speziell für Fahrradfahrer eine geeignete Route zu finden. Zurückgegeben wird eine sortierte Liste an Kanten.

Benötigt von: CalculatingController

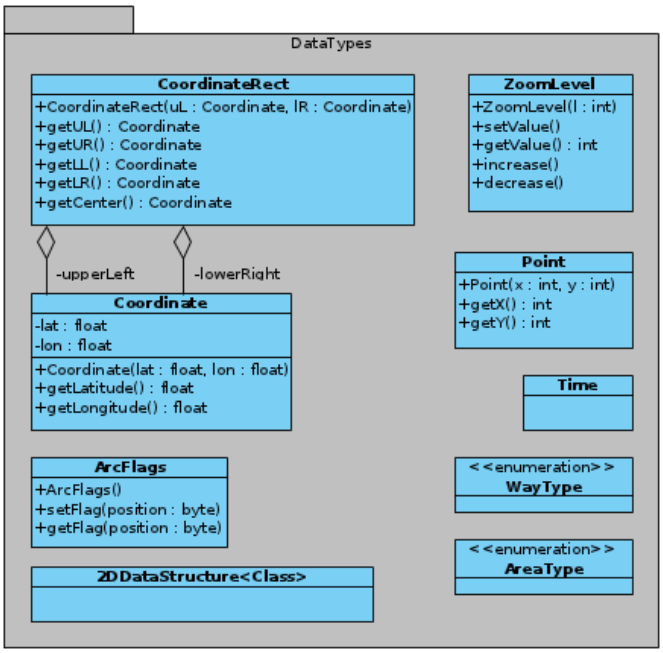
5.7.2 MercatorProjection

Die Klasse MercatorProjection implementiert Algorithmen und Formeln zur Umrechnung zwischen Pixeln und Koordinaten und benutzt dafür die Mercator Projektion. Alle Methoden dieser Klasse sind static und können somit von jeder Klasse aufgerufen werden, die solch eine Umrechnung durchführen muss. Die Mercator Projektion bildet hier Koordinaten auf den Pixelbereich $[0..2^{Zoomstufe} * 256]^2$ ab.

Benötigt von: MapController, PlanningController, MainGUIController, u.A.

5.8 Paket DataTypes

In diesem Paket befinden sich allgemeine Datentypen. Im folgenden sollen nur diejenigen beschrieben werden, welche nicht schon in Abschnitt 3.4.2 behandelt wurden.



5.8.1 ZoomLevel

Das Zoomlevel ist eine Ganzzahl in einem fest definierten Bereich. Diese Klasse vermeidet, dass fehlerhafte ZoomLevel-Zustände im System existieren können.

5.8.2 2DDataStructure ⟨Class⟩

2DDataStructure ist eine generische Klasse, die Daten, wie z.B. Kanten in einer für die Belange des Systems effizienten Weise ablegt. Dies kann beispielsweise ein 2D-Baum sein, der nach Koordinaten unterteilt ist.

5.8.3 Point

Point ist ein Punkt auf der Karte - gemessen in Pixeln.

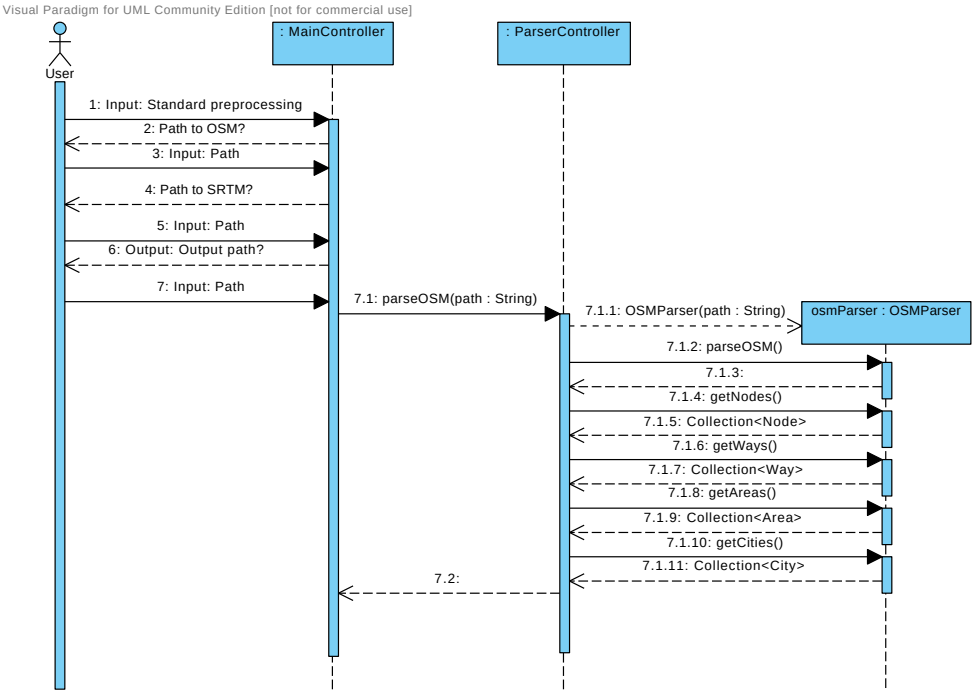
5.8.4 Time

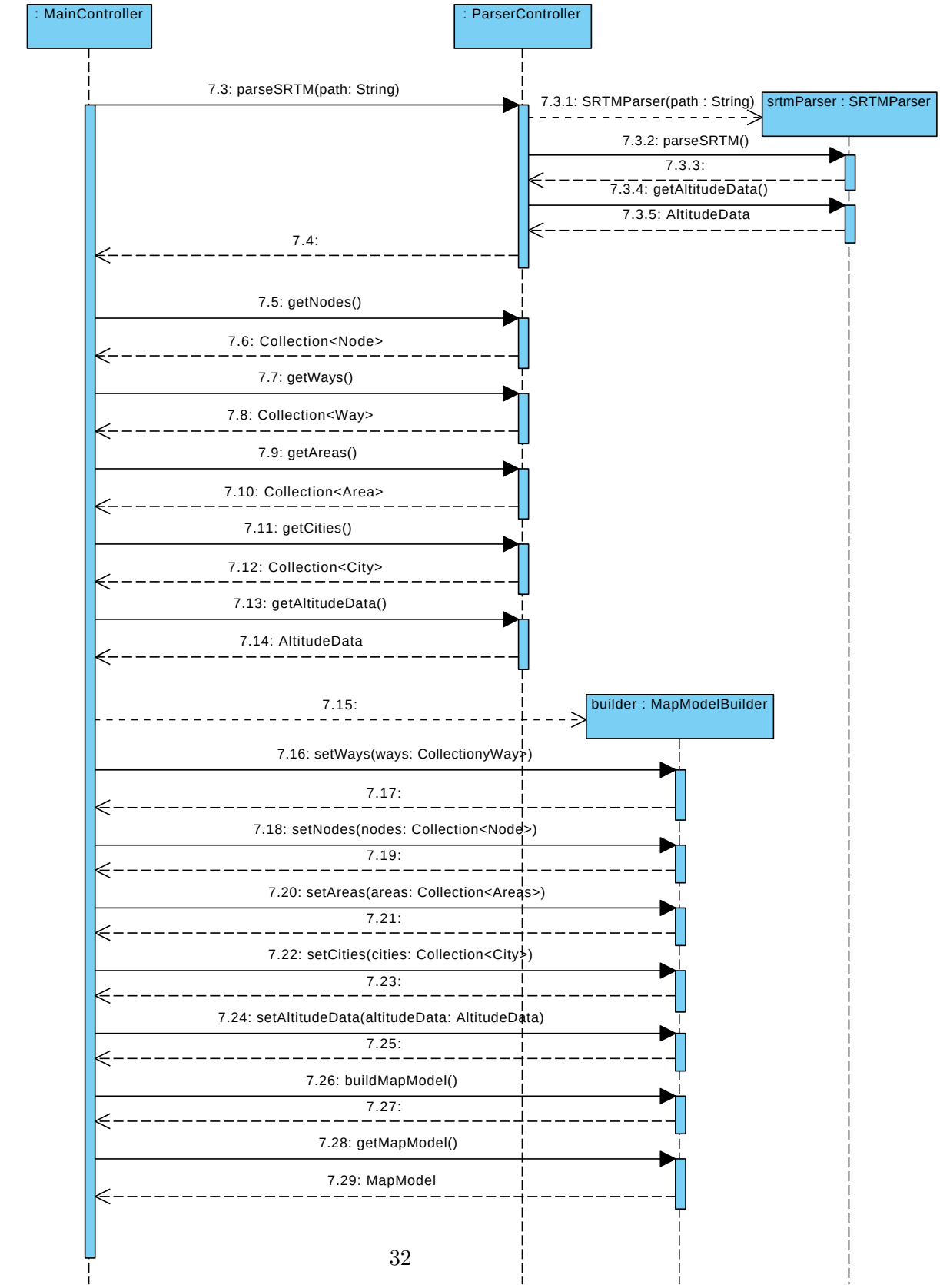
Der Java-eigene Typ Time repräsentiert eine Zeitdauer. Er wird vom RouteInformation-Controller für die geschätzte Fahrtdauer verwendet.

