Validierungsbericht

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 **4**

2 Automatische Komponententests **5**
2.1 Test des Rendering-Stils 5
2.2 Test des Imports und des Exports 5
2.3 Test der Hilfsklasse Geometry 6
2.4 Test der Klasse Dijkstra 6
2.5 Test der Arc-Flags 6
2.6 Test des OSM-Parsers 7
2.7 Test des SRTM-Parsers 7

3 Manuelle Komponententests **9**
3.1 Die Benutzung der Karte 9
3.2 Wegpunktverwaltung, Routenberechnung und -Darstellung, Informationen und Höhenverlauf zur Route 10
3.3 Das Laden und Speichern einer geplanten Route 11
3.4 Das Anzeigen und Drucken der Routenbeschreibung 11
3.5 Bewegen des Kartenausschnitts zu einer Stadt 11
3.6 Testen der übrigen Funktionalität 12

4 Skalierungs- und Belastungstests **13**
4.1 Dauer des Programmstarts 13
4.2 Dauer der Routenberechnung 14
4.3 Dauer der Arc-Flagsberechnung 14
4.4 Arbeitsspeicherverbrauch im Anwendungsprogramm 15
4.5 Arbeitsspeicherverbrauch im Vorberechner 15
4.6 Größe der Binärdaten 15

5 Qualitätsanforderungen **17**

6 Aufgedeckte Softwarefehler **18**
6.1 Zoomstufen und Position werden nicht mit abgespeichert und geladen . . 18
6.2 Codierungsfehler auf Windows 18
6.3 Formatierungsprobleme auf Windows 18
6.4 Darstellungsfehler im Koordinatensystem der Höhenkarte 19
6.5 Arc-Flags wurden nicht korrekt verwendet 19
6.6 Arc-Flagsberechnung endete bei einer Karte nicht 19

6.7 Teilweise abgeschnittene Städtenamen 19
6.8 Fehler beim Wegpunkt verschieben 19
6.9 Unvollständige Darstellung des Rheins 20

7 Weitere Programmänderungen **21**
7.1 Sofortiges Berechnen der Route 21
7.2 Ortsuchfunktion beachtet Groß- und Kleinschreibung 21

Glossar **22**

1 Einleitung

In der Validierungsphase wurden zur Qualitätssicherung des Programms zahlreiche Tests druchgeführt. Diese Tests sollten so weit als möglich automatisiert durchgeführt werden, einige Programmteile mussten jedoch auch manuell getestet werden. Die hieraus gewonnenen Erkentnisse wurden zum Einen zur Beseitigung etwaiger Fehler und zum Anderen zur Verifizierung der Korrektheit von möglichst großen Teilen des Programmcodes genutzt. Außerdem wurden Geschwindigkeitsanalysen durchgeführt und damit die Skalierfähigkeit des Programms bewertet.

Im Folgenden kann die Beschreibung der Tests, sowie der gewonnenen Erkenntnisse nachgelesen werden.

2 Automatische Komponententests

Schon während der Implementierungsphase wurden automatisierte Komponententests◀ entwickelt und durchgeführt. Ziel war, relevante Programmteile zu verifizieren und dabei eine hohe Codeüberdeckung◀ zu erreichen. Während der Validierungsphase kamen nun weitere Komponententests◀ hinzu.

2.1 Test des Rendering-Stils

Beteiligte Klassen: RenderingStyle, AbstractRenderingStyle
Erreichte Codeüberdeckung: 76,2%

Das Modul RenderingStyle, welches mit einfachen Strings◀ Renderingparameter für Elemente der Klassen *Way*, *Area* und *City* setzt, verwaltet und für sie eine Schnittstelle bereitstellt, wurde mithilfe einer Unterklasse von AbstractRenderingStyle getestet. In diesen Tests wurde vorrangig getestet, ob alle definierten String◀ erkannt, und korrekt interpretiert wurden.

Insgesamt wurden alle Methoden von den Tests überdeckt. Aufgrund des Umfangs der Klasse, und der vielen Variationen und Fehlerüberprüfungen haben wir uns deshalb mit einer Codeüberdeckung◀ von 76,2% zufrieden gegeben.

