Bachelorarbeit am Institut für Informatik der Freien Universität Berlin

Entwicklung eines dynamischen Zeit-Weg-Linien-Diagramms für das Eisenbahn-Betriebs- und Experimentierfeld Berlin

Marian Sigler

sigler@inf.fu-berlin.de

Betreuer: Dipl.-Ing. Christian Blome

Prüfer: Prof. Dr. Carsten Schulte

Zweitprüferin: Prof. Dr. Elfriede Fehr

Berlin, 22. April 2015

Zusammenfassung

Im Eisenbahn-Betriebs- und Experimentierfeld (EBuEf) der TU Berlin wird zu Lehrzwecken auf einer Modellbahnanlage ein Eisenbahnbetrieb simuliert. Im realen Eisenbahnbetrieb wird der tagesaktuelle Fahrplan inklusive der aktuellen Betriebssituation und zu erwartender Fahrzeiten in grafischer Form angezeigt, während er im EBuEf bisher nur in statischer, gedruckter Form vorliegt. Im Rahmen dieser Arbeit wurde ein solches Zeit-Wege-Liniendiagramm und eine Fahrzeitenprognose für das EBuEf entwickelt.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder Ähnliches sind im Quellenverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Berlin, April 2015

Marian Sigler

Inhaltsverzeichnis

1 Einleitung

Der Eisenbahnbetrieb ist aufgrund verschiedener technischer Bedingungen von einem hohen Maß an Planung und Organisation geprägt. Ein sehr komplexer Bereich dieses Systems ist die Leit- und Sicherungstechnik, wozu insbesondere die Lenkung und Sicherung von Zugfahrten in Stellwerken zählt. Im Eisenbahn-Betriebs- und Experimentierfeld (EBuEf) der TU Berlin kann mittels echter Stellwerkstechnik ein Eisenbahnbetrieb auf einer Modellbahnanlage simuliert werden, um so die Prinzipien der Leit- und Sicherungstechnik praxisnah zu vermitteln.

Ein wichtiges Element ist dabei der Fahrplan. Er gibt vor, welche Züge zu welchen Zeiten über welche Gleise fahren sollen. Bislang liegen diese Daten nur statisch in gedruckter Form vor. Im realen Eisenbahnbetrieb wird über ein sogenanntes Zeit-Weg-Linien-Diagramm (ZWL-Diagramm) die aktuelle Betriebssituation auf einem Monitor dargestellt. Ferner kann der dort zugrunde liegende Fahrplan tagesaktuell angepasst werden[1].

Im Rahmen dieser Arbeit soll ein solches ZWL-Diagramm sowie eine Anwendung, die auf Basis der aktuellen Betriebssituation zukünftige Fahrzeiten der Züge prognostiziert, entwickelt werden.

Im folgenden Kapitel werden zunächst einige Hintergründe zur Eisenbahn und zum ZWL-Diagramm erläutert sowie das EBuEf vorgestellt. Anschließend werden die Anforderungen, die an die Anwendung gestellt werden, genannt. In Kapitel 3 folgt eine detaillierte Darstellung der Implementierung. Dabei werden zunächst die grundlegenden Designentscheidungen erläutert, danach Details der Implementierung dargestellt. Im vierten Kapitel schließt sich eine abschließende Betrachtung an.

[1] Absatz übernommen aus der Aufgabenstellung ([3], bearbeitet).

2 Hintergrund

2.1 Eisenbahn

Das System Eisenbahn weist einige Besonderheiten insbesondere im Vergleich zum Straßenverkehr auf, deren Kenntnis für das Verständnis der vorliegenden Arbeit sowie der Funktionsweise und des Aufbaus der entwickelten Software notwendig ist. Diese werde ich daher zunächst erläutern. Da die genauen Regelungen zum Teil sehr komplex sind, sind sie an manchen Stellen nicht bis ins letzte Detail korrekt dargestellt, sondern nur so weit, wie dies zum Verständnis notwendig ist.

Die Regelwerke und dahinterliegenden Philosophien unterscheiden sich zwischen verschiedenen Ländern teilweise stark. Das liegt daran, dass der Betrieb der Bahnnetze bis in die jüngere Vergangenheit i. d. R. behördenartig organisiert war und jede Bahn ihren eigenen Betriebs„stil“ entwickelt hat. Die folgenden Ausführungen gelten daher zum Teil nur für das von der Deutschen Bahn AG (DB) betriebene Netz.

Die Eisenbahn ist ein spurgeführtes Verkehrsmittel, die Züge verkehren mit Stahlrädern auf Stahlschienen. Das bietet große Vorteile hinsichtlich der möglichen Geschwindigkeiten und der Sicherheit, bringt jedoch einige Einschränkungen mit sich, die einen großen Einfluss darauf haben, wie das System Eisenbahn aufgebaut ist.

So ist die geringe Reibung zwischen Rad und Schiene zwar wegen niedriger Energieverluste von Vorteil, wirkt sich beim Bremsen aber nachteilig aus: Der Bremsweg ist bei gleicher Geschwindigkeit etwa zehn Mal so lang wie beim Straßenverkehr, ein Personenzug benötigt bei einer Geschwindigkeit von 160 km/h[1] einen Bremsweg von etwa 1000 m, Güterzüge, die länger und schwerer sind, i. d. R. wesentlich mehr. [2, S. 2][1]

Da diese Distanz nicht überblickbar ist, wird bei der Bahn nicht auf Sicht gefahren, sondern nach Signalen, die angeben, ob der folgende Streckenabschnitt befahren werden darf[2]. Weil ein möglicherweise *Halt* zeigendes Signal nicht rechtzeitig gesehen werden kann, um davor zum Halten zu kommen, wird zudem in Bremsweglänge[3] vor dem Hauptsignal ein Vorsignal aufgestellt, das dessen Stellung ankündigt[4]. Ebenso wird auch der Fahrweg nicht durch das Fahrzeug gewählt, sondern von außen durch die Stellung von Weichen vorgegeben. Durch diese beiden Eigenschaften geht die Verantwortung für die Verhinderung von Zugkollisionen, Entgleisungen u. dgl. im Wesentlichen auf die für diese Außensteuerung zuständige Stelle, die Stellwerke und das zugehörige Personal (die Fahrdienstleiter_innen), über.

[1] was keineswegs eine besonders hohe Geschwindigkeit ist, auch Regionalzüge fahren regelmäßig in diesem Geschwindigkeitsbereich.

[2] Die entsprechenden Signalbilder werden *Halt* („rot“) und *Fahrt* („grün“) genannt.