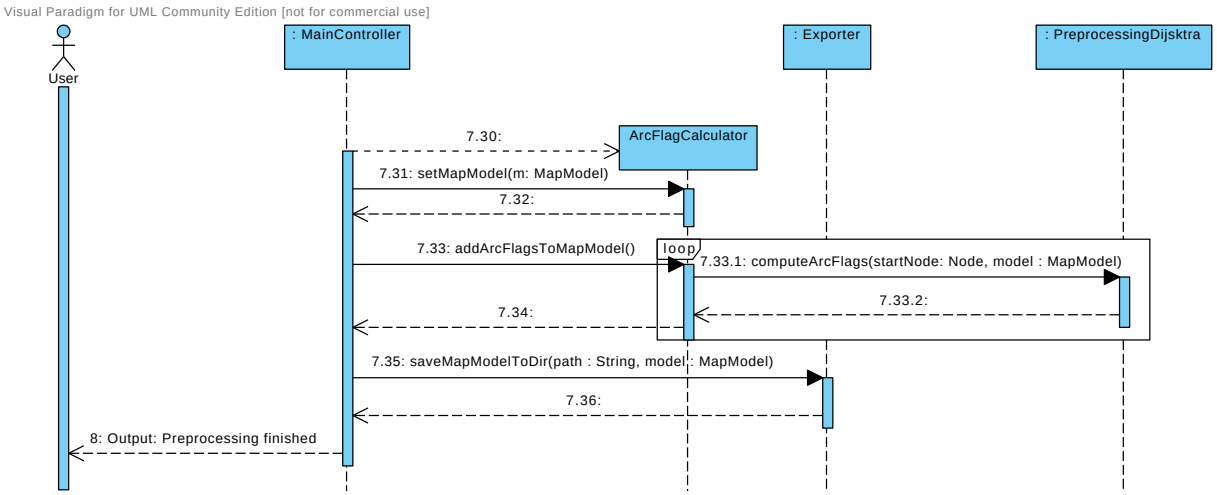
6 Sequenzdiagramme

6.1 Kartendaten Vorberechnen

Dieses Diagramm zeigt den Ablauf des Vorberechnungsprogramms. Zunächst werden die Pfade zu den Quelldateien und zum Zielverzeichnis eingelesen, wonach das Parsen derselben beginnt. Anschließend wird mithilfe des MapModelBuilder eine geeignete Datenstruktur aufgebaut, die dann im Zielverzeichnis gespeichert wird.