2.2 Test des Imports und des Exports

Beteiligte Klassen: Importer, Exporter, alle Klassen des MapModel-Pakets
Erreichte Codeüberdeckung Importer: 93,3%
Erreichte Codeüberdeckung Exporter: 78,4%
Erreichte Codeüberdeckung MapModel-Paket: 45,7%

Zum Test der Klassen Import und Export wird zunächst künstlich ein MapModel erzeugt. Dieses wird exportiert und anschließend wieder aus den Binärdateien geladen. Danach wird überprüft ob die importierte Instanz mit der erzeugten identisch ist.

Das erzeugte MapModel besteht aus 100 Knoten, die als doppelter Ring verbunden sind. Diese Knoten gehören außerdem zu einem Way und begrenzen eine Area. Die Elemente werden mit verschiedenen Eigenschaften versehen. Desweiteren sind Orte verschiedener Größenordnungen enthalten.

2.3 Test der Hilfsklasse Geometry

Beteiligte Klassen: Geometry
Erreichte Codeüberdeckung: 100%

Die Klasse Geometry, welche unabhängig von anderen Klassen geometrische Berechnungen statisch durchführt, wurde mit einfachen Beispieldaten automatisch getestet. Mit 16 Tests für die Methoden

- *isLineIntersectingLine(...)*,
- *isPointInsideRectangle(...)*,
- *isLineIntersectingRectangle(...)*

konnte eine Codeüberdeckung◀ von 100% erreicht werden.

2.4 Test der Klasse Dijkstra

Beteiligte Klassen: Dijkstra, Node, Edge
Erreichte Codeüberdeckung: 97,1%

Die Klasse Dijkstra◀ wurde mithilfe eines, von Hand erstellten Graphen aus acht Knoten und zehn Kanten auf ihre Korrektheit getestet. Es wurde primär getestet ob der kürzeste Pfad gefunden wird. Dabei wurde eine Codeüberdeckung◀ von 97,1% erreicht. Der Testgraph enthält Arc-Flags◀ sowie unpassierbare Ways. Außerdem wurde eine Routenberechnung auf einem Graphen mit einem Knoten durchgeführt und getestet, ob die gewünschte Rückgabe erfolgt.

2.5 Test der Arc-Flags

Erreichte Codeüberdeckung Dijkstra: 97,6%

Die korrekte Berechnung und Verwendung der Arc-Flags◀ wurde auf folgende Weise getestet: Zunächst werden die vorberechneten Daten des MapModel einschließlich der Arc-Flags◀ wie gewohnt geladen. Hieraus werden zufällig zwei Knoten ausgesucht, von denen jeder mit mindestens einer befahrbaren Kante verbunden ist.

Nun wird mittels der Klasse Dijkstra eine geeignete Route zwischen den beiden Knoten unter Verwendung der Arc-Flags◀ berechnet. Das Ganze wird 500 mal wiederholt und dabei jeweils die verwendeten Start- und Zielpunkte, sowie die errechnete Streckenlänge gesichert.

Anschließend werden die Arc-Flags◀ aller Kanten im Graphen zurückgesetzt und zwischen jedem der 500 Start- und Zielpunkte erneut eine geeignete Route berechnet. Der Test besteht, wenn die Längen der berechneten Routen sich jeweils nicht unterscheiden.

2.6 Test des OSM-Parsers

Beteiligte Klassen: OSMHandler, OSMParser, ParserController
Erreichte Codeüberdeckungen◀ OSMParser: 71,9%
Erreichte Codeüberdeckungen◀ OSMHandler: 49,9%
Erreichte Codeüberdeckungen◀ ParserController: 68,8%

Das korrekte Parsen der OSM◀-Datei wurde mit einer Testgruppe aus 288 Knoten, 87 Wegen, 20 Gebieten und 6 Städten überprüft. Das Ergebnis des Parsers wurde mit den erwartetem Ergebnis verglichen und stimmt vollständig überein. Nur die Liste der Knoten ist durch das Chaos Prinzip des HashSets nicht direkt vergleichbar. Hierfür wurden die Anzahl der ermittelten Knoten mit dem Optimalwert verglichen. Anschließend wird geprüft ob alle ermittelten auch erwünscht sind, indem geprüft wird ob die Knoten in dem HashSet liegen.

