routeKIT

Implementierungsbericht

24. März 2014

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

Dieses Dokument erläutert die Implementierung der Anwendung *routeKIT*. Es dokumentiert die durchgeführten Änderungen am Entwurf, die zur Realisierung der Implementierung notwendig waren. Außerdem beschreibt es die Arbeitsplanung und den tatsächlichen Arbeitsablauf der Implementierung.

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 Änderungen am Entwurf

Im Folgenden sind die gegenüber dem ursprünglichen Entwurf geänderten und neu hinzugefügten Klassen, Methoden und Attribute aufgeführt.

2.1 (ohne Paket)

`routeKIT` Neue Klasse. Existiert ausschließlich, um den kurzen Aufruf `java routeKIT` zu ermöglichen.

Methoden

`main` Leitet weiter auf `MainController.main` .
Parameter:

`args` Kommandozeilenargumente. Typ: `String[]`

2.2 Paket Controllers

Es wurde eine neue Klasse ↗`ProfileMapManager` eingeführt, die alle dem Programm bekannten ↗Vorberechnungen halten und sie speichern und laden.

`ProfileManager`

Methoden

`init` (statisch) Initialisiert den `ProfileManager`, indem alle in ↗`rootDirectory` liegenden `.profile`-Dateien `geladen` werden. Darf nur einmal aufgerufen werden.
Parameter:

`rootDirectory` siehe ↗`ProfileMapManager.init.rootDirectory`. Typ: `File`

`getInstance` (statisch) Gibt die `ProfileManager`-Instanz zurück. ↗`init` muss zuvor aufgerufen worden sein.
Für die Diskussion des ↗Einzelstück-Entwurfsmusters siehe ↗`ProfileMapManager.init`. Rückgabetyp: ↗`ProfileManager`

`MapManager`

Methoden

`init` (statisch) Initialisiert den `MapManager`, indem alle in ↗`rootDirectory` liegenden Ordner ↗`geladen` werden. Darf nur einmal aufgerufen werden.
Parameter:

`rootDirectory` siehe ↗`ProfileMapManager.init.rootDirectory` Typ: `File`

`getInstance` (statisch) Gibt die `MapManager`-Instanz zurück. ↗`ProfileManager.init` muss zuvor aufgerufen worden sein.
Für die Diskussion des ↗Einzelstück-Entwurfsmusters siehe ↗`ProfileMapManager.init`. Rückgabetyp: ↗`MapManager`

`ProfileMapManager` Neue Klasse. Der `ProfileMapManager` verwaltet alle ↗Vorberechnungen: er lädt sie beim Programmstart von der Festplatte, löscht sie auf Anfrage, speichert sie und hält eine Liste aller ↗Vorberechnungen. Außerdem kennt er die aktuell ausgewählte ↗`ProfileMapCombination` (die nicht zwangsläufig vorberechnet ist).

Methoden

`init` (statisch) Initialisiert den `ProfileMapManager`, indem der ↗`ProfileManager` und der ↗`MapManager` initialisiert werden und eine Index-Datei (`routeKIT.idx`) gelesen wird, die alle Vorberechnungen auflistet und die aktuelle ↗`ProfileMapCombination` angibt.
Der Rückgabewert ist die aktuelle ↗`ProfileMapCombination`.
Parameter:

`rootDirectory` Das Verzeichnis, in dem die Daten dieser *routeKIT*-Installation liegen. Typ: `File`

Rückgabetyp: ↗`ProfileMapCombination`

`getInstance` (statisch) Gibt die `ProfileMapManager`-Instanz zurück. ↗`init` muss zuvor aufgerufen worden sein.
Es handelt sich hierbei um eine Variante des ↗Einzelstück-Entwurfsmusters: ↗`init` wird normalerweise durch ↗`getInstance` implizit durchgeführt, wenn noch keine Instanz vorhanden ist. Hier wurden die beiden Methoden getrennt, da es sich bei ↗`init` um eine verhältnismäßig teure Operation handelt (beinhaltet Festplattenaktivität), die nicht beliebig beim ersten Aufruf von ↗`getInstance` stattfinden sollte, sondern nur an einer bestimmten Stelle während des Programmstarts. Rückgabetyp: ↗`ProfileMapManager`

`getPrecalculations` Gibt alle dem `ProfileMapManager` bekannten ↗Vorberechnungen zurück. Rückgabetyp: `Set<ProfileMapCombination>`

`getCurrentCombination` Gibt die aktuell ausgewählte ↗`ProfileMapCombination` zurück. Diese muss nicht unbedingt ein Element aus ↗`getPrecalculations` sein, da sie möglicherweise nicht vorberechnet ist. Rückgabetyp: ↗`ProfileMapCombination`

`addCurrentCombinationListener` Registriert einen ↗`CurrentCombinationListener`, der bei Änderungen der aktuellen ↗`ProfileMapCombination` benachrichtigt werden soll.
Parameter:

`listener` Der Listener, der benachrichtigt werden soll. Typ: ↗`CurrentCombinationListener`

`selectProfileAndMap` Wählt das angegebene Profil und die angegebene Karte aus. Existiert eine ↗Vorberechnung für dieses Profil und diese Karte, so wird sie verwendet, ansonsten wird eine neue ↗`ProfileMapCombination` erstellt.
Gibt die ausgewählte ↗`ProfileMapCombination` zurück. Diese Methode ersetzt die im Entwurf in `MainController` angegebenen Methoden selectProfile(:Profile) und selectMap(:StreetMap).
Parameter:

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

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

Rückgabetyp: ↗`ProfileMapCombination`

`getPrecalculation` Sucht nach einer Vorberechnung für dieses Profil und diese Karte und gibt sie zurück, ansonsten wird `null` zurückgegeben.

Parameter:

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

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

Rückgabetyp: ↗`ProfileMapCombination`

`setCurrentCombination` Setzt die angegebene ↗`ProfileMapCombination` als aktuelle Kombination. Ist sie vorberechnet, so wird sie gespeichert und eine neue Index-Datei wird geschrieben. Außerdem werden registrierte ↗`CurrentCombinationListener` benachrichtigt.