[3] d. h. normalerweise 1000 m vor dem Signal. Da bei Geschwindigkeiten von über 160 km/h die Bremswege übermäßig lang werden, wird dort ein anderes Verfahren verwendet, bei dem die Signalstellung direkt im Führerstand angezeigt wird. Letzteres wird aber im EBuEf nicht simuliert und ist daher für diese Arbeit irrelevant.

[4] Es zeigt (entsprechend des Begriffs des Hauptsignals) *Halt erwarten* oder *Fahrt erwarten*.

Die Verantwortung des_der Triebfahrzeugführer_in für die Zugsicherung beschränkt sich somit im Wesentlichen darauf, die gegebenen Signale[5] zu befolgen. Um menschliche Fehler zu vermeiden, wird er_sie dabei von der *Zugbeeinflussung* überwacht, die dafür sorgt, dass ein Zug automatisch gebremst wird, wenn er beispielsweise vor einem ein *Halt* zeigenden Signal nicht stark genug bremst oder es überfährt.

Um dieses *Fahren auf Signal* sicher umzusetzen, werden Bahnstrecken und Bahnhofsgleise in sogenannte Blockabschnitte eingeteilt. Jeder Blockabschnitt wird durch ein Signal gedeckt, das nur auf *Fahrt* gestellt werden darf, wenn die Bedingungen für eine sichere Zugfahrt sichergestellt sind. Diese sind – vereinfacht –, dass sich weder ein anderer Zug im betreffenden Gleisabschnitt befindet noch die Erlaubnis bekommen hat, in ihn hineinzufahren, sowie dass das Signal am Ende des Abschnitts *Halt* zeigt (wenn nicht für den folgenden Abschnitt diese Bedingungen ebenfalls erfüllt sind). Außerdem gibt es einen sogenannten Durchrutschweg hinter dem Signal am Ende des Abschnitts, der ebenfalls frei sein muss[6]. Die Prüfung all dieser Bedingungen wird heute, um menschliche Fehler zu vermeiden, im Regelfall von der Stellwerkstechnik übernommen. [2, S. 9, S. 39]

Diese Regeln sichern die Zugfolge auf einer freien Strecke ohne Weichen. Weichen gibt es nach dem deutschen Regelwerk nur innerhalb von bestimmten Betriebsstellen wie Abzweigstellen u. dgl., aber insbesondere in Bahnhöfen. Dort kommen als Bedingungen hinzu, dass alle im Fahrweg liegenden Weichen passend gestellt sein müssen und keine weiteren kreuzenden oder einmündenen Fahrzeugbewegungen zugelassen sind. Das letzte Signal vor einem Bahnhof heißt Einfahrsignal; bevor es auf *Fahrt* gestellt wird, werden diese Bedingungen überprüft. Analog beim Ausfahrsignal, das i. d. R. am Ende des Bahnsteigs steht, wo außerdem die Bedingungen für die Fahrt auf die freie Strecke erfüllt sein müssen (siehe oben).

Dieses Verfahren bedeutet, dass zwei Züge nur in einem großen Abstand zueinander fahren können. Die Zeit, während der ein Blockabschnitt von einem Zug belegt ist, ist dabei um einiges länger als die reine Fahrzeit. Diese *Sperrzeit* setzt sich aus folgenden Zeiten zusammen [2, S. 46, S. 115]:

Signalsicht- und Annäherungsfahrzeit Das Signal muss schon auf *Fahrt* stehen, wenn sich der Zug dem *Vor*signal nähert, da im Falle eines Vorsignals, das *Halt erwarten* zeigt, schon mit der Bremsung begonnen werden muss.

Fahrzeit und Räumfahrzeit Die Fahrzeit gibt nur die Zeit an, die die Zugspitze braucht, um den Streckenabschnitt zu durchqueren. Hinzu kommt die Zeit, die der Rest des Zuges benötigt, um den Streckenabschnitt und den Durchrutschweg zu räumen.

Fahrstraßenbilde- und -auflösezeit Vor und nach einer Zugfahrt wird für ver-

[5] Hierzu werden neben den auch umgangssprachlich so genannten Signalen, die Halt oder Fahrt anordnen (eigentlich: *Hauptsignale*), auch Schilder, die Geschwindigkeitsbeschränkungen u. dgl. anordnen, gezählt. Im Folgenden wird der Begriff in der engeren Bedeutung verwendet.

[6] Die Zugbeeinflussung stellt zwar sicher, dass ein Zug nur langsam auf ein *Halt* zeigendes Signal zufahren kann, und leitet zudem eine Zwangsbremsung ein, wenn er es überfährt. Jedoch kann auch mit diesem mehrstufigen System ein Befahren einer Strecke von i. d. R. 200 m hinter dem Signal nicht ausgeschlossen werden; Das ist der Durchrutschweg, der folglich ebenfalls freigehalten werden muss.

schiedene technische Vorgänge, z. B. das Umstellen der Weichen, Zeit benötigt. Die genaue Dauer schwankt, abhängig vom Automatisierungsgrad des Stellwerks, zwischen einigen Sekunden und bis zu zwei Minuten.

Die Sperrzeiten hintereinander fahrender Züge dürfen sich nicht überlappen, somit definieren die Sperrzeiten die minimal mögliche Zugfolge auf einer Strecke. In der Praxis ergibt sich selbst bei optimalen Verhältnissen (sehr kurze Blockabschnitte, artenreiner Verkehr[7]) eine Fahrzeugfolge von einigen Minuten [1].

2.1.1 Zum Zeit-Wege-Linien-Diagramm

Schon da nur ein Zug gleichzeitig auf einem Streckenabschnitt fahren kann und Überholungen und – im Fall von eingleisigen Strecken – Kreuzungen von Zügen nur an geeigneten Stellen möglich sind, ist eine vorherige Koordination verschiedener Zugfahrten beim Schienenverkehr notwendig. Aus den genannten Überlegungen ergeben sich weitere Gründe dafür, so kann die Streckenkapazität erhöht werden, wenn beispielsweise erst mehrere Güterzüge hintereinander fahren und danach eine Gruppe schneller Fernzüge.

Natürlich ist ein Fahrplan auch innerhalb eines Verkehrsunternehmens für die Information der Fahrgäste sowie die Personal- und Fahrzeugplanung notwendig. Die Besonderheit der Bahn (beispielsweise im Vergleich zum Fernbus) ist jedoch, dass auch die Nutzung des Netzes im Vorhinein genau geplant werden muss, weil i. d. R. Züge vieler verschiedener Eisenbahnverkehrsunternehmen auf einer Strecke unterwegs sind und deren Interessen sowie die des Infrastrukturbetreibers in Einklang gebracht werden müssen.