Die geringe Codeüberdeckungen◀des OSMHandlers erklärt sich mit der fehlenden Ordnung in den OSM◀-Dateien. Jede Tag-Permutation wird überprüft und behandelt. Mit der angegebenen Testgruppe wurde aber nur ein Bruchteil der möglichen Permutationen abgedeckt.

2.7 Test des SRTM-Parsers

Beteiligte Klassen: SRTMParser, Altitude◀Data, ParserController
Erreichte Codeüberdeckung SRTMParser: 93,9%
Erreichte Codeüberdeckung ParserController: 25%

Für diesen Test wurden drei Arc-Ascii Srtm-Datein erstellt. Die von dem SRTM-Parser erstellten Daten werden mit den erwarteten Daten abgeglichen. Sie stimmen vollständig überein.

getestete Komponente/Klasse	**Codeüberdeckung◀ [in %]**
Rendering-Stils	76,2
Importer	93,3
Exporter	78,4
MapModel-Paket	45,7
Geometry	100
Dijkstra	97,1
Arc-Flags	97,6
OSM-Handler	49,9
OSM-Parser	71,9
SRTM-Parser	93,9
ParserController	93,8

Tabelle 2.1: Übersicht der erreichten Codeüberdeckungen◀ bei den automatischen Komponententests◀

3 Manuelle Komponententests

In der Sektion „Testfälle und Szenarien“ des Pflichtenhefts sind viele Tests gefordert, die sich nicht oder nur schlecht automatisiert testen lassen. Diese sollen mit Hilfe von Benutzerinteraktion in dieser Sektion getestet werden. Als Grundlage jedes Tests dient ein Programmstart des Anwendungsprogramms mit einem sehr großen OSM◀-Datensatz für die Umgebung Straßburg-Karlsruhe-Heidelberg-Stuttgart.

Die Codeabdeckung nach Durchführung sämtlicher Tests nacheinander betrug 93.1%, wenn man nur die Klassen betrachtet, die in den einzelnen Testfällen aufgelistet sind. Betrachtet man hingegen alle Klassen des Anwendungsprogramms ergibt sich eine durchschnittliche Codeabdeckung von 90,9%. Die Werte für jedes einzelne Paket lassen sich aus Tabelle 4.4 ablesen.

Paket	**Codeüberdeckung◀ [in %]**
Controllers	86,3
GUIControllers	98,3
Main	93,8
MapRendering	86,9
RunTimeData	90,6
UserInterface	97,2

Tabelle 3.1: Übersicht der erreichten Codeüberdeckungen◀ mit allen manuellen Testfällen

3.1 Die Benutzung der Karte

Abgedeckte Testfälle: /T10/ Die Darstellung der Karte, /T11/ Das Verschieben der Karte, /T12/ Das Vergößern/Verkleinern der Karte
Beteiligte Klassen: MapController, GUIController, MainGUIController, MapTile, MapTileCache, MapTileRenderer, MainGUI, MapView
Aufgedeckte Fehler: Teilweise abgeschnittene Städtenamen, Unvollständige Darstellung des Rheins

Um die Darstelleung der Karte zu testen, wurde einerseits die Position, andererseits die

Zoomstufe◀ der Karte sowohl unter Benutzung der dafür vorgesehenen Oberflächenelemente als auch durch Verwendung der Tastatur und Maus verändert.

Die Darstelleungskriterien, wie sie in „/F10/ Die Karte darstellen“ im Pflichtenheft festgelegt sind, wurden eingehalten.

Anfänglich waren bei manchen Zoomstufen◀ Städtenamen vereinzelt abgeschnitten (siehe Fehler 6.7). Dies konnte behoben werden. Die Darstellung des Rheins konnte leider nicht in ausreichender Breite erfolgen, widerspricht jedoch nicht den Forderungen des Pflichtenhefts (siehe Fehler 6.9).