Parameter:

`combination` Die ausgewählte ↗`ProfileMapCombination`. Typ: ↗`ProfileMapCombination`

`savePrecalculation` Speichert die angegebene Vorberechnung in der internen Liste und auf der Festplatte. Dies ist nur zulässig, wenn es sich dabei überhaupt um eine Vorberechnung handelt (und nicht etwa um eine ↗`ProfileMapCombination` ohne Vorberechnung). Schreibt eine neue Index-Datei.

Parameter:

`precalculation` Die ↗`ProfileMapCombination`, die gespeichert werden soll. Typ: ↗`ProfileMapCombination`

`deletePrecalculation` Löscht die angegeben Vorberechnung aus der internen Liste und, falls nicht anders angegeben, von der Festplatte. Dies ist nur zulässig, wenn es sich dabei überhaupt um eine Vorberechnung handelt (und nicht etwa um eine ↗`ProfileMapCombination` ohne Vorberechnung). Schreibt eine neue Index-Datei. Die zweite Variante ist nur dann sinnvoll, wenn alle ↗Vorberechnungen einer ↗Karte zusammen mit der Karte gelöscht werden sollen; da sie in Unterordnern der Karte gespeichert sind, würden sie durch das Löschen des Karten-Ordners ohnehin gelöscht.

Parameter:

`precalculation` Die ↗`ProfileMapCombination`, die gelöscht werden soll. Typ: ↗`ProfileMapCombination`

`deleteFromDisk` *optional*: Wenn `false`, wird die ↗Vorberechnung nicht von der Festplatte gelöscht. Typ: `boolean`

`MainController`

Methoden

`MainController`

Parameter:

`pr` Wird zum Anzeigen des Fortschritts beim Start des Programms gebraucht. Typ: ↗`ProgressReporter`

`getInstance` (statisch) Gibt die Instanz des `MainControllers` zurück, der ein ↗Einzelstück ist. Rückgabetyp: ↗`MainController`

`getHistory` Gibt den Verlauf zurück. Rückgabetyp: `History`

`startPrecalculation`
Parameter:

`reporter` Geändert: Der Parameter `combination` fällt weg; stattdessen wird immer die aktuelle ↗`ProfileMapCombination` berechnet. ↗`reporter` wird über den Fortschritt der Vorberechnung informiert. Typ: ↗`ProgressReporter`

`ProfileManagerController`

Methoden

`getDeletedPrecalculations` Gibt alle zu löschenden ↗Vorberechnungen zurück, damit sie dem Benutzer vor dem Löschen angezeigt werden. Rückgabetyp: `Set<ProfileMapCombination>`

`getSelectedProfile` Gibt das ↗Profil zurück, das der Benutzer ausgewählt hat. Wenn der Dialog noch nicht oder durch „Abbrechen“ geschlossen wurde, wird `null` zurückgegeben. Rückgabetyp: ↗`Profile`

`MapManagerController`

Methoden

`MapManagerController`
Parameter:

`view` Das Hauptfenster der Anwendung. Typ: ↗`MainView`

`getChanges` Gibt alle Änderungen zurück, die der Benutzer bis jetzt vorgenommen hat, aber die noch nicht bestätigt wurden. Rückgabetyp: `MapManagementDiff`

`getSelectedMap` Gibt die ↗Karte zurück, die der Benutzer ausgewählt hat. Wenn der Dialog noch nicht oder durch „Abbrechen“ geschlossen wurde, wird `null` zurückgegeben. Rückgabetyp: ↗`StreetMap`

`changeMap`
Parameter:

`mapName` Geändert: Die GUI kann keine Objekte vom Typ `Map` übergeben. Die GUI verwaltet nur die Namen der Karten. Typ: `String`

`addProfile` Geändert: Die GUI übergibt nicht mehr das Profil als Parameter, stattdessen öffnet der ↗`MapManagerController` selbst die Profilverwaltung, in welcher der Benutzer dann ein Profil auswählen kann. In der Profilverwaltung können auch Profile und damit Vorberechnungen gelöscht werden, und diese Änderungen müssen dann auch in der ↗`MapManagerView` wieder übernommen werden. Diese Änderungen zu verfolgen, ist nicht Aufgabe der View, sondern des Controllers.

`removeProfile`
Parameter:

`profileName` Geändert: Die GUI kann keine Objekte vom Typ `Profile` übergeben. Die GUI verwaltet nur die Namen der Profile. Typ: `String`

`MapManagerController.MapManagementDiff` Neue Klasse. Innere Klasse von ↗`MapManagerController`, welche die Änderungen, die der Benutzer getätigt hat, kapselt. Sie war ursprünglich nur für den internen Gebrauch in `MapManagerController.saveAllChanges` gedacht, wird jetzt aber auch der ↗`MapManagerView` zugänglich gemacht, welche das Ergebnis im „Bestätigen“-Dialog verwendet und anzeigt.

Die analoge Klasse `ProfileManagerController.ProfilesDiff` ist weiterhin `private`, da sie außerhalb des ↗`ProfileManagerController`s nicht benötigt wird.

Methoden

`getNewOrUpdatedMaps` Alle Karten, die importiert oder aktualisiert wurden, also neu importiert werden müssen.
(`FutureMap` ist eine interne Klasse, die nur den vom Benutzer angegebenen Dateinamen hält.) Rückgabetyp: `Set<FutureMap>`

`getDeletedMaps` Alle Karten, die gelöscht werden müssen (auch die alten Versionen von aktualisierten Karten). Rückgabetyp: `Set<StreetMap>`

`getDeletedPrecalculations` Alle Vorberechnungen, die gelöscht werden müssen (weil ihre Karten aktualisiert oder gelöscht wurden, oder weil die Vorberechnung explizit gelöscht wurde). Rückgabetyp: `Set<ProfileMapCombination>`

`getNewPrecalculations` Alle Vorberechnungen, die durchgeführt werden müssen (neue und solche mit aktualisierter Karte). Rückgabetyp: `Set<ProfileMapCombination>`

`CLI` Neue Klasse. Kapselt das ↗Command Line Interface (CLI). Implementiert ↗`ProgressListener`.