Ein wichtiges Hilfsmittel für die Erstellung des Fahrplans ist das Zeit-Wege-Linien-Diagramm (ZWL-Diagramm), auch Bildfahrplan oder ZWL-Bild genannt. Darin sind die Fahrzeiten der Züge an einer Orts- und einer Zeitachse abgetragen. In Deutschland

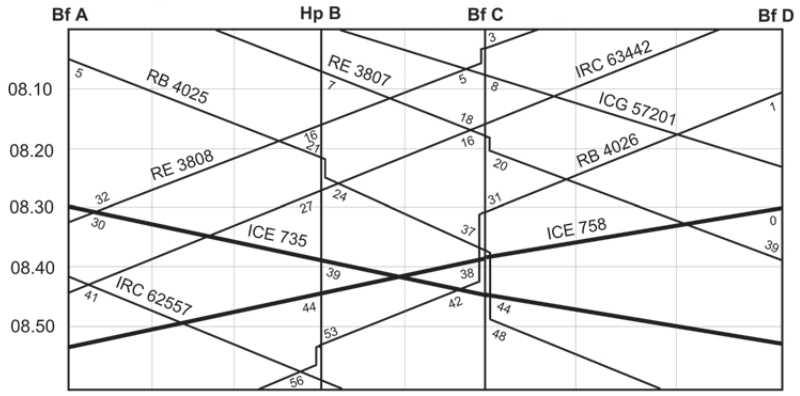

Abbildung 1: Idealisiertes Beispiel eines ZWL-Diagramms (Quelle: [2, S. 171], bearbeitet)

[7] Verkehr von Zügen mit ähnlicher Geschwindigkeit und Haltehäufigkeit, keine Mischung von z. B. Fern-, Regional- und Güterverkehr.

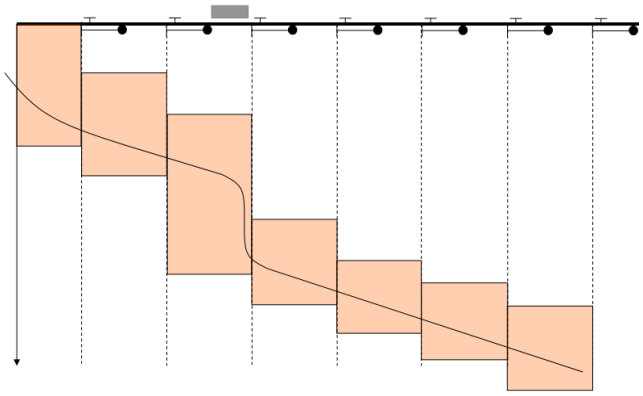

Abbildung 2: Zeit-Wege-Linie und Sperrzeitentreppe einer Zugfahrt. (Quelle: [1])

hat sich die „liegende“ [2, S. 170] Form eingebürgert [4], bei der die Wegachse horizontal und die Zeitachse vertikal ist, eine umgekehrte Darstellung wäre aber ebenso möglich. Im liegenden Diagramm sind Halte durch senkrechte Linien erkennbar; je schneller ein Zug fährt, desto flacher ist seine Linie. Ein Beispiel eines ZWL-Diagramms ist in Abbildung 1 dargestellt.

Das ZWL-Diagramm stellt anschaulich die auf einer Strecke verkehrenden Züge dar und ermöglicht eine schnelle Abschätzung, wo Züge zu nah hintereinander fahren oder wo sich auf einer eingleisigen Strecke Züge auf freier Strecke kreuzen müssten – Aufgaben, die mit einem tabellarischen Fahrplan alleine nur sehr mühsam erledigt werden könnten.

Diese Abschätzung erfordert jedoch Wissen über die minimal mögliche Zugfolge, die außerdem von Zug zu Zug schwanken kann. Da eine genaue Beurteilung nur unter Berücksichtigung der Sperrzeiten möglich ist, ist es naheliegend, diese mit einzublenden. Die Belegung eines Streckenabschnittes wird dabei durch ein Rechteck von der Länge des jeweiligen Streckenabschnitts und mit einer der Sperrzeit entsprechenden Höhe symbolisiert. Diese Darstellung wird *Sperrzeitentreppe* genannt. Mit ihrer Hilfe kann zweifelsfrei beurteilt werden, ob eine bestimmte Zugabfolge möglich ist[8] und wie viel Pufferzeit zwischen den Zugfahrten besteht. Abbildung 2 zeigt eine solche Sperrzeitentreppe.

Neben der Fahrplanerstellung ist das ZWL-Bild auch für die Steuerung des alltäglichen Betriebs ein wichtiges Hilfsmittel. Bei DB Netz[9] wird es von Fahrdienstleiter_innen und Disponent_innen[10] genutzt, um einen Überblick über die aktuelle Betriebssituation zu bekommen (siehe Abbildung 3). In das ZWL-Diagramm sind auch Schnittstellen zu anderer Software integriert, mit denen weitere Informationen über die Züge abgerufen werden und bearbeitet werden können. So können Dispositionsentscheidungen in das

[8] Hier wird nur die freie Strecke betrachtet. Weitere Konflikte können entstehen, beispielsweise wenn Züge an Abzweigstellen oder bei der Ausfahrt aus dem Bahnhof das Gegengleis kreuzen müssen.

[9] Tochtergesellschaft der Deutschen Bahn AG, die für den Betrieb des Netzes zuständig ist.

[10] Fahrdienstleiter_innen sind für die Steuerung des Zugverkehrs auf Stellwerksebene und die Kommunikation mit Triebfahrzeugführer_innen zuständig. Disponent_innen sind ihnen übergeordet, für einen größeren Bereich zuständig und koordinieren den Betriebsablauf.