3.2 Wegpunktverwaltung, Routenberechnung und -Darstellung, Informationen und Höhenverlauf zur Route

Abgedeckte Testfälle: /T20/ Das Festlegen eines Wegpunktes, /T21/ Das Verschieben eines Wegpunktes, /T22/ Das Löschen eines Wegpunktes, /T30/ Die Berechnung einer geeigneten Route, /T40/ Das Darstellen einer berechnete Route auf der Karte, /*T60/ Das Anzeigen der Informationen zur berechneten Route, /*T70/ Das Anzeigen des Höhenverlaufs der berechneten Route
Beteiligte Klassen: AltitudeData, AltitudeMapController, AltitudeMapView, AltitudeMapViewController, CalculationController, CalculatedRoute, Dijkstra, MainGUI, MainGUIController, MapController, MapView, MapViewController, PlanningController, PlannedWaypoints, RouteInformationController, RouteTile, RouteTileCache, RouteTileRenderer, TunnelStroke
Aufgedeckte Fehler: Darstellungsfehler im Koordinatensystem der Höhenkarte, ArcFlags◀berechnung endete bei einer Karte nicht

In diesem Testfall wurden mehrere Wegpunkte gesetzt, teilweise wieder gelöscht sowie verschoben. Da bei mindestens zwei gleichzeitig gesetzten Wegpunkten die Route direkt berechnet und eingezeichnet wird, wurde dadurch auch gleichzeitig das Berechnen und Darstellen einer geeigneten Route getestet.
Das Programm verhielt sich gemäß den Vorgaben im Pflichtenheft. Das heißt insbesondere, dass die berechneten Routen nicht auf durch Fahrräder unbefahrbaren Straßen verliefen.
Es wurden auch Wegpunkte absichtlich so gesetzt, dass keine gültige Route zwischen den Punkten existiert. Nachdem das Programm in diesem Fall zunächst in einer Endlosschleife verharrte, gibt es nach einem Bugfix◀ nun stattdessen eine Fehlermeldung aus (siehe Fehler 6.6).
Durch das Berechnen der Route wurden wie vorgesehen im unteren Bereich der Benutzeroberfläche plausible Informationen zur Route gemäß den Vorgaben im Pflichenheft angezeigt. Auch das Höhenverlaufsdiagramm im oberen Bereich der Benutzeroberfläche wurde dargestellt und erfolgreich auf die Vorgaben des Pflichtenhefts gesprüft. Die Färbung des Höhenverlaufsdiagramms war ebenfalls korrekt.

3.3 Das Laden und Speichern einer geplanten Route

Abgedeckte Testfälle: /*T50/ Das Speichern einer geplanten Route, /*T55/ Das Laden einer geplanten Route
Beteiligte Klassen: PlanningController, PlannedWaypoints
Aufgedeckte Fehler: Zoomstufe◀ und Position werden nicht mit abgespeichert und geladen

Um das Speichern und Laden einer Route zu testen, wurden viele stark verteilte Wegpunkte gesetzt. Diese wurden mit der Funktion „Route speichern“ des Programms gespeichert. Dann wurden die aktuellen Wegpunkte verworfen und der Kartenausschnitt in einer unterschiedlichen Zoomstufe◀ zu einer anderen Position bewegt. Abschließend wurden die Wegpunkte und die Zoomstufe sowie Position der Karte mittels der Funktion „Route laden“ erfolgreich wiederhergestellt. Das Sichern und Wiederherstellen des aktuellen Zoomlevels und der Kartenposition musste nachträglich noch hinzugefügt werden (siehe Fehler6.7).

3.4 Das Anzeigen und Drucken der Routenbeschreibung

Abgedeckte Testfälle: /*T80/ Das Anzeigen der Beschreibung der berechneten Route, /*T85/ Das Drucken der Beschreibung der berechneten Route
Beteiligte Klassen: DescriptionGUI, DescriptionGUIController, InstructionsController
Aufgedeckte Fehler: Codierungsfehler auf Windows

Nachdem eine nichttriviale Route geplant und berechnet wurde, wurde in diesem Test mithilfe des Buttons „Routenbeschreibung“ erfolgreich ein neues Fenster mit einer ausführlichen textuellen Beschreibung der Route samt dazugehörigen Informationen, Höhenverlaufsdiagramm sowie einem Kartenausschnitt, auf dem die eingezeichnete Route vollständig zu sehen ist, aufgerufen. Die Funktion deckt somit alle Anforderungen aus dem Pflichtenheft ab.