Methoden

`doImport` Importiert eine Karte und führt anschließend eine Vorberechnung für diese Karte durch.
Parameter:

`mapPath` Der Pfad zur Kartendatei, die importiert werden soll. Typ: `String`

`mapName` Der Name der Karte, die importiert werden soll. Typ: `String`

`profileName` Der Name des Profils, für das auf der importierten Karte eine Vorberechnung durchgeführt werden soll. Typ: `String`

2.3 Paket Precalculation

Die Klassen `ArkFlagCalculator` und ↗`EdgeWeighter` wurden zu Interfaces, die von den neuen Klassen `ArcFlagCalculatorImpl`, `ArcFlagCalculatorParallel` und ↗`EdgeWeighterImpl` implementiert werden. Vielen Klassen wird nun ein ↗`ProgressReporter` als Parameter übergeben.

`PreCalculator` Führt die Vorberechnung für die angegebene Kombination aus.

Methoden

`doPrecalculation` Führt die Vorberechnung für die angegebene Kombination durch.
Parameter:

`reporter` Wird zum Anzeigen des Fortschritts bei der ↗Arc-Flags-Berechnung gebraucht. Typ: ↗`ProgressReporter`

`ArcFlagsCalculator` Geändert: Ein Interface für die Berechnung der ↗Arc-Flags

Methoden

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

`reporter` Wird zum Anzeigen des Fortschritts bei der ↗Arc-Flags-Berechnung gebraucht. Typ: ↗`ProgressReporter`

`ArcFlagsCalculatorImpl` Neue Klasse. Ersetzt die Klasse `ArcFlagsCalculator` aus dem Entwurf. Implementiert das Interface ↗`ArcFlagsCalculator`

Methoden

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

`reporter` Wird zum Anzeigen des Fortschritts bei der ↗Arc-Flags-Berechnung gebraucht. Typ: ↗`ProgressReporter`

`ArcFlagsCalculatorParallel` Neue Klasse. Verwendet im Gegensatz zu ↗`ArcFlagsCalculatorImpl` mehrere parallel laufende Threads, um die Vorberechnung zu beschleunigen. Implementiert das Interface ↗`ArcFlagsCalculator`

Methoden

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

`reporter` Wird zum Anzeigen des Fortschritts bei der ↗Arc-Flags-Berechnung gebraucht. Typ: ↗`ProgressReporter`

`EdgeWeighter` Geändert: Ein Interface für die Berechnung der Kantengewichte.

Methoden

`weightEdges` Versieht den kantenbasierten Graphen mit Kantengewichten.
Parameter:

`reporter` Wird zum Anzeigen des Fortschritts bei der Gewichte-Berechnung gebraucht. Typ: ↗`ProgressReporter`

`EdgeWeighterImpl` Neue Klasse. Ersetzt die Klasse `EdgeWeighter` aus dem Entwurf. Implementiert das Interface ↗`EdgeWeighter`

Methoden

`weightEdges` Versieht den kantenbasierten Graphen mit Kantengewichten.
Parameter:

`reporter` Wird zum Anzeigen des Fortschritts bei der Gewichte-Berechnung gebraucht. Typ: ↗`ProgressReporter`

`OSMMapImporter` Neuer Name für die Klasse `MapImporter`, welche nun die Schnittstelle `MapImporter` implementiert.

`OSMParser`

Methoden

`parseOSM` Geändert: Der Parameter `name` wurde entfernt. Der Name der Karte wird erst später vom `MapImporter` gesetzt.

`OSMWay`

Methoden

`isReversedOneway` Bestimmt, ob es sich um eine Einbahnstraße entgegen der Wegrichtung handelt (OSM-Tag `oneway=-1`). Rückgabetyp: `boolean`

`isHighwayLink` Bestimmt, ob es sich um eine Anschluss- bzw. Verbindungsstraße (z. B. Auf- oder Abfahrt) handelt. Rückgabetyp: `boolean`

`MapEdge`

Attribute

`targetNode` Geändert: Speichert nun direkt den Zielknoten und nicht mehr dessen ID in der OSM-Datei. Dadurch werden unnötige Hashtabellenabfragen in ↗`OSMParser` vermieden. Typ: `Node`

`Id` Der interne Bezeichner der Kante. Typ: `Edge`

`TurnRestriction`

Attribute

`from` Die OSM-Way-ID des Wegs, von dem die Abbiegebeschränkung definiert ist. Typ: `int`

`to` Die Kante, auf die die Abbiegebeschränkung definiert ist. Typ: ↗`MapEdge`

Methoden

`TurnRestriction`
Parameter:

`from` Die ID des OSM Wegs von dem ein Abbiegevorgang beschränkt ist. Typ: `int`

`to` Eine Kante auf die ein Abbiegevorgang beschränkt ist. Typ: ↗`MapEdge`

`allowsTo` Gibt an, ob diese Abbiegebeschränkung das Abbiegen auf die angegebene Kante erlaubt oder nicht.
Parameter:

`to` Die Kante, auf die abgebogen werden soll. Typ: ↗`MapEdge`

Rückgabetyp: `boolean`

2.4 Paket RouteCalculator

`Route` Diese Klasse implementiert nun die Schnittstelle `Iterable`.

Methoden

`iterator` Gibt einen Iterator über die Koordinaten der Routenpunkte einschließlich Start- und Zielpunkt zurück. Diese Methode ersetzt `getNodeIterator`. Rückgabetyp: `Iterator<Coordinates>`

`FibonacciHeap` Neue Klasse. Diese Klasse stellt einen Fibonacci-Heap dar, der vom `ArcFlagsDijkstra` und der Vorberechnung genutzt wird.