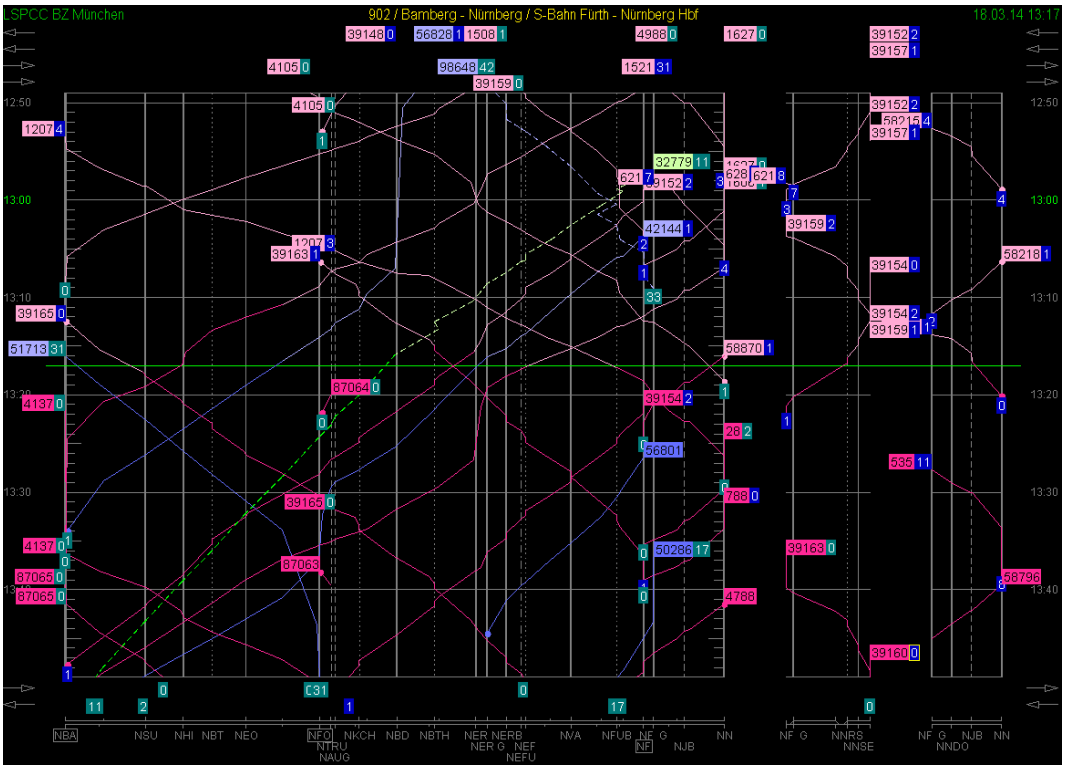

Abbildung 3: Das ZWL-Bild, wie es von Fahrdienstleiter_innen und Disponent_innen der DB Netz genutzt wird. Die grüne Linie stellt die aktuelle Zeit dar, die blasseren Linien (darüber) Zugläufe der Vergangenheit. In den roten/blauen Kästen stehen die Zugnummern, die Zahlen in den blauen/türkisen Kästchen geben die Verspätung an. Die Abbildung zeigt eine längere Strecke und am rechten Rand zwei kurze Nebenstrecken. (Quelle: [5])

System eingepflegt werden, z. B. einen Güterzug außerplanmäßig halten zu lassen, um eine Überholung durch einen verspäteten Personenzug zu ermöglichen [4].

Das ZWL-Bild zeigt dabei für die Vergangenheit die realen Fahrzeiten der Züge bzw. für die Zukunft prognostizierte Werte an. Diese Prognose wird jedoch bei der von DB Netz verwendeten Software nur für jeden Zug individuell ermittelt, d. h. es wird zwar berücksichtigt, dass Verspätung durch z. B. kürzere Haltezeiten und Ausnutzung von Pufferzeiten bei der Fahrzeit abgebaut werden kann, jedoch werden Verspätungen, die vorraussichtlich durch verspätete vorausfahrende Züge u. dgl. entstehen, nicht einberechnet [6].

2.2 Das Eisenbahn-Betriebs- und Experimentierfeld

Das Eisenbahn-Betriebs- und Experimentierfeld (EBuEf) gehört zum Fachgebiet Schienenfahrwege und Bahnbetrieb der Technischen Universität Berlin [7]. Es besteht neben einer Außenanlage mit einigen Exponaten in Originalgröße aus einer Modellbahnanlage mit daran angeschlossener echter Stellwerkstechnik verschiedener

Abbildung 4: Überblick über die Anlage (Quelle: EBuEf e.V.)

Generationen[11]. Es wird sowohl für Lehrveranstaltungen des Fachgebiets als auch für Schulungen Externer genutzt, um Eisenbahnbetrieb und Eisenbahnsicherungstechnik wirklichkeitsnah zu vermitteln.

Dazu besetzen die Teilnehmenden als *Fahrdienstleiter_innen* die Stellwerke und es wird ein realistischer Betriebsablauf simuliert. Das „Ausgabemedium“ der Simulation ist die Modellbahnanlage, so dass die Auswirkungen der eigenen Handlungen unmittelbar sichtbar sind.

Die Anlage besteht aus einer zweigleisigen, ringförmigen Strecke[12] mit vier größeren Bahnhöfen sowie einem davon abzweigenden eingleisigen Netz mit insgesamt sechs weiteren Haltepunkten und kleinen Bahnhöfen.

2.2.1 Technische Umsetzung

Die zentrale Schnittstelle aller Komponenten stellt eine Datenbank dar, an die alle Komponenten angeschlossen sind.

Die Bedienhandlungen an den analog arbeitenden Stellwerken werden digital abgegriffen[13]. Anhand dieser Daten wird dann die Anlage gesteuert, d. h. Weichen und

[11] Diese reichen von rein mechanisch arbeitenden Stellwerken, bei denen im Original die Signale und Weichen über Drahtseile gestellt werden über elektrische Stellwerke auf Relaisbasis bis zu modernen elektronischen Stellwerken, die am Computer bedient werden. Alle diese Generationen sind auch beim Original heute noch im Einsatz.

[12] im Folgenden: „der Ring“

[13] Im Fall der (selbst entwickelten) elektronischen Stellwerke liegen sie natürlich schon digital vor.

Abbildung 5: Das Stellwerk und (teilweise verdeckt) die Gleise des Bahnhofs Derau. *Links im Hintergrund die Stellhebel und Anzeige eines anderen Stellwerks älterer Bauart. Am rechten Bildrand ist der zugehörige Arbeitstisch zu sehen, auf dem dortigen Monitor soll später das ZWL-Bild angezeigt werden. (Quelle: EBuEf e.V.)*

Signale umgestellt sowie die Züge angehalten bzw. ihre Geschwindigkeit gesteuert. (Teilweise werden die Züge auch durch einen direkt vom Stellwerk gesteuerten stromlosen Abschnitt angehalten.) Außerdem werden die Daten zur Verwendung durch andere Komponenten in der Datenbank gespeichert.