Auch die Vorbedingung, dass nur Informationen zu einer Route angezeigt werden können, wenn diese bereits berechnet wurde, wird eingehalten, indem der Button „Routenbeschreibung“ ausgegraut wird und somit nicht anklickbar ist, solange keine Route berechnet wurde.

Anschließend konnte der Inhalt des Fensters mit der Routenbeschreibung erfolgreich über den Button ” ´Drucken“ gedruckt werden.

Unter Windows gab es zunächst Probleme bei der Darstellung von Navigationspfeilen innerhalb der Routenbeschreibung (siehe Fehler 6.2). Dies konnte behoben werden.

3.5 Bewegen des Kartenausschnitts zu einer Stadt

Abgedeckte Testfälle: /*T90/ Bewegen des Kartenausschnitts zu einer großen deutschen Stadt

Beteiligte Klassen: City, CityData, MainGUI, MainGUIController, MapController, MapModel
Aufgedeckte Fehler: Ortsuchfunktion beachtet Groß- und Kleinschreibung

In diesem Test wurde die Karte sowohl mithilfe des neuen Suchfelds auf der Programmoberfläche als auch mit dem im Menü dafür vorgesehenen Eintrag " ´zu Ort springen" erfolgreich zu verschiedenen Städten innerhalb des abgedeckten Kartenbereichs verschoben.

Auch Grenzfälle wie das Eingeben eines leeren Strings◀ oder Städtenamen mit Umlauten wurden dabei abgedeckt und funktionierten wie gewünscht. Nachdem zunächst die Großschreibung berücksüchtigt wurde, macht es nun keinen Unterschied mehr, ob einzelne Buchstaben des SuchStrings◀ groß oder klein geschrieben werden (siehe Fehler7.2).

3.6 Testen der übrigen Funktionalität

Zum Abschluss wurden noch die Funktionen des Programms getestet, die von keinem anderen Testfall abgedeckt wurden. Dies sind im Konkretn der Vollbildmodus sowie der Dialog, der erscheint, wenn der Menüeintrag „Über" gewählt wird. Beides funktionierte wie vorgesehen.

4 Skalierungs- und Belastungstests

Mit Skalierungs- und Belastungstest sollten auch die quantitativen Aspekte des Programms betrachtet werden. Die Tests sind nicht allgemeingültig, da sie jeweils nur auf einem einzelnen Rechner ausgeführt werden. Interessant ist jedoch der Vergleich der Ergebnisse im Verhältnis zur Eingabegröße. Soweit dies unter der beschränkten Anzahl an Testwerten möglich ist, kann so auch ein asymptotisches Verhalten der jeweiligen Teilbereiche geschätzt werden.

4.1 Dauer des Programmstarts

Zum Testen der Dauer des Programmstarts, maßgeblich trägt dazu nur der Import der Binärdaten bei, wurden vier verschiedengroße Kartendatensätze verwendet.

Karlsruhe klein

10.161 Knoten, 2 Orte, 1.600 Wege, 349 Gebiete, 13.432 Kanten
25.544 Elemente insgesamt
Start in *960ms*.

Karlsruhe groß

33.694 Knoten, 12 Orte, 5.372 Wege, 2.031 Gebiete, 42.904 Kanten
84.013 Elemente insgesamt.
Start in *1s 321ms*.

Berlin

1.365.123 Knoten, 110 Orte, 82.457 Wege, 160.213 Gebiete, 705.204 Kanten
2.313.107 Elemente insgesamt
Start in *6s 330ms*.

Straßburg-Karlsruhe-Heidelberg-Stuttgart

6.462.340 Knoten, 1.473 Orte, 384.047 Wege, 679.229 Gebiete, 4.791.814 Kanten
12.318.903 Elemente insgesamt
Start in *37s 242ms*

4.2 Dauer der Routenberechnung

Zur Messung der Dauer der Routenberechnung wurden auf drei verschiedengroßen Karten jeweils 500 zufällige Routenberechnungen durchgeführt. Die Routenberechnung verwendet hierbei Arc-Flags◀ mit 64 Regionen. Der durchschnittliche gemessene Speedup◀ gegenüber der Berechnung ohne Arc-Flags◀ beträgt 800%.