Methoden

`isEmpty` Prüft, ob der Fibonacci-Heap leer ist. Rückgabetyp: `boolean`

`getSize` Gibt die Anzahl der Elemente, die sich gerade im Fibonacci-Heap befinden, zurück. Sehr nützlich z. B. für Testfälle. Rückgabetyp: `int`

`add` Fügt einen neuen Eintrag mit dem gegebenen Wert und der Priorität in den Fibonacci-Heap ein und gibt das neu eingefügte Element zurück.
Parameter:

`value` Der Wert für den neuen Eintrag. Typ: `int`

`priority` Die Priorität für den neuen Eintrag. Typ: `int`

Rückgabetyp: ↗`FibonacciHeapEntry`

`deleteMin` Entfernt das Element mit der niedrigsten Priorität aus dem Fibonacci-Heap und gibt es zurück. Rückgabetyp: ↗`FibonacciHeapEntry`

`decreaseKey` Aktualisiert die Priorität eines gegebenen Eintrags.
Parameter:

`entry` Der Eintrag, welcher aktualisiert werden soll. Typ: ↗`FibonacciHeapEntry`

`newPriority` Die neue Priorität für den Eintrag. Typ: `int`

`FibonacciHeapEntry` Neue Klasse. Stellt einen Eintrag des ↗`FibonacciHeap` dar.

Attribute

`degree` Der Grad eines Eintrags. Typ: `int`

`marked` Legt fest, ob ein Eintrag markiert wurde. Typ: `boolean`

`parent` Der Eltern-Eintrag zur diesem Eintrag. Typ: ↗`FibonacciHeapEntry`

`child` Der Kind-Eintrag zur diesem Eintrag. Typ: ↗`FibonacciHeapEntry`

`next` Der nächste Eintrag. Typ: ↗`FibonacciHeapEntry`

`prev` Der vorherige Eintrag. Typ: ↗`FibonacciHeapEntry`

`value` Speichert den Wert des Eintrags. Typ: `int`

`priority` Speichert die Priorität des Eintrags. Typ: `int`

Methoden

`FibonacciHeapEntry` Konstruktor für eine FibonacciHeapEntry.
Parameter:

`value` Der Wert für den neuen Eintrag. Typ: `int`

`priority` Die Priorität für den neuen Eintrag. Typ: `int`

`increaseDegree` Erhöht den Grad des Eintrags um eins.

`decreaseDegree` Verringert den Grad des Eintrags um eins.

2.5 Paket MapDisplay

`TileCache`

Methoden

`stop` Bewirkt, das dieser aktuelle `TeileCache` aufhört Anfragen zu bearbeiten.

`waitForStop` Bewirkt, das dieser aktuelle `TeileCache` aufhört Anfragen zu bearbeiten und wartet, bis alle threads gestorben sind.

2.6 Paket Models

`ProfileMapCombination`

Methoden

`save` Speichert die Vorberechnung in den angegebenen Ordner.
Parameter:

`directory` Der Ordner, in den geschrieben werden soll. Typ: `File`

`load` (statisch) Lädt eine Vorberechnung aus dem angegebenen Ordner.
Parameter:

`directory` Der Ordner, aus dem geladen werden soll. Typ: `File`

Rückgabetyp: ↗`ProfileMapCombination`

`loadLazily` (statisch) Gibt eine ↗`ProfileMapCombination` zurück, die erst beim ersten Zugriff auf `getWeights` bzw. `getArcFlags` diese auch aus dem angegebenen Ordner lädt. Diese Methode wird verwendet, um die Startzeit des Programms zu verbessern: Die Information, welche Vorberechnungen existieren, soll zwar jederzeit verfügbar sein, allerdings werden nicht alle Vorberechnungen sofort benötigt.
Parameter:

`directory` Der Ordner, aus dem geladen werden soll. Typ: `File`

Rückgabetyp: ↗`ProfileMapCombination`

`ensureLoaded` Nach Aufruf dieser Methode ist garantiert, dass auch eine Vorberechnung, welche mittels ↗`loadLazily` geladen wurde, vollständig geladen ist. Spätere Aufrufe von `getWeights` bzw. `getArcFlags` sind dann garantiert schnell.
Parameter:

`reporter` Der ↗`ProgressReporter`, dem der Ladefortschritt gemeldet werden soll. Typ: ↗`ProgressReporter`

`Weights`

Methoden

`save` Speichert die ↗`Weights` in die angegebene Datei.
Parameter:

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

`load` (statisch) Lädt ↗`Weights` aus der angegebenen Datei.
Parameter:

`file` Die Datei, aus der gelesen werden soll. Typ: `File`

Rückgabetyp: ↗`Weights`

`ArcFlags`

Methoden

`save` Speichert die ↗`ArcFlags` in die angegebene Datei.
Parameter:

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

`load` (statisch) Lädt ↗`ArcFlags` aus der angegebenen Datei.
Parameter:

`file` Die Datei, aus der gelesen werden soll. Typ: `File`

Rückgabetyp: ↗`ArcFlags`

`ProgressReporter` Neue Klasse. Dient zur Anzeige vom Fortschritt bei verschiedenen Aktionen, etwa der Vorberechnung, dem Benutzer.

Der ↗`ProgressReporter` ist Aufgaben-basiert: Es gibt eine „Basis“-Aufgabe, die in verschiedene Teilaufgaben unterteilt wird, welche wieder in unterschiedliche Teilaufgaben unterteilt wird und so weiter. Für Aufgaben ohne Teilaufgaben kann dann der Fortschritt auch direkt gesetzt werden.