Es gibt einen eigens für das EBuEf entwickelten Fahrplan, auch dieser liegt in der Datenbank vor. Der Fahrplan erstreckt sich über eine Dauer von sechs Stunden, wobei mit späterer Zeit die Zugdichte langsam zunimmt. So kann durch die Auswahl eines Zeitbereichs der Schwierigkeitsgrad einer Übung variiert werden. Nur die diesen Bereich betreffenden Fahrplandaten werden zur Vorbereitung einer Sitzung in zwei jeweils schemagleiche Tabellen kopiert; meine Software nutzt nur letztere Fahrplantabellen.

Bisher ist dieser Fahrplan den „Fahrdienstleiter_innen“ im EBuEf nur in Form einer tabellarischen Auflistung der Ankunft- und Abfahrtszeiten an der/den ihrem Stellwerk zugeordneten Betriebsstelle(n)[14] bekannt. Den genauen Fahrplan und die aktuellen Fahrzeiten an benachbarten Betriebsstellen kennen sie nicht, auch von Verspätungen erfahren sie nur durch telefonische Mitteilungen benachbarter Fahrdienstleiter_innen.

[14] Oberbegriff für Bahnanlagen wie Bahnhöfe, Haltepunkte, Abzweigstellen

Simulations-Uhrzeit Der Betrieb richtet sich nicht nach der aktuellen tatsächlichen Uhrzeit, sondern es gibt eine Simulations-Uhrzeit, so dass unabhängig von der tatsächlichen Tageszeit nach einer beliebigen Uhrzeit gefahren werden kann, außerdem kann die Zeit bei Bedarf angehalten werden. Es gibt im EBuEf Analog-Wanduhren, die so gesteuert werden, dass sie die Simulationszeit anzeigen. Diese Steuerung übernimmt ein Server, bei dem auch über ein einfaches TCP-Protokoll die aktuelle Uhrzeit und der Status (noch nicht gestartet, gestartet, pausiert) abgefragt werden kann.

2.2.2 Datenstruktur

Für meine Software sind nur wenige der fast 100 Datenbanktabellen relevant. Diese werden im Folgenden beschrieben.

`fahrplan_sessionzuege` enthält Basisdaten zu allen Zügen, die im für die Sitzung ausgewählten Zeitraum verkehren, z. B. Zuggattung (Verweis auf `zuege_zuggattungen`, s. u.) und Zugnummer (die `id` des Eintrags, der *Primary Key*, ist nicht mit der Zugnummer identisch).

`fahrplan_sessionfahrplan` enthält die Fahrzeiten der Züge. Jede Zeile der Tabelle enthält die Ankunfts- und Abfahrtszeiten an einer Betriebsstelle[15] sowie (in einem Bahnhof) welches Gleis der Zug nutzt, außerdem einige weitere Informationen, z. B. ob der Zug auf der folgenden freien Strecke das Gegengleis benutzt (was im ZWL-Bild anders dargestellt werden soll) und Ähnliches. Die Zeit- und Gleisangaben wurden dabei während dieser Arbeit aufgeteilt in vier verschiedene Gruppen (in Klammern die Namen der entsprechenden Spalten[16]):

Plan-Daten (`ankunft_plan`, `abfahrt_plan`, `gleis_plan`) Die vom Fahrplan vorgegebenen Zeiten. Diese werden nur am Anfang einer Sitzung aus den statischen Fahrplantabellen hierher kopiert und danach nicht mehr verändert.

Soll-Daten (`..._soll`) Diese Spalten werden ebenfalls am Anfang einer Sitzung befüllt und entsprechen zunächst den Plan-Daten. Wenn jedoch z. B. ein Zug an einem Bahnhof einen außerplanmäßigen Halt einlegen soll, um einen anderen überholen zu lassen, kann das in diese Spalten eingetragen werden. Hierfür können nicht die Plan-Spalten mitbenutzt werden, da dann z. B. die Verspätung eines Zuges nicht mehr berechnet werden könnte (diese bezieht sich ja grundsätzlich auf die Plan-Daten).

Ist-Daten (`..._ist`) Hier werden, nachdem eine Zugfahrt stattgefunden hat, die tatsächlichen Fahrzeiten und benutzten Gleise eingetragen.

[15] „Betriebsstelle“ ist hier etwas weiter gefasst als nach der eigentlichen Definition: Es gibt jeweils auch eigene Fahrplaneinträge für die Einfahr- und Ausfahrsignale eines Bahnhofs, auch wenn diese per Definition gemeinsam *eine* Betriebsstelle darstellen.

[16] wie sie in der Datenbank heißen; in meinem Code verwende ich durchgehend englische Bezeichner.

Prognose-Daten (`ankunft_prognose`, `abfahrt_prognose`) In diese Spalten werden von der innerhalb dieser Arbeit entwickelten Prognosefunktion die prognostizierten Fahrzeiten eingetragen. Das Gleis ändert sich bei Verspätungen nicht automatisch[17], das vorraussichtlich benutzte Gleis entspricht daher immer dem Wert der Spalte `gleis_soll`.

`zuege_zuggattungen` ordnet den verschiedenen Zuggattungen (das sind z. B. IC, ICE, RB, RE) eine Kurz- und Langbezeichnung (z. B. *IC* und *InterCity*) sowie Verkehrsart (Fern-, Regional-, Güter- und sonstiger Verkehr) zu. Anhand letzterer werden die Züge im ZWL-Diagramm verschieden eingefärbt.

`fahrplan_mindesthaltezeiten` Für die Prognose ist es wichtig, zu wissen, welcher Teil der Haltezeit in einem Bahnhof in jedem Fall benötigt wird und welcher Pufferzeit ist und bei Verspätung eingespart werden kann. Zukünftig sollen diese Daten direkt aus dem Fahrplanerstellungsprogramm exportiert und für jeden Halt die jeweilige Mindesthaltezeit in die Datenbank eingetragen werden. Da das aber bisher noch nicht implementiert ist, wurde diese Tabelle eingerichtet, in der von Gleis, Bahnhof und Zuggattung abhängende Standardwerte eingetragen werden können.

2.3 Anforderungen

Ziel der Arbeit ist, dass den Fahrdienstleiter_innen in Zukunft neben den statischen, gedruckten Fahrplaninformationen auch in Form eines ZWL-Diagramms die aktuelle Betriebssituation (also auch Verspätungen etc.) zur Verfügung steht, wie dies bei der Deutschen Bahn der Fall ist. Außerdem sollen anhand der aktuellen Betriebssituation die zu erwartenden zukünftigen Fahrzeiten der Züge prognostiziert werden, was bei der DB in dieser Form noch nicht existiert [6].