Karlsruhe klein

10.161 Knoten, 2 Orte, 1.600 Wege, 349 Gebiete, 13.432 Kanten
25.544 Elemente insgesamt
Durchschnittszeit *3ms*.

Karlsruhe groß

33.694 Knoten, 12 Orte, 5.372 Wege, 2.031 Gebiete, 42.904 Kanten
84.013 Elemente insgesamt.
Durchschnittszeit *7ms*.

Berlin

1.365.123 Knoten, 110 Orte, 82.457 Wege, 160.213 Gebiete, 705.204 Kanten
2.313.107 Elemente insgesamt
Durchschnittszeit *228ms*.

Straßburg-Karlsruhe-Heidelberg-Stuttgart

6.462.340 Knoten, 1.473 Orte, 384.047 Wege, 679.229 Gebiete, 4.791.814 Kanten
12.318.903 Elemente insgesamt
Durchschnittszeit *413ms*.

4.3 Dauer der Arc-Flagsberechnung

Diese Zeiten wurden im Vorberechnungsprogramm abgelesen.

Karte	**Elemente**	**Dauer der Arc-Flagsberechnung**
Karlsruhe klein	25.544	1m 30s
Karlsruhe groß	84.013	5m 10s
Saarland	1.863.102	11h 53m 19s
Berlin	2.313.107	13h 40m 29s
Straßburg-Karlsruhe-Heidelberg-Stuttgart	12.305.903	3T 21h 19m 6s

Tabelle 4.1: Übersicht über die Zeiten der Arc-Flagsberechnungen

4.4 Arbeitsspeicherverbrauch im Anwendungsprogramm

Der Arbeitsspeicherverbrauch des Anwendungsprogramms wurde direkt zum Zeitpunkt des Erscheinens des Hauptfensters gemessen.

Karte	**Elemente**	**Arbeitsspeicherverbrauch [in MB]**
Karlsruhe klein	25.544	93,8
Karlsruhe groß	84.013	138,1
Saarland	1.863.102	281,7
Berlin	2.313.107	330,6
Straßburg-Karlsruhe-Heidelberg-Stuttgart	12.305.903	942,4

Tabelle 4.2: Übersicht über den Arbeitsspeicherverbrauch

4.5 Arbeitsspeicherverbrauch im Vorberechner

Bei der Vorberechnung, inklusive Höhendaten und Arc-Flags, wurde jeweils der maximale Arbeitsspeicherverbrauch gemessen.

Karte	**Elemente**	**Arbeitsspeicherverbrauch [in MB]**
Karlsruhe klein	25.544	390
Karlsruhe groß	84.013	250
Berlin	2.313.107	610
Straßburg-Karlsruhe-Heidelberg-Stuttgart	12.305.903	2132

Tabelle 4.3: Übersicht über den Arbeitsspeicherverbrauch

4.6 Größe der Binärdaten

Um einen Überblick über die Größe der Binärdaten zu geben, welche nach der Vorberechnung entstehen, betrachten wir wieder vier verschiedene Datensätze.

Karte	**Elemente**	**Größe Binärdaten [in MB]**
Karlsruhe klein	25.544	0.582
Karlsruhe groß	84.013	1.89
Saarland	1.863.102	46
Berlin	2.313.107	57
Straßburg-Karlsruhe-Heidelberg-Stuttgart	12.305.903	200

Tabelle 4.4: Übersicht über die größe der Binärdaten

5 Qualitätsanforderungen

Im Pflichtenheft haben wir einige Qualitätsanforderungen an unser Programm gestellt. Nach den automatischen Komponenten Tests(S.8), sowie den Skalierungs- und Belastungstest(S.11), konnten wir bestimmen, welche davon erfüllt wurden. Dies ist in der folgenden Tabelle dargestellt.