Methoden

`pushTask` Beginnt eine neue (Teil-)Aufgabe mit dem angegeben Namen.
Parameter:

`name` Der Name der Aufgabe. Typ: `String`

`setSubTasks` Legt fest, wie viele Teilaufgaben die aktuelle Aufgabe hat und wie sie gewichtet werden sollen. In der zweiten Variante wird nur die Anzahl der Teilaufgaben festgelegt und sie werden als gleich gewichtet angenommen (jeweils Gewicht $\frac{1}{\nearrow\texttt{count}}$).
Parameter:

`weights` Die Gewichte der Teilaufgaben, jeweils im Intervall $[0, 1]$. Typ: `float[]`

oder

`count` Die Anzahl der Teilaufgaben. Typ: `int`

`popTask` Beendet die Aufgabe mit dem angegebenen Namen und ihre Teilaufgaben, wenn ↗`name` gegeben ist, sonst die aktuelle Aufgabe.
Die Variante mit ↗`name` erlaubt es, sich von Fehlern in Teilaufgaben zu erholen: Wenn eine Aufgabe Exceptions in ihren Teilaufgaben abfängt, kann sie durch Angabe von ↗`name` auch die Teilaufgaben beenden, die wegen der Exception nicht von den Teilaufgaben selbst beendet werden konnten.
Parameter:

`name` *optional*: Der Name der Aufgabe, die entfernt werden soll. Typ: `String`

`nextTask` Abkürzung für ↗`popTask(); ↗pushTask(name);`.
Parameter:

`name` Der Name der neuen Aufgabe. Typ: `String`

`openTask` Beginnt eine neue (Teil-)Aufgabe mit dem angegebenen Namen und gibt eine ↗`CloseableTask` zurück, die in einem try-with-resources Statement verwendet werden kann, um die Aufgabe auf jeden Fall abzuschließen, selbst wenn eine Teilaufgabe eine Exception wirft.
Parameter:

`name` Der Name der Aufgabe. Typ: `String`

Rückgabetyp: ↗`CloseableTask`

`addProgressListener` Registriert einen ↗`ProgressListener`, der bei Fortschritten benachrichtigt wird.
Parameter:

`listener` Der ↗`ProgressListener`, der bei Fortschritten benachrichtigt werden soll. Typ: ↗`ProgressListener`

`setProgress` Setzt den Fortschritt der aktuellen Aufgabe direkt. Darf nicht für Aufgaben verwendet werden, die Teilaufgaben haben (siehe ↗`setSubTasks`).
Parameter:

`progress` Der aktuelle Fortschritt. Typ: `float`

`getProgress` Gibt den aktuellen Gesamtfortschritt zurück. Rückgabetyp: `float`

`getCurrentTask` Gibt den Namen der aktuellen Aufgabe zurück. Rückgabetyp: `String`

`CloseableTask` Hilfsklasse für ↗`ProgressReporter.openTask`. Implementiert `AutoCloseable`, kann also in try-with-resources Statements verwendet werden.

Methoden

`close` Beendet die Aufgabe, die mit dem zugehörigen ↗`ProgressReporter.openTask` begonnen wurde.

`ProgressListener` Neues Interface. Wird benötigt um bei Änderungen den Fortschritts in ↗`ProgressReporter` benachrichtigt zu werden.

Methoden

`startRoot` Wird aufgerufen, wenn die Basisaufgabe des ↗`ProgressReporter`s begonnen wird (erster Aufruf von ↗`ProgressReporter.pushTask`).
Parameter:

`name` Der Name der Basisaufgabe. Typ: `String`

`beginTask` Wird aufgerufen, wenn eine neue (Teil-)Aufgabe begonnen wird (Aufruf von ↗`ProgressReporter.pushTask` oder ↗`ProgressReporter.nextTask`). Für die Basisaufgabe wird diese Methode *nach* ↗`startRoot` aufgerufen.
Parameter:

`name` Der Name der (Teil-)Aufgabe. Typ: `String`

`progress` Wird aufgerufen, wenn sich der Fortschritt ändert. Bei Beenden einer Aufgabe wird diese Methode *nach* ↗`endTask` aufgerufen.
Parameter:

`progress` Der aktuelle Gesamtfortschritt. Typ: `float`

`name` Der Name der aktuellen (Teil)-Aufgabe. Typ: `String`

`endTask` Wird aufgerufen, wenn eine (Teil-)Aufgabe beendet wird (Aufruf von ↗`ProgressReporter.popTask` oder ↗`ProgressReporter.nextTask`). Für die Basisaufgabe wird diese Methode *vor* ↗`finishRoot` aufgerufen.
Parameter:

`name` Der Name der beendeten (Teil)-Aufgabe. Typ: `String`

`finishRoot` Wird aufgerufen, wenn die Basisaufgabe des ↗`ProgressReporter`s beendet wird.
Parameter:

`name` Der Name der beendeten Basisaufgabe. Typ: `String`

`CurrentCombinationListener` Neues Interface. Dient der Benachrichtigung bei Änderung der aktuellen ↗`ProfileMapCombination` (gehalten vom ↗`ProfileMapManager`).

Die ↗`MainView` registriert einen anonymen ↗`CurrentCombinationListener`, um die ↗`MapView` zu de- oder aktivieren und gegebenenfalls eine Nachricht anzuzeigen, dass eine Vorberechnung durchgeführt werden muss.

Methoden

`currentCombinationChanged` Wird aufgerufen, wenn sich die aktuelle ↗`ProfileMapCombination` ändert, etwa durch Aufruf von ↗`ProfileMapManager.setCurrentCombination`.
Parameter:

`newCombination` Die neue aktuelle ProfileMapCombination. Typ: ↗`ProfileMapCombination`

`RouteModel` Das `RouteModel` implementiert nun ↗`CurrentCombinationListener`, um die Route zu löschen, wenn die aktuelle ↗`ProfileMapCombination` sich ändert.

Methoden

`currentCombinationChanged` Setzt Start- und Zielpunkt sowie die Route und ihre Beschreibung auf `null` und benachrichtigt dann registrierte `RouteModelListener`. Siehe auch ↗`CurrentCombinationListener.currentCombinationChanged`.
Parameter:

`newCombination` Wird ignoriert. Typ: ↗`ProfileMapCombination`

2.7 Paket Views

Alle Views außer der ↗`MapView` haben im Konstruktor einen neuen Parameter `Window parent`, der ein `Frame`, aus dem dieser Dialog geöffnet, wird darstellt.