Im Folgenden werden die Anforderungen genannt, die sich an das ZWL-Diagramm und die Prognose stellen. Nicht alle dieser Anforderungen sollten innerhalb dieser Arbeit erfüllt werden, es sollte aber bei der Konzeption darauf geachtet werden, dass diese später einfach hinzugefügt werden können.

Der grundsätzliche Aufbau des ZWL-Diagramms besteht aus der horizontalen Achse mit den Kürzeln der Betriebsstellen der angezeigten Strecke und einer Zeitachse in der vertikalen sowie den die Fahrzeiten der Züge darstellenden Linien. Es soll einfach möglich sein, den anzuzeigenden Zeitraum in die Zukunft bzw. Vergangenheit zu verschieben, auch soll die Länge dieses dargestellten Zeitintervalls variiert werden können. Außerdem soll die Auswahl der anzuzeigenden Streckenabschnitte flexibel anpassbar sein: sowohl die Anzeige nur eines Ausschnitts einer Strecke als auch die gleichzeitige Anzeige mehrerer verschiedener Strecken sollen möglich sein.

[17] sondern nur durch eine dispositive Entscheidung, diese würde jedoch in die Soll-Spalte eingepflegt.

Im Diagramm soll die aktuelle Simulations-Uhrzeit durch eine horizontale Linie markiert sein, der in der Vergangenheit liegende Teil der Zuglinien soll blasser dargestellt sein. Die gesamte Anzeige soll sich regelmäßig automatisch aktualisieren.

Zu jeder Zuglinie soll gut sichtbar die Zugnummer angezeigt werden, da diese einen Zug eindeutig identifiziert und für viele betriebliche Vorgänge verwendet wird. Beim Überfahren der Zuglinie mit der Maus sollen weitere Informationen zu diesem Zug angezeigt werden (Zuggattung, Start, Ziel, Höchstgeschwindigkeit, Länge, ...), auch soll nach einer gegebenen Zugnummer gesucht werden können. Verschiedene Zuggattungen sollen in verschiedenen Farben dargestellt werden, diese Zuordnung soll leicht anpassbar sein. Ebenso sollen verschiedene Themes ausgewählt werden können, insbesondere solche mit hellem und mit dunklem Hintergrund.

Das Diagramm soll die realen Fahrzeiten der Züge in der laufenden Simulation anzeigen, außerdem sollen aus diesen Zeiten zukünftige Fahrzeiten prognostiziert werden. Die Prognose soll auch eine Konflikterkennung beinhalten, so dass beispielsweise eine vorraussichtlich entstehende Verspätung eines zum Prognosezeitpunkt pünktlichen Zuges wegen eines verspäteten, vor ihm fahrenden Zuges korrekt erkannt wird.

Im Diagramm sollen Sperrungen von Strecken und -gleisen angezeigt werden, es soll auch möglich sein, solche Sperrungen einzutragen. Besondere Zugfahrten (Sperrfahrten, Lademaßüberschreitungen[18]) sollen ebenfalls besonders hervorgehoben werden. Ferner soll möglich sein, Kommentare zu Zügen einzutragen sowie solche, die einer Uhrzeit und einem Ort (Betriebsstelle oder Strecke zwischen zwei Betriebsstellen) zugeordnet sind.

In Zukunft soll das ZWL-Diagramm von einer reinen Anzeige weiterentwickelt werden und (wie die entsprechende Software der DB Netz) eine Eingabe von Dispositionsentscheidungen ermöglichen, wie z. B. einen verspäteten Zug warten zu lassen, um einen anderen, pünktlichen, nicht zu behindern. Um solche Entscheidungen zu erleichtern, soll auch die Einblendung von Sperrzeitentreppen möglich sein. Diese beiden Funktionen sollen bei der Konzeption mit bedacht werden.

[18] Sperrfahrten sind Fahrten in gesperrte Gleise, z. B. Bau- oder Hilfszüge. Als Lademaßüberschreitungen („Lü“) werden Züge bezeichnet, bei denen wegen erhöhter Breite Einschränkungen für das Nachbargleis gelten. [8]

3 Implementierung

In diesem Kapitel werde ich die technische Umsetzung der Anwendung beschreiben. Im ersten Abschnitt erläutere ich die grobe Struktur und die genutzten Technologien, darauf folgt zunächst eine genauere Darstellung der einzelnen Teile der Anwendung, danach eine detaillierte Betrachtung ausgewählter Codestellen.

Für alle Abschnitte gilt, dass teilweise Funktionalitäten beschrieben sind, die noch nicht oder noch nicht vollständig umgesetzt sind. Diese sind bereits hier mit genannt und z. T. auch beschrieben, wenn sie bei der Implementierung der Anwendung schon mit berücksichtigt wurden.

3.1 Vorüberlegungen

Von Seiten des EBuEf war lediglich eine webbasierte Implementierung vorgegeben, da bereits andere an den Stellwerksarbeitsplätzen genutzte Anwendungen so implementiert sind, insbesondere die Bedienoberfläche der elektronischen Stellwerke. Auf diese Weise besteht die größte Flexibilität hinsichtlich des genutzten Betriebssystems und es ist nur eine einmalige, zentrale Installation nötig.

Die Anwendung ist daher aufgeteilt in ein Frontend, das im Browser läuft und für die grafische Darstellung der Fahrplandaten zuständig ist, und ein Backend, das auf einem Server läuft und dem Frontend die benötigten Daten in einer aufbereiteten Art zur Verfügung stellt.

3.1.1 Frontend

Als Browser werden nur Chrome/Chromium[1] und Firefox in den aktuellen Versionen genutzt, auf andere Browser und insbesondere ältere Versionen muss also keine Rücksicht genommen werden.

Die grafische Darstellung besteht hauptsächlich aus Linien, Flächen und kleineren Textelementen, die unabhängig voneinander manipuliert werden können sollen, damit ist SVG perfekt für diesen Anwendungszweck geeignet. SVG ist eine auf XML basierende Sprache für Vektorgrafiken, die sich direkt in HTML-Dokumente integrieren lässt. Als grafische Elemente stehen Grundformen wie Rechtecke, Geraden, Kreise, aber auch Textelemente, Polygone und frei definierbare Pfade zur Verfügung. SVG-Elemente können (wie HTML-Elemente) mit JavaScript manipuliert und mit CSS gestaltet werden, auch eine hierarchische Gruppierung ist möglich. Auf SVG-Elemente und auch ganze -Gruppen können nachträglich Transformationen (z. B. Verschieben und Verzerren) angewandt werden. Weitere Gründe für SVG sind längere persönliche Erfahrung damit und dass die beiden zu berücksichtigenden Browser schon seit einiger Zeit nativ SVG unterstützen.