Qualitätsanforderung	**Erfüllt**
/Q10/ Die Berechnung der Route darf höchstens fünf Sekunden dauern	Ja
/Q20/ Die Aktualisierung der Karte darf höchstens fünf Sekunden dauern	Ja
/Q30/ Maximaler Hauptspeicherverbrauch: 1,5 GiB	Ja
/Q50/ Die Testfälle /T50/ bis /T90/ (S.9 3.2-3.4) müssen ohne Fehler bestanden werden	Ja
/Q60/ JUnit-Tests müssen eine Codeüberdeckungen◀ von 80% oder höher erreichen	Nein
/Q70/ Das Programm muss zwei Stunden unter wiederholter Durchführung der Testfälle ohne Absturz laufen	Ja
/Q80/ Die Bedienung der Benutzerschnittstelle soll keine unerwarteten Ereignisse auslösen	Ja
/Q90/ Die berechneten Routen müssen nachvollziehbar sein	Ja

Tabelle 5.1: Übersicht der gestellten Qualitätsanforderungen an das Programm

6 Aufgedeckte Softwarefehler

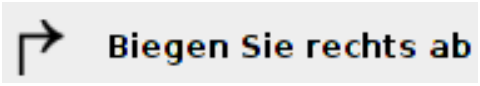
6.1 Zoomstufen und Position werden nicht mit abgespeichert und geladen

Im Pflichtenheft ist gefordert, dass eine gespeicherte Route nicht nur die aktuell gesetzten Wegpunkte, sondern auch die Zoomstufe◀ sowie die Kartenposition enthält und diese beim Laden der Route wiederhergestellt werden. Diese Funktionalität wurde vergessen und nachträglich erfolgreich hinzugefügt.

Betroffene Klassen: PlanningController, MainGUIController

6.2 Codierungsfehler auf Windows

BiKeIT verwendet bei der Routenbeschreibung kleine Pfeilpiktogramme um die Anweisungen zu veranschaulichen. Da alle benötigten Symbole im Unicode Zeichensatz vorhanden sind, wurde zunächst auf die Verwendung von eigenen Grafiken verzichtet. Unter Windows sind mit den Standardschriften jedoch einige Zeichen, wie die Start- und Zielfähnchen, nicht verügbar. Um dies zu vermeiden, werden nun doch eigene Grafiken eingebunden.



Betroffene Klasse: DescriptionGUI

6.3 Formatierungsprobleme auf Windows

Windows ignorierte an einigen Stellen die Größeneinstellung von Oberflächenelementen, die in Java◀-Swing mit *setPreferredSize()* gesetzt werden kann. Abhilfe schaffte ein anderer Layoutmanager.

Betroffene Klasse: DescriptionGUI

6.4 Darstellungsfehler im Koordinatensystem der Höhenkarte

Die Beschriftung der Y-Achse im Höhendiagramm war in bestimmten Fällen fehlerhaft, da die oberste Höhenangabe abgeschnitten war. Im Code wurde die Beachtung des Offsets zur Oberkante vergessen.

Betroffene Klasse: AltitudeMapController

6.5 Arc-Flags wurden nicht korrekt verwendet

Bei den Arc-Flags◀ wurde ein Fehler gefunden, der auf vielen Routen nicht auftritt und somit erst im Nachhinein gefunden wurde. Die Klasse Edge hat die Methode *getFlag(x)*, mit der geprüft werden kann, ob das Flag für Region x gesetzt wurde. Diese werwendete die arithmetische Rechtsshiftoperation >>, welche Vorzeichenkorrektur beinhaltet und somit zu unerwarteten Fehlern in Region 63 führte. Nun wird der logische Rechtsshift >>> verwendet.

Betroffene Klasse: Edge

6.6 Arc-Flagsberechnung endete bei einer Karte nicht

Auch bei der Berechnung der Arc-Flags◀ gab es einen Fehler, der nur auf einer einzigen Karte reproduziert werden konnte. Auf dieser Karte gab es zwei Knoten die exakt die gleiche Distanz hatten und die bei der Abarbeitung sich immer gegenseitig in die Warteschlange einordneten. Die betreffende Methode *add()* der Warteschlange wird nun nur noch dann aufgerufen, wenn sie wirklich benötigt wird.