`MainView`

Methoden

`currentCombinationChanged` Setzt in alle dafür vorgesehenen Felder den aktuellen Profil- und Kartennamen. Blockiert die ↗`MapView`, falls es keine Vorberechnung für die aktuelle Kombination gibt und gibt durch `textMessage` die Meldung für den Benutzer aus.
Parameter:

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

`textMessage` Gibt Fehlermeldungen für den Benutzer auf den Bildschirm aus.
Parameter:

`str` Der auf den Bildschirm auszugebende Text. Typ: `String`

`MapManagerView`

Methoden

`MapManagerView` Konstruktor, der eine neue ↗`MapManagerView` erzeugt.
Parameter:

`mmc` Ein ↗`MapManagerController` der für die ↗`MapManagerView` verantwortlich ist solange sie offen ist. Typ: ↗`MapManagerController`

`parent` Ein `Frame`, aus dem dieser Dialog geöffnet wird. Typ: `Window`

`currentMap` Die aktuell ausgewählte Karte. Typ: ↗`StreetMap`

`maps` Alle Karten, die importiert wurden. Der Benutzer kann in der Kartenverwaltung aus diesen Karten wählen. Typ: `Set<StreetMap>`

`currentMapProfiles` Alle Profile, für die eine Vorberechnung für die aktuell ausgewählte Karte existiert. Typ: `Set<Profile>`

`ProfileManagerView`

Methoden

`ProfileManagerView` Konstruktor, der eine neue ↗`ProfileManagerView` erzeugt.
Parameter:

`parent` Ein `Frame`, aus dem dieser Dialog geöffnet wird. Typ: `Window`

`pmc` Ein ↗`ProfileManagerController`, der für die ↗`ProfileManagerView` verantwortlich ist, solange sie offen ist. Typ: ↗`ProfileManagerController`

`currentProfile` Das aktuell ausgewählte Profil. Typ: ↗`Profile`

`availableProfiles` Alle erstellten Profile. Typ: `Set<Profile>`

`MapView`

Methoden

`MapView`
Parameter:

`source` Eine Renderer, der die Kacheln für die Darstellung der Karte liefert. Typ: `TileSource`

`ProgressDialog` Neue Klasse. Ein ↗`ProgressListener`, der den aktuellen Fortschritt und die aktuellen Aufgaben in einem Dialog anzeigt. Bei ↗`ProgressListener.finishRoot` wird der Dialog unsichtbar gemacht (`setVisible(false)`).

Methoden

Keine außer geerbten und privaten Methoden.

2.8 Paket Profiles

`Profile`

Methoden

`clone` Erstellt eine Kopie eines Profils, welche nicht als Standardprofil gekennzeichnet ist. Rückgabetyp: ↗`Profile`

2.9 Paket Map

`StreetMap`

Methoden

`save` Speichert die Karte in den angegebenen Ordner.
Parameter:
`file` Die Datei, in die geschrieben werden soll. Typ: `File`

`load` (statisch) Lädt eine Karte aus dem angegebenen Ordner.
Parameter:
`file` Die Datei, aus der gelesen werden soll. Typ: `File`
Rückgabetyp: ↗`StreetMap`

`loadLazily` (statisch) Gibt eine ↗`StreetMap` zurück, die erst beim ersten Zugriff auf `getGraph` bzw. `getEdgeBasedGraph` diese auch aus dem angegebenen Ordner lädt. Diese Methode wird verwendet, um die Startzeit des Programms zu verbessern: Die Information, welche Karten vorliegen, soll zwar jederzeit verfügbar sein, allerdings werden nicht alle Karten sofort benötigt.
Parameter:
`file` Die Datei, aus der gelesen werden soll. Typ: `File`
Rückgabetyp: ↗`StreetMap`

`ensureLoaded` Nach Aufruf dieser Methode ist garantiert, dass auch eine Karte, welche mittels ↗`loadLazily` geladen wurde, vollständig geladen ist. Spätere Aufrufe von `getGraph` bzw. `getEdgeBasedGraph` sind dann garantiert schnell.
Parameter:
`reporter` Der ↗`ProgressReporter`, dem der Ladefortschritt gemeldet werden soll. Typ: ↗`ProgressReporter`

`Graph`

Methoden

`getCorrespondingEdge` Gibt die Kante in entgegengesetzter Richtung, oder -1, wenn sie nicht vorhanden ist (Erkennen von Einbahnstraßen).
Parameter:

`edge` Eine ID der Kante. Typ: `int`

Rückgabetyp: `int`

`getNumberOfEdges` Gibt die Anzahl der Kanten zurück. Dies ermöglicht es, über alle Kanten zu iterieren. Rückgabetyp: `int`

`save` Speichert den Graphen in die angegebene Datei.
Parameter:

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

`load` (statisch) Lädt einen Graphen aus der angegebenen Datei.
Parameter:

`file` Die Datei, aus der gelesen werden soll. Typ: `File`

Rückgabetyp: ↗`Graph`

`IntArraySet` Neue Klasse. Ist ein `java.util.Set<Integer>` das die Integers aus einem Interval eines int-arrays darstellt.

Methoden

`IntArraySet` Erzeugt ein neues Set. Die Daten werden nicht kopiert sondern direkt wiedergegeben.
Parameter:

`base` Der Index des ersten Elements Typ: `int`

`count` Die Anzahl der Elemente Typ: `int`

`data` Die zugrundeliegenden Daten Typ: `int[]`

`GraphIndex`

Methoden

`GraphIndex` Der Graph entscheidet, welche Indices gebaut werden sollen und bei welchen Zoomlevel welche Straßen verschwinden, sodass derselbe Index mehrfach benutzen kann.
Parameter:

`maxType` Der Index des letzten StreetTypes, der noch angezeigt werden soll. Typ: `int`

`EdgeBasedGraph`

Methoden

`getNumberOfTurns` Gibt die Anzahl der Abbiegemöglichkeiten zurück. Dies ermöglicht es, über alle Abbiegemöglichkeiten zu iterieren. Rückgabetyp: `int`

`save` Speichert den kantenbasierten Graphen in die angegebene Datei.
Parameter:

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

`load` (statisch) Lädt einen kantenbasierten Graphen aus der angegebenen Datei.
Parameter:

`file` Die Datei, aus der gelesen werden soll. Typ: `File`

Rückgabetyp: ↗`EdgeBasedGraph`

`NodeProperties`

Methoden

`NodeProperties` Konstruktor: Erzeugt ein neues Objekt mit den gegebenen Eigenschaften.
Parameter:

`junctionRef` Die Nummer der Anschlussstelle. Typ: `String`

`junctionName` Der Name der Anschlussstelle. Typ: `String`

`isMotorwayJunction` Gibt an, ob der Knoten eine Schnellstraßen- oder Autobahnanschlussstelle ist. Typ: `boolean`

`isTrafficLights` Gibt an, ob es sich bei dem Knoten um eine Ampelkreuzung handelt. Typ: `boolean`

`save` Speichert die ↗`NodeProperties` in die angegebene Datei.
Parameter:

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

`load` (statisch) Lädt ↗`NodeProperties` aus der angegebenen Datei.
Parameter:

`file` Die Datei, aus der gelesen werden soll. Typ: `File`

Rückgabetyp: ↗`NodeProperties`

`EdgeProperties`

Methoden

`save` Speichert die ↗`EdgeProperties` in die angegebene Datei.
Parameter:

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

`load` (statisch) Lädt ↗`EdgeProperties` aus der angegebenen Datei.
Parameter:

`file` Die Datei, aus der gelesen werden soll. Typ: `File`

Rückgabetyp: ↗`EdgeProperties`

`Restriction`

Methoden

`save` Speichert die ↗`Restriction`s in die angegebene Datei.
Dazu wird zunächst der Typ der Restriction geschrieben und dann die Methode `saveInternal` aufgerufen, welche die einzelnen Unterklassen überschreiben.
Parameter:

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

`load` (statisch) Lädt ↗`Restriction`s aus der angegebenen Datei.
Dazu wird zunächst der Typ der Restriction gelesen und dann die Methode `loadInternal` aufgerufen, welche die einzelnen Unterklassen definieren.
Parameter:

`file` Die Datei, aus der gelesen werden soll. Typ: `File`

Rückgabetyp: ↗`Restriction`

2.10 Paket History

`HistoryEntry`

Methoden

`fromString` Gibt einen aus dem String generierten ↗`HistoryEntry` zurück.
Parameter:
`s` Ein Text, der zu einem ↗`HistoryEntry` erhalten kann. Typ: `String`
Rückgabetyp: ↗`HistoryEntry`

`toString` Gibt einen aus dem ↗`HistoryEntry` generierten Text in Form eines Strings zurück. Rückgabetyp: `String`

2.11 Paket Util

`Coordinates`

Methoden

`goIntoDirection` Berechnet die Koordinaten des Punkts mit der angegebenen Position auf einer Linie. Dieses Verfahren kann verwendet werden, um die Koordinaten eines `PointOnEdge` zu bestimmen.
Parameter:
`to` Gibt die Richtung an, in die gegangen wird Typ: ↗`Coordinates`
`position` Ein Wert zwischen 0 und 1, der angibt wie weit in diese Richtung gegangen wird. Typ: `float`
Rückgabetyp: ↗`Coordinates`

`toString` Gibt eine String-Repräsentation der Koordinaten zurück, die von ↗`fromString` wieder geparst werden kann. Rückgabetyp: `String`

`fromString` (statisch) Parst einen Koordinaten-String, wie er von ↗`toString` zurückgegeben wird.
Parameter:
`coords` Der Koordinaten-String. Typ: `String`
Rückgabetyp: ↗`Coordinates`

`parseLatitude` (statisch) Parst eine Breitengrad-Angabe aus einer Zeichenkette.
Parameter:
`lat` Die Breitengrad-Angabe. Typ: `String`
Rückgabetyp: `float`

`parseLongitude` (statisch) Parst eine Längengrad-Angabe aus einer Zeichenkette.
Parameter:
`lon` Die Längengrad-Angabe. Typ: `String`
Rückgabetyp: `float`

`FileUtil` Neue Klasse. Erhält verschiedene statische Methoden, die Dateien und Pfade behandeln.

Methoden

`rmRf` (statisch) Löscht ein Verzeichnis rekursiv. Wird verwendet von `MapManager.deleteMap` und ↗`ProfileMapManager.deletePrecalculation`.
Parameter:
`directory` Das Verzeichnis, das gelöscht werden soll. Typ: `File`

`getRootDir` (statisch) Gibt das Ursprungsverzeichnis der *routeKIT*-Installation zurück. Dieses Verzeichnis enthält alle Karten, Profile und Vorberechnungen.

Windows `%APPDATA%\routeKIT`

Mac `$HOME/Library/Application Support/routeKIT`

Unix/Linux `$HOME/.config/routeKIT`

Rückgabetyp: `File`

`getHistoryFile` Gibt die Datei zurück, in der der Verlauf (siehe `History`) gespeichert wird: `routeKIT.history` in ↗`getRootDir`. Rückgabetyp: `File`

`TimeUtil` Neue Klasse. Enthält Methoden zur Zeitumrechnung.

Methoden

`timeSpanString` Wandelt eine Zeitspanne in Text um und hängt sie an ↗`text` an. Wird verwendet von ↗`ProfileManagerView` und ↗`MapManagerView`.
Parameter:

`text` Der `StringBuilder`, an den die Zeitspanne angehängt werden soll. Typ: `StringBuilder`

`interval` Die Zeitspanne in Millisekunden. Typ: `int`

`Dummies` Neue Klasse. Enthält drei Methoden, die eine *routeKIT*-Installation erstellen, welche mehr oder weniger nützlich ist.

Methoden

`createDummies` Erstellt eine Dummy-Karte (fast leer) mit einer Dummy-Vorberechnung (Arc-Flags alle gesetzt). Die älteste der drei Methoden, und jetzt nur noch nützlich, wenn man schnell irgendeine Vorberechnung braucht.
Parameter:

`rootDir` Das Wurzelverzeichnis der *routeKIT*-Installation. Typ: `File`

`createInstall` Lädt eine Karte des Regierungsbezirks Karlsruhe von `geofabrik.de` herunter und führt mit dieser Karte eine Vorberechnung für das Profil „PKW (Standard)“ durch. Dies erstellt eine vollständige und vollwertige Installation, kann allerdings eine Weile dauern.
Parameter:

`rootDir` Das Wurzelverzeichnis der *routeKIT*-Installation. Typ: `File`

`downloadInstall` Lädt eine komplette, gezippte Installation von `dogcraft.de` herunter und extrahiert sie in ↗`rootDir`. Die Installation enthält eine Karte des Regierungsbezirks Karlsruhe sowie eine Karte von Baden-Württemberg, jeweils mit einer Vorberechnung für das Profil „PKW (Standard)“.
Parameter:

`rootDir` Das Wurzelverzeichnis der *routeKIT*-Installation. Typ: `File`

`main` Startet ↗`downloadInstall`.
Parameter:

`args` Wird ignoriert. Typ: `String[]`

3 Komponententests

Die automatisierten Komponententests wurden zum größten Teil noch nicht während der Implementierung erstellt und daher erst im Qualitätssicherungsbericht beschrieben.