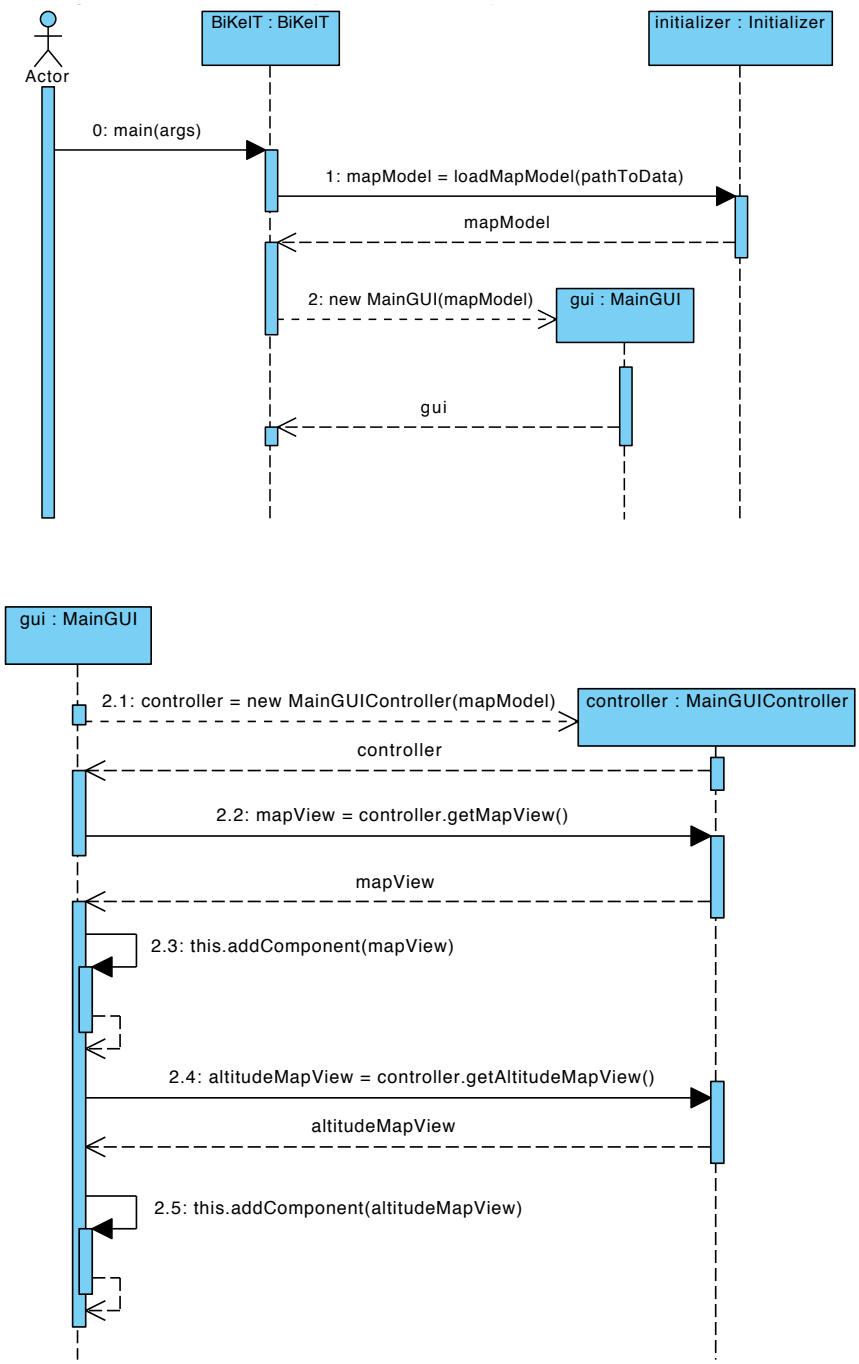


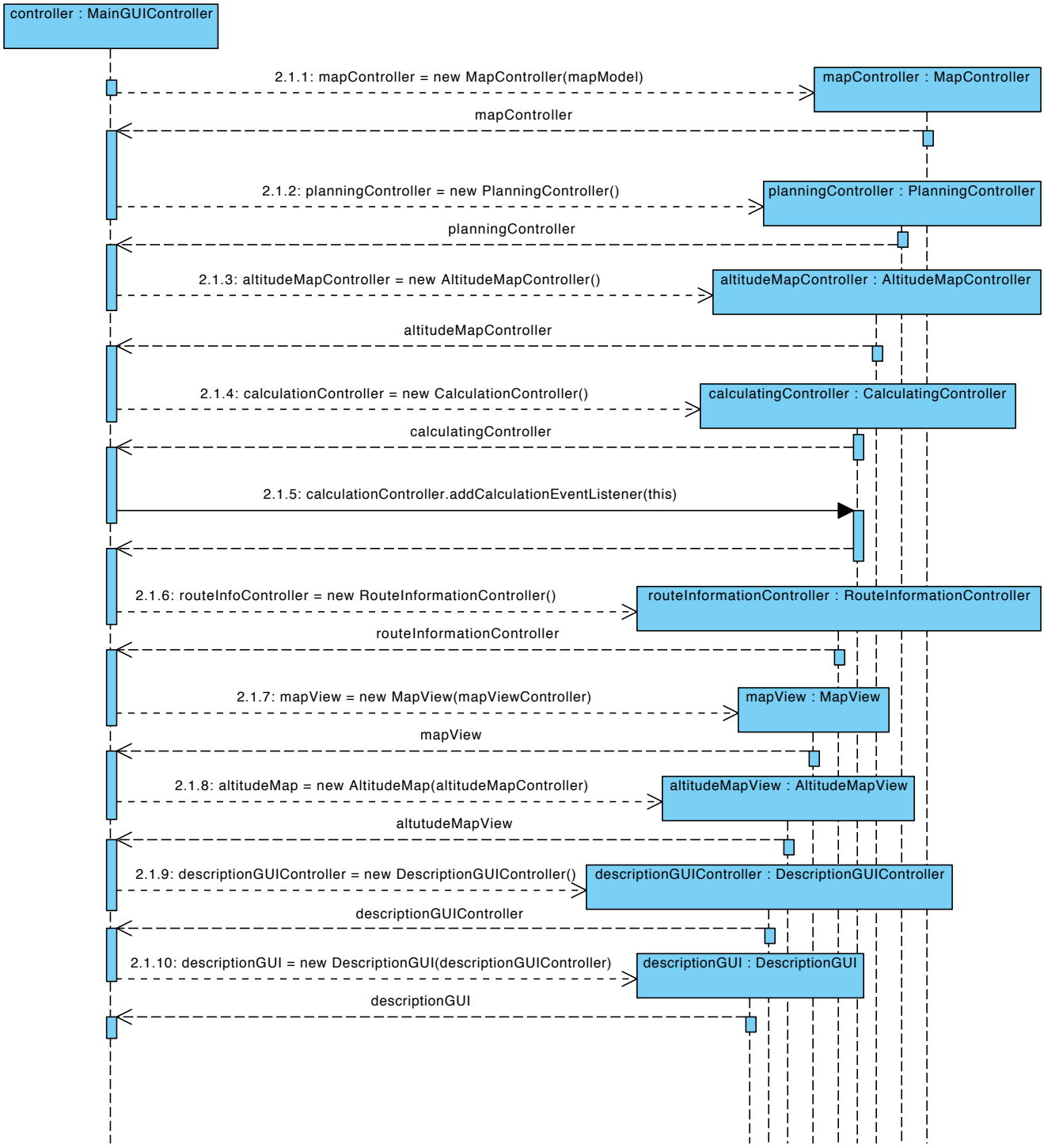


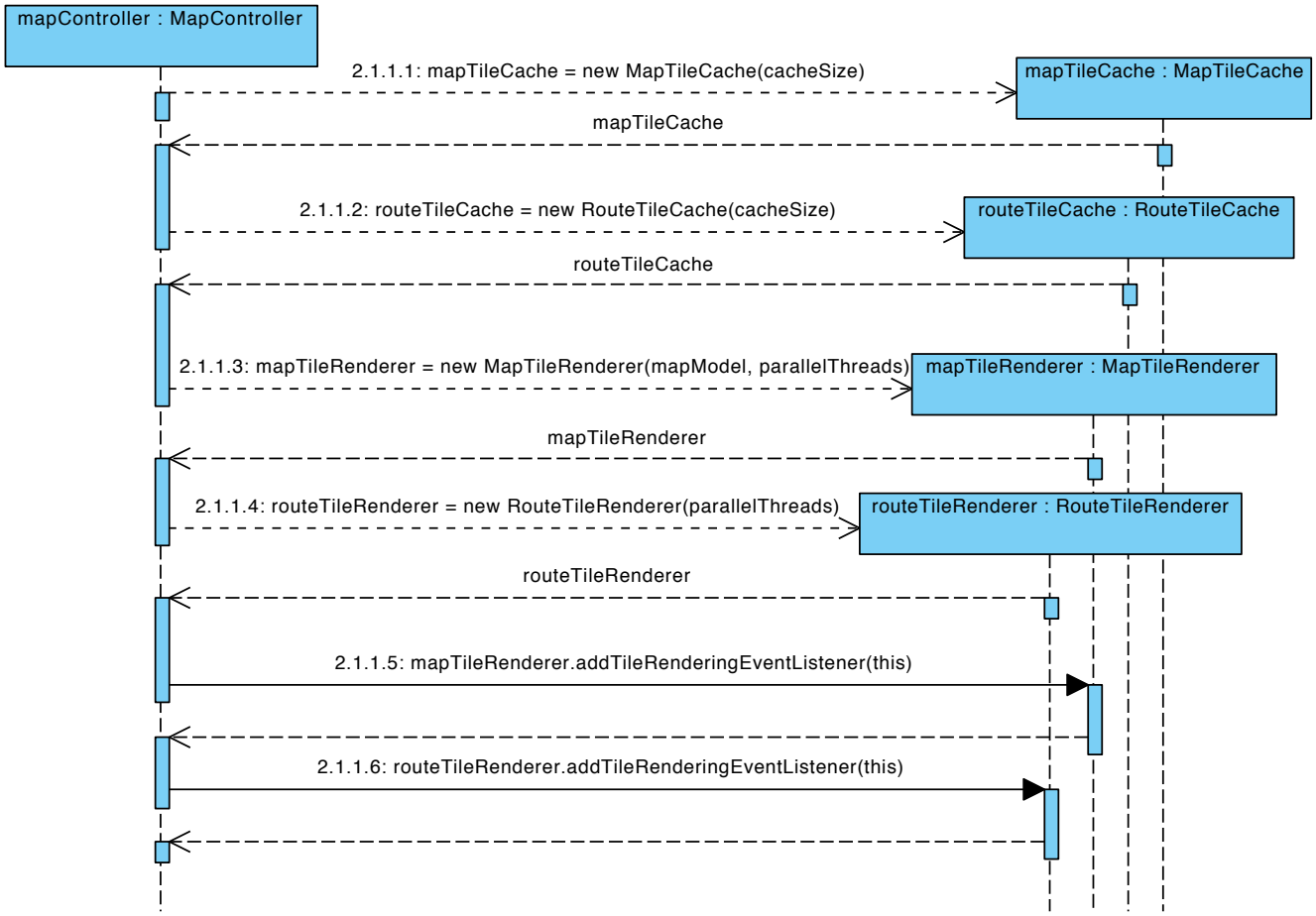

Visual Paradigm for UML Community Edition [not for commercial use]
User
: MainController
ArcFlagCalculator
: Exporter
: PreprocessingDijsktra
7.30:
7.31: setMapModel(m: MapModel)
7.32:
7.33: addArcFlagsToMapModel()
loop
7.33.1: computeArcFlags(startNode: Node, model : MapModel)
7.33.2:
7.34:
7.35: saveMapModelToDir(path : String, model : MapModel)
7.36:
8: Output: Preprocessing finished

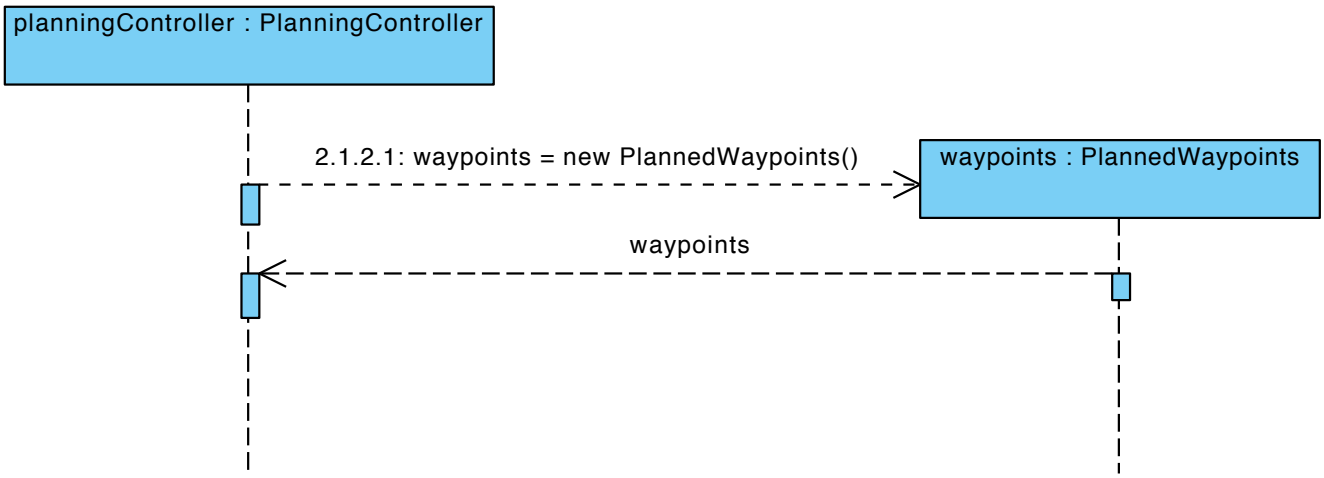
6.2 Systeminitialisierung

Das folgende Diagramm beschreibt den Ablauf direkt nach dem Programmstart. Dies umfasst unter anderem die Kartendaten zu laden, Controller zu initialisieren und Listener zu registrieren.