[1] Beide sind prinzipiell der selbe Browser: Chromium ist der quelloffene Teil, Chrome enthält einige proprietäre Erweiterungen von Google. Die Rendering- und JavaScript-Engines sind jedoch die selben.

Eine rein statische Umsetzung ist schon deshalb nicht zweckmäßig, da selbst für die Kernfunktionalität die Seite ständig im Abstand von wenigen Sekunden neu geladen werden müsste. Hinzu kommen viele Funktionalitäten, die sich nur dynamisch angenehm implementieren lassen, wie die Wahl des angezeigten Zeitfensters oder die Eingabe von Kommentaren.

Als clientseitige Sprache liegt im Browser aufgrund der nativen Unterstützung JavaScript nahe. Pures JavaScript bietet jedoch für viele Funktionalitäten (z. B. DOM-Manipulationen und Ajax-Abfragen) nur primitive Schnittstellen an, daher verwende ich zusätzlich die Bibliothek *jQuery*[2], die dafür einfacher zu benutzende Funktionen bereithält.

Für die Erzeugung des SVG-Codes gab es mehrere Möglichkeiten. Eine dynamische Erstellung aller SVG-Elemente mit purem JavaScript scheidet aufgrund der Komplexität aus. Es wäre möglich, SVG-Code für Teile der Anzeige, beispielsweise die Zuglinien, serverseitig zu generieren und auf Clientseite nur einzubinden. Die dritte Möglichkeit ist die Verwendung einer Bibliothek, die die Erzeugung von SVG-Elementen abstrahiert. Letztere Lösung bietet die meiste Flexibilität. Nach kleineren Tests mit verschiedenen Bibliotheken habe ich mich für *svg.js*[3] entschieden. Es zeichnet sich durch eine schlanke Syntax, Erweiterbarkeit und guter Unterstützung von Gruppierung (SVG-Element `<g>`) und Transformationen (Verschieben, Verzerren, etc. von SVG-Elementen) aus.

3.1.2 Backend

Hier hatte ich weitgehend freie Wahl. Die einzige Einschränkung ist, dass eine Anbindung an die MySQL-Datenbank möglich sein muss. Meine Wahl fiel wegen eigener langjähriger Erfahrungen auf die Programmiersprache *Python*[4]. Python zeichnet sich durch eine starke, aber dynamische Typisierung, eine zweckmäßige Kombination verschiedener Programmierparadigmen (v. a. objektorientierte und funktionale Programmierung), eine elegante Syntax und eine große Anzahl verfügbarer Bibliotheken aus.

Für die Datenbankanbindung verwende ich die Bibliothek *SQLAlchemy*[5]. Sie kann einerseits als Object Relational Mapper[6] (ORM) genutzt werden, gleichzeitig ist es aber auch – für komplexere Anfragen oder wenn z. B. nur Werte einer Spalte benötigt werden – möglich, Anfragen direkt zu formulieren, jedoch ohne direkt SQL-Code schreiben zu müssen, was unflexibel und fehleranfällig ist. SQLAlchemy ermöglicht die Nutzung verschiedener Datenbank-Management-Systeme, i. d. R. ohne dass etwas am Code verändert werden muss.

[2] https://jquery.com/

[3] http://svgjs.com/

[4] https://www.python.org/

[5] http://www.sqlalchemy.org/

[6] Datenbanktabellen werden Python-Klassen zugeordnet, eine Datenbankabfrage auf eine Tabelle gibt eine oder mehrere Instanzen dieser Klasse zurück; in die Datenbank zu schreiben, ist ebenfalls direkt über diese Objekte möglich.

Für die HTTP-seitige Anbindung kommt das Webframework *Flask*[7] zum Einsatz. Flask verteilt eingehende Anfragen an verschiedene Prozesse/Threads, bildet URLs auf Python-Funktionen ab (*Routing*) und bietet viele Hilfsfunktionen, die bei der Bearbeitung von HTTP-Anfragen hilfreich sind. Da für die meiste Funktionalität von Flask auf die Low-Level-Bibliothek *Werkzeug*[8] zurückgegriffen wird, ist auch bei Flask neben der Nutzung als Framework ein Zugriff auf einer niederen Ebene möglich.

Im Jahr 2010 wurde Version 3 von Python veröffentlicht, die jedoch wegen einiger tiefgreifender Änderungen inkompatibel zu Version 2 ist. Insbesondere einfacher Python-2-Code kann jedoch automatisiert zu Python 3 konvertiert werden. Python 3 soll langfristig Python 2 ersetzen, inzwischen sind die meisten Bibliotheken auch unter Python 3 lauffähig [9]. Insbesondere im Webbereich gibt es aber noch Probleme, weshalb die Entwickler von Flask dazu raten, vorerst weiter Python 2 zu benutzen [10]. Deshalb habe ich mich dazu entschieden, die Anwendung in Python 2 zu entwickeln, ich gehe aber davon aus, dass sie sich später mit geringem Aufwand zu Python 3 konvertieren lässt.

3.2 Frontend

3.2.1 Grundsätzliches

Eine der Anforderungen war, dass das Diagramm mehrere verschiedene Strecken gleichzeitig darstellen kann, um z. B. einem_r für eine Abzweigstelle zuständigen Fahrdienstleiter_in die Möglichkeit zu geben, den Verkehr auf allen drei (oder mehr) benachbarten Strecken zu beobachten. Hier gab es zwei Implementierungsmöglichkeiten. Zum einen hätte, da es nur ca. zehn Arbeitsstellen gibt, für jede eine optimale Ansicht konfiguriert und über eine eigene URL abrufbar gemacht werden können. Die andere Möglichkeit war, den Code zur Anzeige des Graphen abzutrennen und zu ermöglichen, durch mehrfachen Aufruf dieses Codes mehrere Graphen nebeneinander anzuzeigen, die jeweils verschiedene Strecken(abschnitte) anzeigen[9].