4 Programmfehler

Im folgendem Abschnitt werden die gefundenen und nicht behobenen Fehler im Programm aufgelistet, sowie gefundene Lösungsansätze für deren Behebung.

1. **Symptom** Für große Karten ist das Rendern der Kacheln zu langsam.

   Ursache Es existieren zu viele Kanten auf einer Kachel.

   Lösungsidee Vergröberunsalgorithmus realisieren.

2. **Symptom** Die Anwendung braucht lange zum Starten.

   Ursache `GraphIndex`e und Navigationsarrays im Graphen werden beim Starten neu aufgebaut.

   Lösungsidee Mehr Daten auf die Festplatte schreiben und beim Start laden.

3. **Symptom** Nach dem Wechseln der Karte sind möglicherweise keine Kanten sichtbar.

   Ursache Die neuen Kanten sind in einer anderen geografischen Region.

   Lösungsidee Kartenmittelpunkt bestimmen und die Karte dort zentrieren.

4. **Symptom** Die Nachricht „Sie haben ihr Ziel erreicht.“ wird bei jeder Wegbeschreibung ausgegeben.

   Ursache Im Programmcode ist keine Bedingung vorhanden, die Nachricht nur bei existierender Beschreibung auszugeben.

   Lösungsidee Code verfeinern.

5. **Symptom** Eine Nachricht „Es wurde keine Route gefunden.“ wird niemals ausgeben.

   Ursache Diese Nachricht ist noch nicht im Code vorgesehen.

   Lösungsidee Spezialfall implementieren.

6. **Symptom** Es ist möglich, so weit herauszuzoomen, dass die Karte nicht mehr sichtbar ist.

   Ursache Die Zoomstufe wird nur statisch auf $[0, 19]$ eingeschränkt.

   Lösungsidee Die Zoomstufe abhängig von der Größe der Karte einschränken.

5 Arbeitsplanung

Auf den folgenden Seiten sind der ursprüngliche Implementierungsplan (↗Abbildung 1) sowie der tatsächliche Zeitplan der Implementierung (↗Abbildung 2) als Gantt-Diagramme dargestellt.

Die Anzeige der Karte, für welche die Klasse ↗`MapView` verantwortlich war, wurde vor den `TileRenderer` platziert, da dies als wichtiger für andere Teammitglieder eingestuft wurde. Im Großen und Ganzen erwies sich der abgeschätzte Arbeitsaufwand als zutreffend; allerdings ergaben sich einige Abweichungen im tatsächlichen Arbeitsablauf dadurch, dass nicht immer so viel Zeit wie geplant zur Verfügung stand. Dafür wurde jedoch in der ursprünglichen Planung ein einwöchiger Zeitpuffer vorgesehen, der auch zur Fehlerbehebung genutzt wurde.

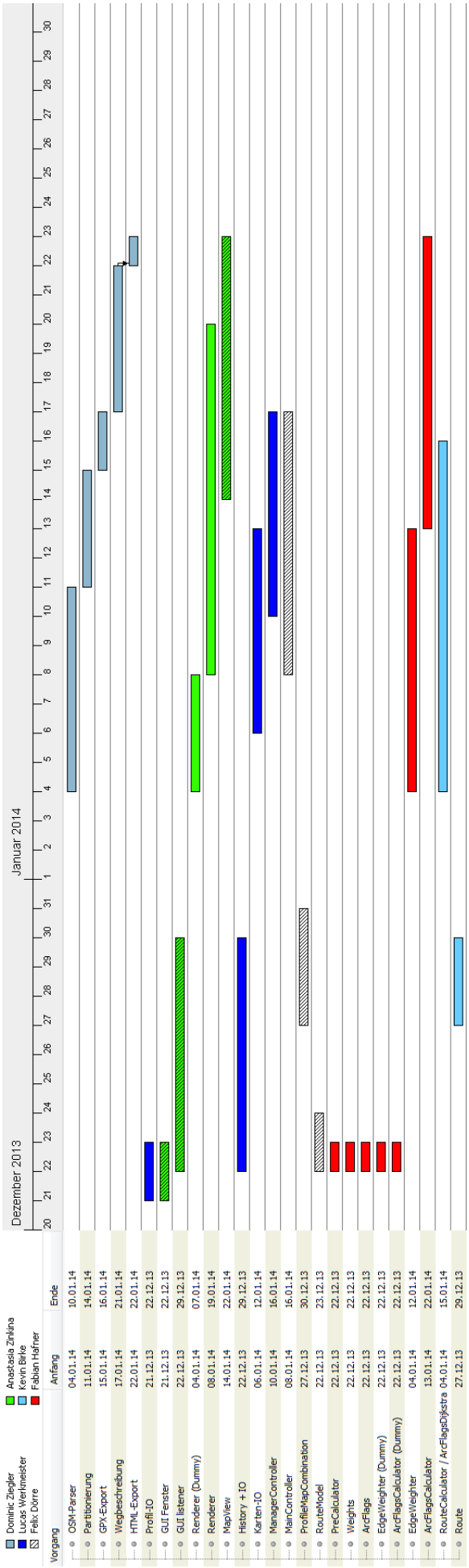

Abbildung 1: Ursprünglicher Zeitplan der Implementierungsphase

Abbildung 2: Tatsächlicher Arbeitsablauf der Implementierungsphase