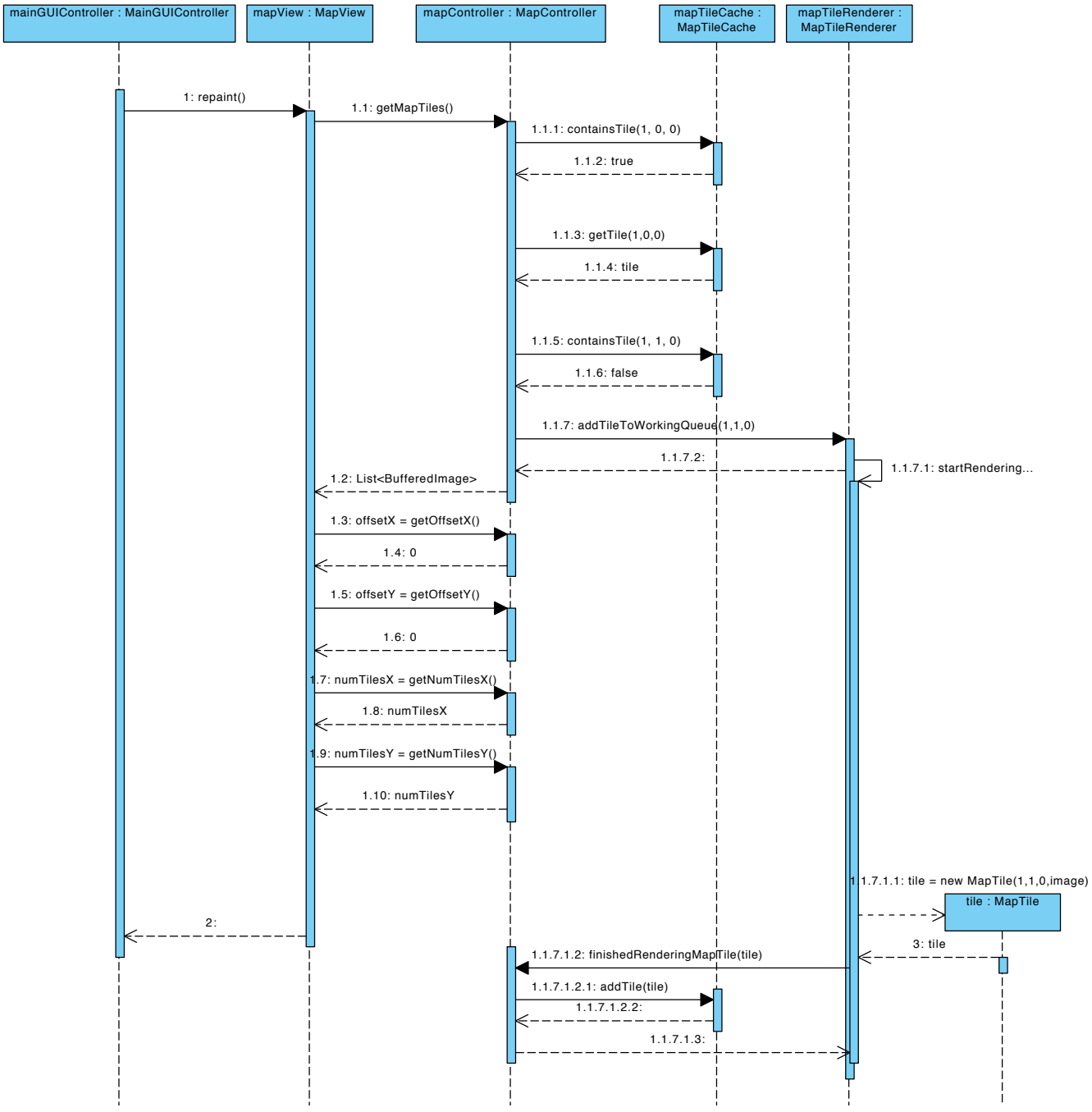

controller : MainGUIController
2.1.1: mapController = new MapController(mapModel)
mapController : MapController
mapController
2.1.2: planningController = new PlanningController()
planningController : PlanningController
planningController
2.1.3: altitudeMapController = new AltitudeMapController()
altitudeMapController : AltitudeMapController
altitudeMapController
2.1.4: calculationController = new CalculationController()
calculatingController : CalculatingController
calculatingController
2.1.5: calculationController.addCalculationEventListener(this)
2.1.6: routeInfoController = new RouteInformationController()
routeInformationController : RouteInformationController
routeInformationController
2.1.7: mapView = new MapView(mapViewController)
mapView : MapView
mapView
2.1.8: altitudeMap = new AltitudeMap(altitudeMapController)
altitudeMapView : AltitudeMapView
altutudeMapView
2.1.9: descriptionGUIController = new DescriptionGUIController()
descriptionGUIController : DescriptionGUIController
descriptionGUIController
2.1.10: descriptionGUI = new DescriptionGUI(descriptionGUIController)
descriptionGUI : DescriptionGUI
descriptionGUI


mapController : MapController
2.1.1.1: mapTileCache = new MapTileCache(cacheSize)
mapTileCache : MapTileCache
mapTileCache
2.1.1.2: routeTileCache = new RouteTileCache(cacheSize)
routeTileCache : RouteTileCache
routeTileCache
2.1.1.3: mapTileRenderer = new MapTileRenderer(mapModel, parallelThreads)
mapTileRenderer : MapTileRenderer
mapTileRenderer
2.1.1.4: routeTileRenderer = new RouteTileRenderer(parallelThreads)
routeTileRenderer : RouteTileRenderer
routeTileRenderer
2.1.1.5: mapTileRenderer.addTileRenderingEventListener(this)
2.1.1.6: routeTileRenderer.addTileRenderingEventListener(this)


planningController : PlanningController
2.1.2.1: waypoints = new PlannedWaypoints()
waypoints : PlannedWaypoints
waypoints