Betroffene Klasse: PreprocessingDijkstra

6.7 Teilweise abgeschnittene Städtenamen

Bei bestimmten Städten wurde der Namen unter manchen Zoomstufen◀ nicht komplett angezeigt. Das Problem trat dann auf, wenn ein Stadtname über mehrere Kartenkacheln verlief. Es ließ sich dadurch beheben, dass eine Stadt nun auch den umgebenden Kacheln bekannt ist, sodass auf diese nötige Teile des Schriftzugs gezeichnet werden.

Betroffene Klasse: CityData

6.8 Fehler beim Wegpunkt verschieben

Wenn der Nutzer einen Wegpunkt verschob und dabei ausversehen gleichzeitig die rechte Maustaste klickte, so lies sich der halbtransparente Wegpunkt nicht weiterverschieben.

Dieses Symptom wurde behoben, indem im Eventhandler der Maustasten nun überprüft wird, ob der Nutzer sich gerade im Verschiebemodus befindet.

Betroffene Klasse: MainGUI

6.9 Unvollständige Darstellung des Rheins

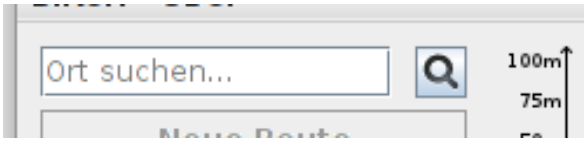
Da der OSM-Parser den OSMs◀-Typen "Relation"nicht bearbeitet, werden die daraus entstehenden Informationen nicht beachtet. Es wäre möglich die Relationen zu bearbeiten, was aber einiges an Zeit benötigen würde, da es mehrere Relationsarten gibt. Zudem würde dies weitere Klassen und eine weitere Verarbeitungsstruktur im OSM-Handler in der Größenordnung der Way-Verarbeitung nach sich ziehen. Gemessen an den Vorzügen die dies bringen würde (z.B. Korrekte Darstellung des Rheins, Landesgrenzen) sind die nötigen Modifikationen zu umfassend.

7 Weitere Programmänderungen

7.1 Sofortiges Berechnen der Route

Aus Benutzbarkeitsgründen wurde das Programmverhalten beim Planen einer Route verändert. Das Pflichtenheft schrieb vor, dass zunächst alle Wegpunkte gesetzt werden und der Nutzer anschließend manuell die Routenberechnung anstößt. Dieser zusätzliche Schritt ist jedoch nicht nötig, wenn die Route sofort berechnet wird, sobald genügend Wegpunkte gesetzt sind.

Da der Knopf *Route berechnen* somit obsolet ist, wurde dieser durch ein Suchfeld ersetzt, mit dem die Funktion *Zu einem Ort springen* nun direkt über die Hauptoberfläche erreichbar ist.



Betroffene Klasse: MainGUI

7.2 Ortsuchfunktion beachtet Groß- und Kleinschreibung

Die Funktion „Springe zu Ort“ konnte zu Missverständnissen führen, da diese nur Ortsnamen mit korrekter Groß- und Kleinschreibung akzeptierte. Dieses Verhalten wurde nun geändert, sodass die Suchfunktion nun tolerant gegenüber abweichender Groß-/Kleinschreibung ist.

Betroffene Klasse: CityData

Glossar

Altitude Höhe über Normal-Null.

Arc-Flag Beschleunigungstechnik für den Dijkstra◀ Algorithmus zur schnelleren Suche des kürzesten Pfades.

Bugfix Programmänderung zur Beseitigung eines Fehlers.

Codeüberdeckung Code wird überdeckt wenn er im laufenden Programm aufgerufen wird.

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

Java Programmiersprache.

Komponententest Test eines Teils des Programms. Dies kann auch nur den Test einer einzelnen Klasse beinhalten.

OSM steht für OpenStreetMap. Dies ist ein für jeden zugängliches Projekt, das weltweite geographische Daten sammelt..

Speedup die prozentuale Verbesserung einer Laufzeit.

String Zeichenkette.

Zoomstufe gibt den Grad der Vergrößerung/Verkleinerung an.