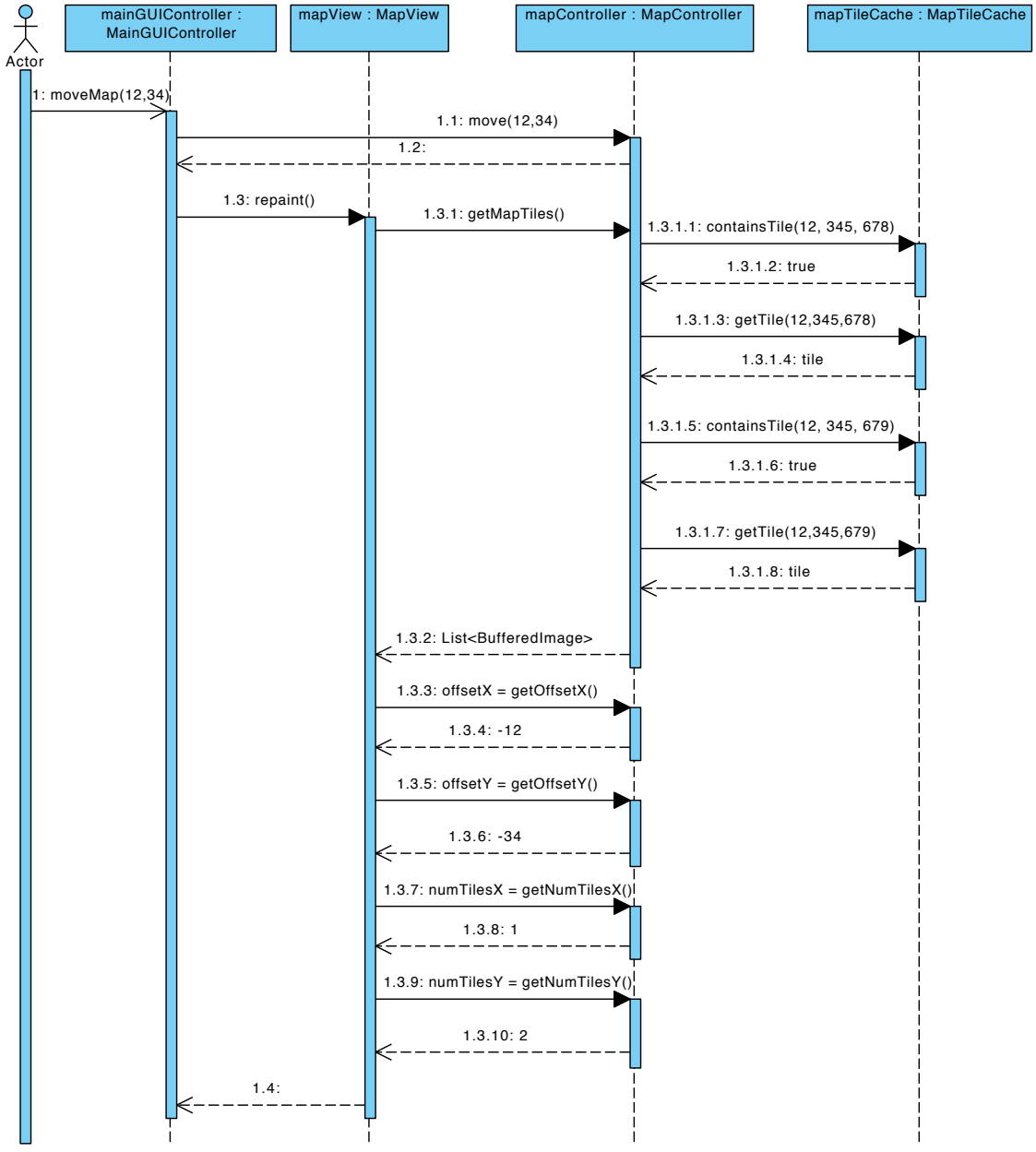
6.3 Die Karte darstellen

Hier wird der Ablauf beim Neuzeichnen der Karte dargestellt. Es wird exemplarisch einmal eine Kachel angefordert, die im Cache schon vorhanden ist und einmal eine, die erst noch neu gerendert werden muss.



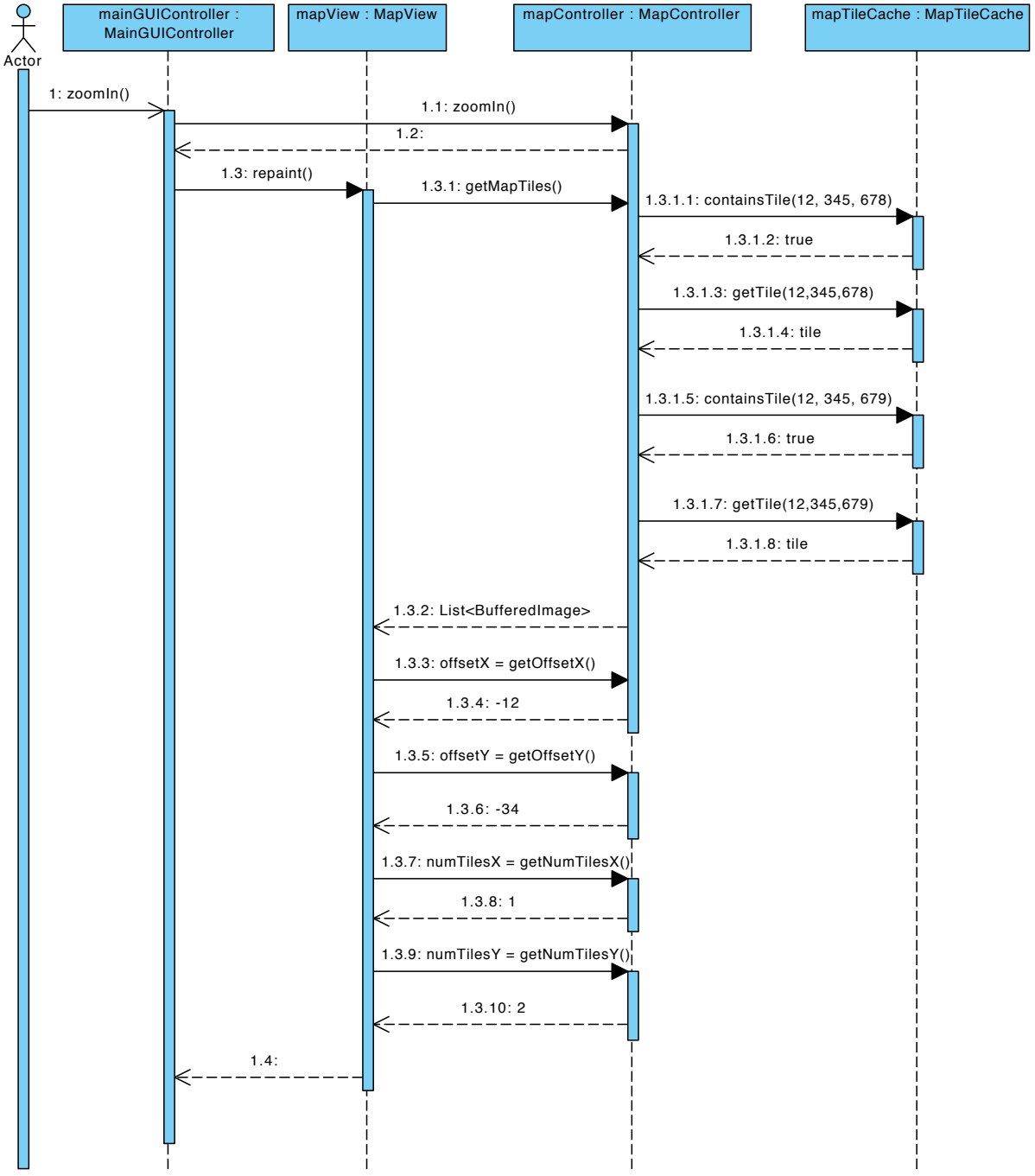
6.4 Die Karte verschieben

Im folgenden wird exemplarisch das Verschieben der Karte dargestellt. Die Karte wird um 12 Pixel in x- und und um 34 Pixel in y-Richtung verschoben. Anschließend wird die Karte neu gezeichnet, wozu zwei Kacheln aus dem Cache geladen werden. Im Beispiel liegen keine Wegpunkte und keine berechnete Route vor.



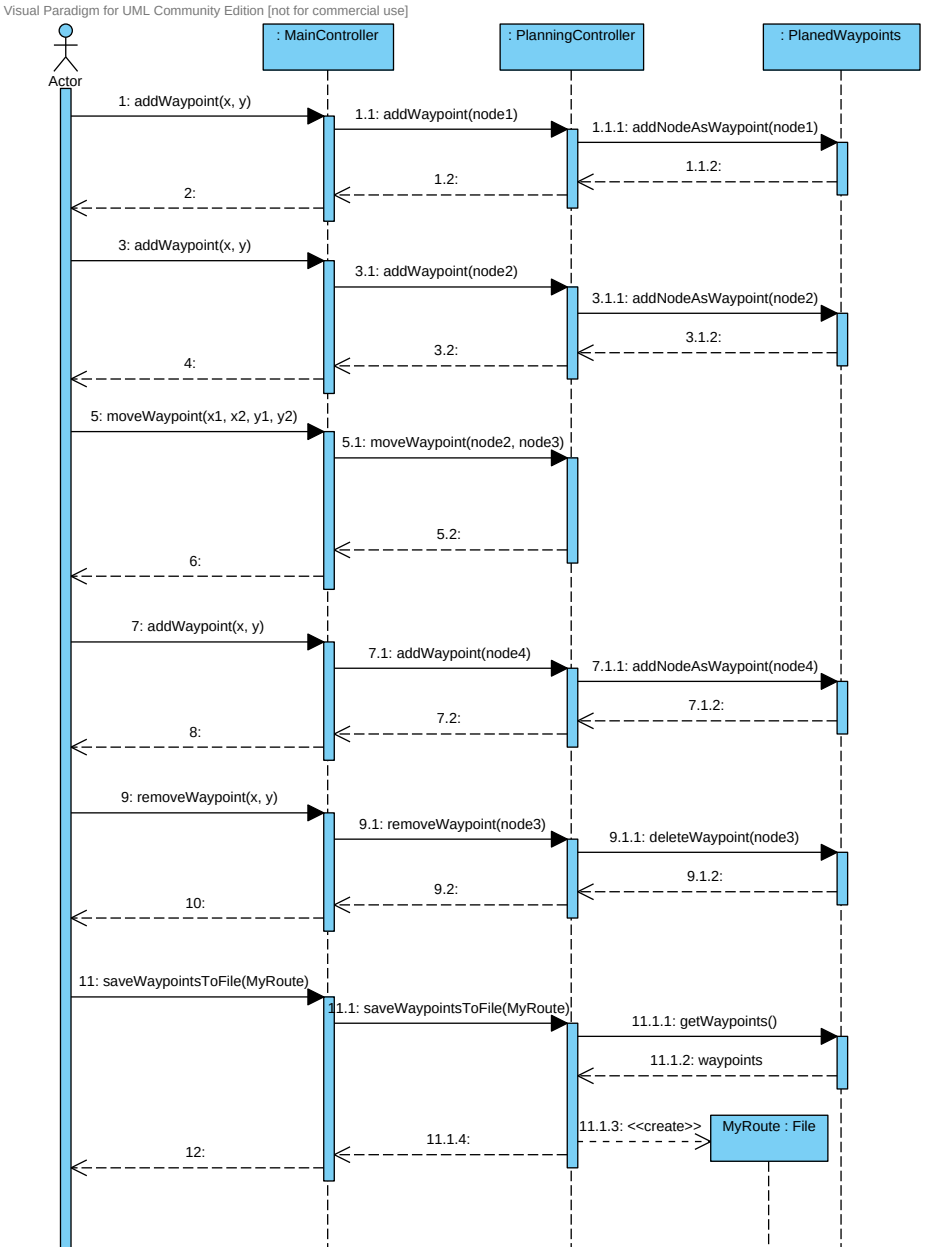
6.5 Die Karte vergrößern/verkleinern

Im folgenden wird exemplarisch das Vergrößern der Karte dargestellt. Anschließend wird die Karte neu gezeichnet, wozu zwei Kacheln aus dem Cache geladen werden. Im Beispiel liegen keine Wegpunkte und keine berechnete Route vor.



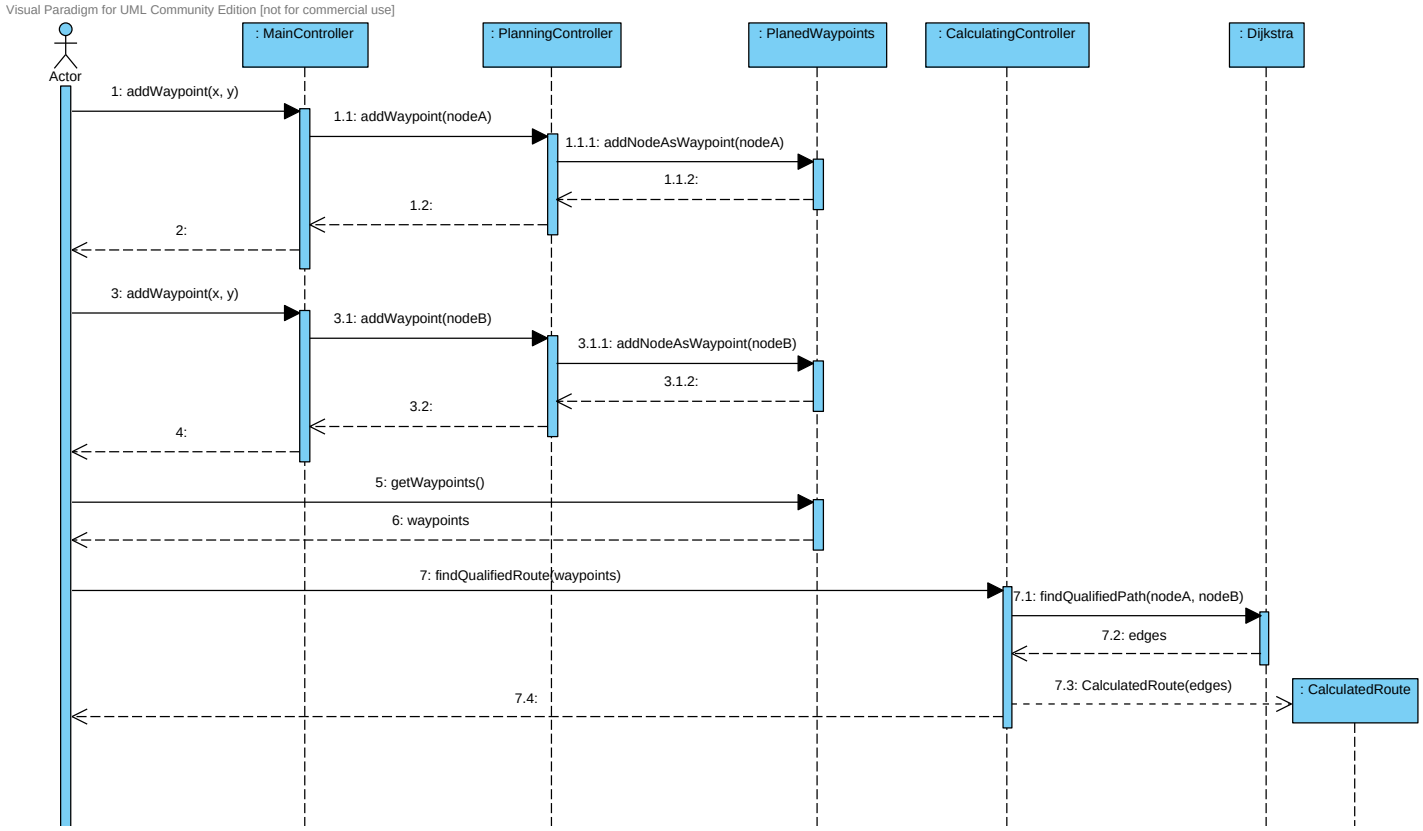
6.6 Route planen und anschließend speichern

Im folgenden Sequenzdiagramm wird eine beispielhafte Planung einer Route dargestellt, welche anschließend abgespeichert wird. Hierbei werden zu Beginn zwei Wegpunkte vom Nutzer hinzugefügt, wobei der zweite Wegpunkt direkt danach verschoben wird. Zusätzlich wird noch ein weiterer Wegpunkt hinzugefügt, sowie ein anderer wieder entfernt. Abschließend wird die geplante Route, d.h. alle gesetzten Wegpunkte, in einer Datei abgespeichert.



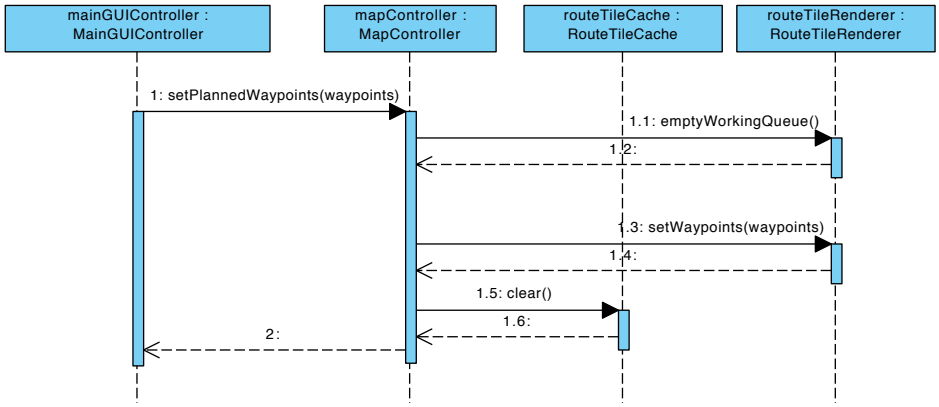
6.7 Geeignete Route berechnen

Das angegebene Sequenzdiagramm beschreibt den einfachen Fall einer Routenberechnung ohne Zwischenstops. Hierbei fügt der Nutzer zwei Wegpunkte hinzu. Darauffolgend wird von ihm die Routenberechnung angestoßen und es wird eine geeignete Route berechnet.

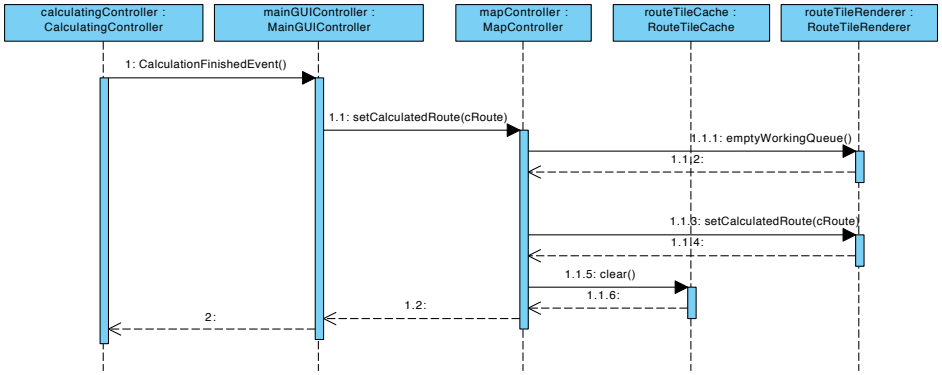


6.8 Route und Wegpunkte aktualisieren

Nach jeder Änderung, die über den MainGUIController an den Wegpunkten vorgenommen wird, aktualisiert dieser die Wegpunkte des MapControllers:

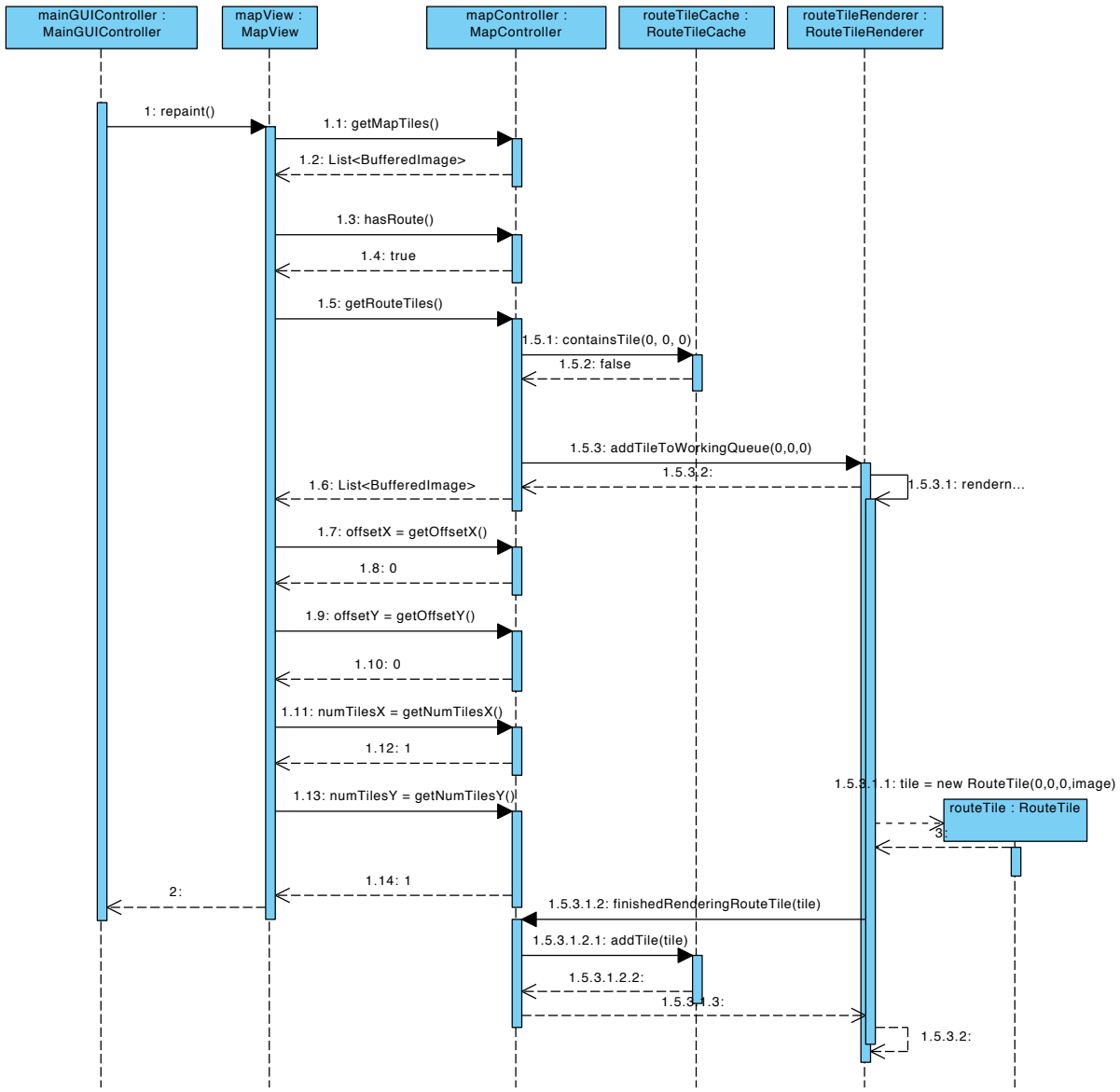


Nachdem der CalculationController in einem separaten Thread die Berechnung der geplanten Route abgeschlossen hat, benachrichtet er alle, bei ihm angemeldeten CalculationListener. In dem folgenden Beispiel wird der MainGUIController benachrichtigt, welcher wiederum die berechnete Route seines MapControllers aktualisiert.



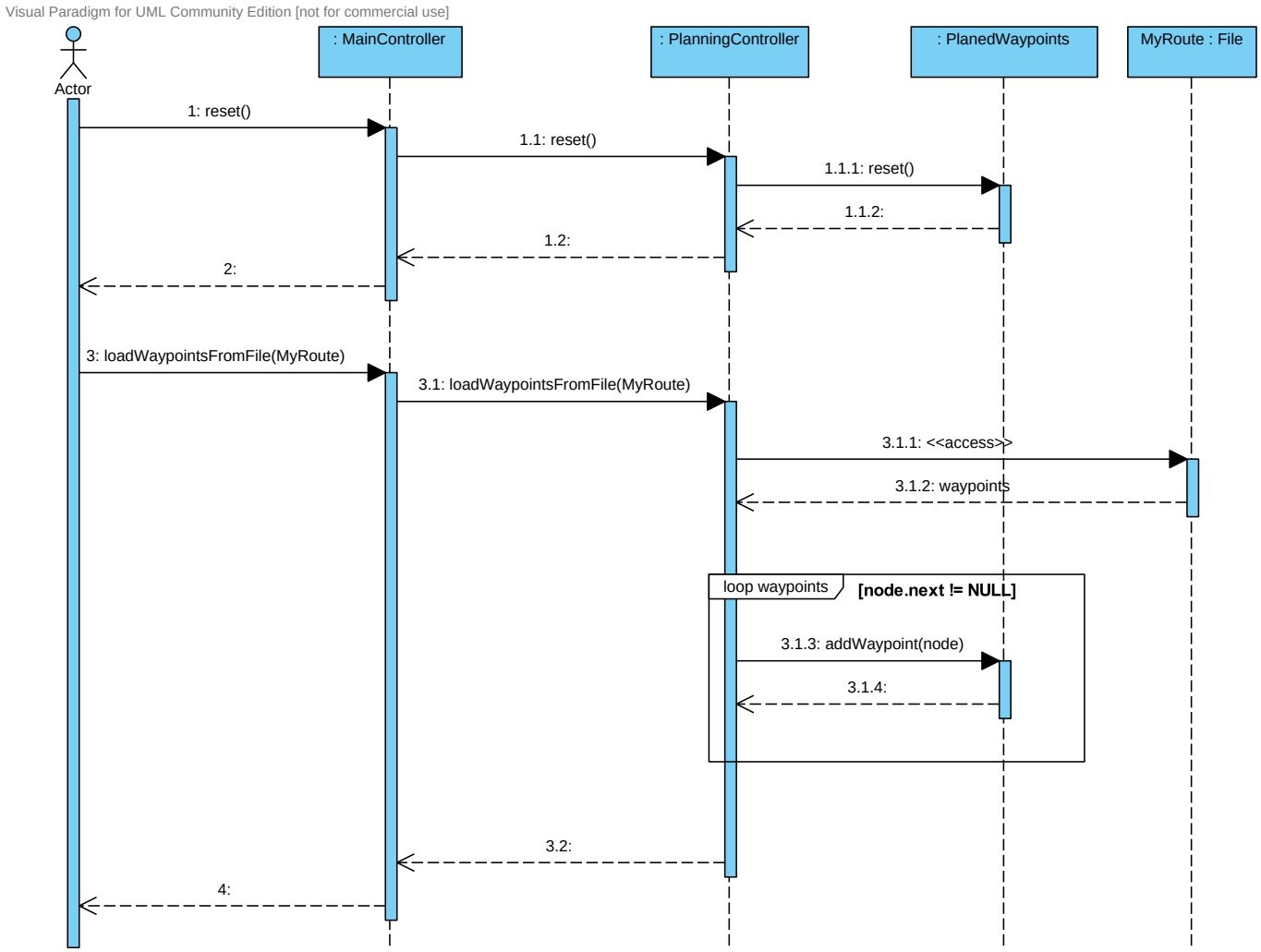
6.9 Berechnete Route auf der Karte darstellen

Im folgenden Sequenzdiagramm wird beispielhaft das Anzeigen der Route dargestellt. Nachdem sich der mapView nach *1.13* alle benötigten Informationen geholt hat, zeichnet er zuerst das Bild der Karten-Kachel. Anschließend wird das Bild der Routen-Kachen, welches bis auf die Route transparent ist darüber gezeichnet. Da in diesem Beispiel die Routen-Kachel erst gerendert werden muss, was in einem separaten Thread passiert, wird die Route erst beim nächsten zeichnen der Karte sichtbar sein.



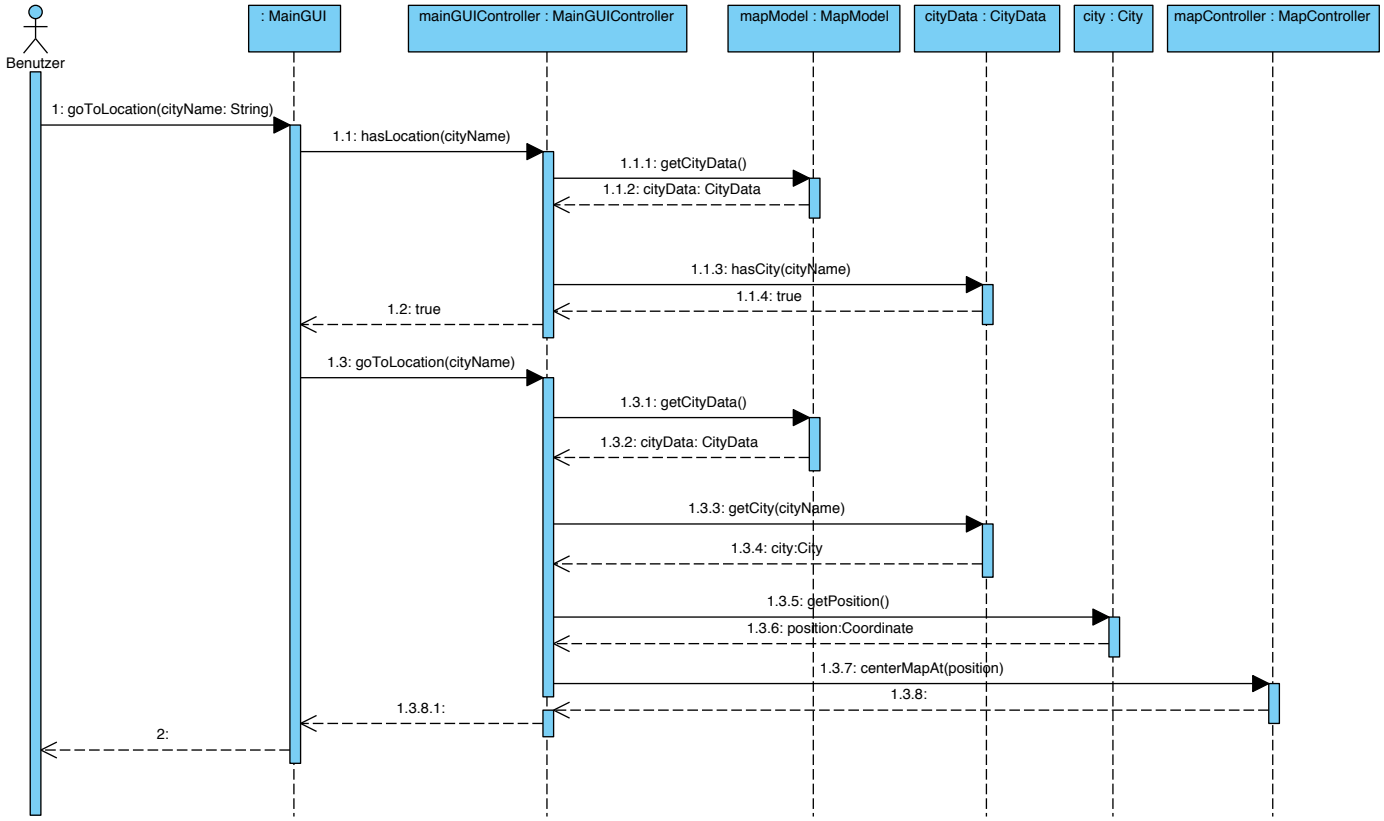
6.10 Laden einer Route

Beim Laden einer Route werden zu Beginn die aktuell geplanten Wegpunkte zurückgesetzt. Im Anschluss daran, werden aus der angegebenen Datei, die abgespeicherten Wegpunkte in Form einer Liste geladen und nacheinander den geplanten Wegpunkten hinzugefügt.



6.11 Kartenausschnitt zu Stadt verschieben

Hier wird dargestellt, was passiert, wenn der Benutzer den Kartenausschnitt zu einer Stadt bewegen will, die im System bekannt ist.



7 Glossar

Altitude Höhe über Normal-Null

Arc-Flags Beschleunigungstechnik für den Dijkstra Algorithmus zur schnelleren Suche des kürzesten Pfades

Dijkstra-Algorithmus Algorithmus zur Berechnung einer kürzesten Route zwischen zwei Punkten, Name geht auf Erfinder Edsger W. Dijkstra zurück

Entwurfsmuster Entwurfsmuster (engl. design patterns) sind bewährte Lösungsschablonen für wiederkehrende Entwurfsprobleme in Softwarearchitektur und Softwareentwicklung.

Graph Stellt eine Menge von Objekten dar, sowie die Verbindungen die zwischen diesen stehen

GUI Benutzeroberfläche

Höhenverlaufsdiagramm Diagramm zur Anzeige der Höhe in Abhängigkeit zur Distanz

Java Programmiersprache

Knoten-ID eindeutige Nummer eines Knotens zur Identifizierung

Mercator-Projektion Zylinderprojektion, welche Länge- und Breitengrade auf eine ebene rechteckige Karte abbildet, genaueres: http://de.wikipedia.org/wiki/Mercator-Projektion

OpenStreetMap für jeden zugängliches Projekt, das weltweite geographische Daten sammelt

Route Pfad zwischen mindestens zwei verschiedenen Wegpunkten

Rückwärts-Dijkstra Ein Dijkstra auf einem Graphen mit umgekehrten Kanten

SRTM-Bilddatei Digitale Karte die Informationen über Höhenmeter enthält

Tile Eine Kartenkachel, d.h. ein quadratischer Ausschnitt der Karte

UML Unified Model Language, wird benutzt um Programme möglichst übersichtlich zu beschreiben

Wegpunkt Ein Punkt, der auf einer Route liegt

Zoomen Vergrößern/Verkleinern eines Bildschirmausschnittes

Zoomstufe gibt den Grad der Vergrößerung/Verkleinerung an