Ich habe mich für die zweite Möglichkeit entschieden. So muss nur eine sehr geringe Anzahl Strecken konfiguriert werden, da durch Auswahl eines Ausschnitts und ggf. Anzeige mehrerer Strecken beliebige Kombinationen möglich sind. Auch ist es mit weniger Aufwand verbunden, die Anwendung anzupassen, falls sich die Streckenanordnung des EBuEf ändern sollte. Außerdem ist es ohne weiteres möglich, die verschiedenen Strecken in unterschiedlichem Detailgrad anzuzeigen (z. B. eine Strecke nur bis zum nächsten Bahnhof, die andere vollständig). Es entsteht jedoch ein höherer Programmieraufwand u. a. zur Synchronisation zwischen den angezeigten Graphen und zur passenden Positionierung.

Ein weiteres zentrales Element ist die Zeitleiste. Sie hat neben ihrer Funktion als

[7] http://flask.pocoo.org/

[8] http://werkzeug.pocoo.org/

[9] Im Folgenden bezeichnet *Diagramm* die Gesamtheit aller angezeigten Elemente, während *Graph* die Darstellung einer Strecke (die Zuglinien und Achsenbeschriftung) meint. Das Diagramm kann mehrere dieser Graphen enthalten.

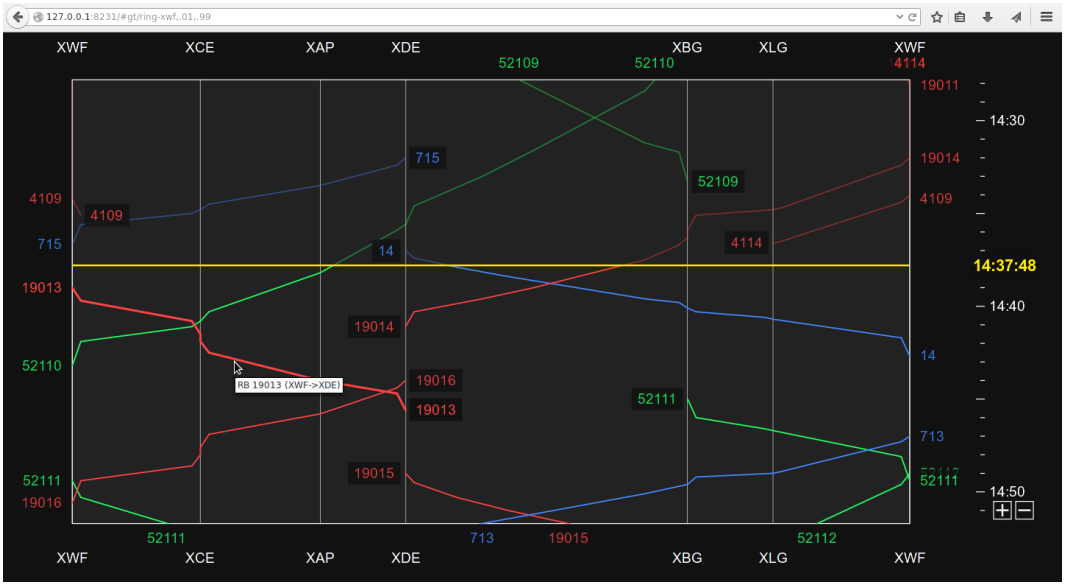

Abbildung 6: Ansicht des ZWL-Diagramms mit Anzeige einer einzigen Strecke, dem Ring. (Erkennbar am Bahnhof XWF und dem durchlaufenden Zug 52111 am unteren Bildrand; die meisten anderen Züge wechseln in XWF die Zugnummer, siehe z. B. 715/713, blau.)

vertikale Achse der Graphen noch eine weitere Funktion: durch Ziehen (*drag&drop*) kann man den angezeigten Zeitraum verändern, um weiter in die Zukunft oder Vergangenheit zu schauen. Mit Buttons kann der „Zeit-Zoom“, d. h. die zeitliche Auflösung der Anzeige, stufenweise verändert werden.

3.2.2 zu den verwendeten Technologien

SVG basiert auf XML, die grafischen Elemente eines Dokuments sind durch entsprechende SVG-Elemente im XML-Baum repräsentiert. Sie werden in der Code-Reihenfolge gezeichnet, d. h. im Dokument weiter unten stehende Elemente überdecken vorher definierte.

Es gibt eine Reihe verschiedener Basis-Elemente, verwendet werden Rechtecke (`<rect>`), Linien (`<line>`), Polygonzüge (`<polyline>`) und Text (`<text>`); außerdem gibt es z. B. Kreise, Ellipsen sowie Pfade, mit denen beliebige Formen gezeichnet werden können. Zu jedem Element kann eine Füllung und ein Rahmen (bzw. Strich bei Linien etc.) sowie dessen Breite definiert werden, dabei wird volle Alpha-Transparenz[10] unterstützt. Alle Elemente[11] können in Gruppen zusammengefasst werden, in dem sie als Kindelemente eines `<g>`-Elements eingeordnet werden.

[10] Der Alpha-Kanal ist ein vierter Kanal neben den Farbkanälen Rot, Grün und Blau, in dem die Transparenz gespeichert wird. So sind beliebige Abstufungen zwischen unsichtbar und vollständig deckend möglich.

[11] Auch Gruppen sind Elemente, d. h. es sind auch verschachtelte Gruppen möglich.

Allen Elementen können (wie bei HTML) Klassen zugewiesen werden (über das Attribut `class`), mittels derer eine Selektion in CSS-Regeln möglich ist. Alle grafischen Eigenschaften, also Füllfarbe, Rahmen-/Strichfarbe und -stärke sowie Transparenz, können mittels CSS definiert werden. Eine direkte Zuweisung dieser Eigenschaften ist auch möglich, wird aber nicht genutzt, da eine Trennung von Struktur und Aussehen sauberer und auch besser handhabbar ist.

Weiter ist es möglich, Elemente zu transformieren; in der Anwendung wird nur die Translation genutzt, aber auch Verzerrungen, Rotationen u. dgl. sind möglich. Weiterhin kann der sichtbare Bereich eines Elements eingeschränkt werden: Die Teile des Elements, die außerhalb eines definierten Ausschneidepfades liegen, sind dann nicht sichtbar. Zusätzlich zu diesem binären Zuschnitt gibt es auch Masken, womit verschiedenen Bereichen eines Elements verschiedene Transparenzwerte zugeordnet werden können.

Mittels des `<use>`-Elements können Klone von Elementen erzeugt werden, so dass mehrfach benötigte, komplexere Zeichnungen nur einmalig erstellt werden müssen, diese werden dann i. d. R. an die gewünschte Stelle verschoben. Ein Klon ist keine Kopie, Änderungen am Originalelement wirken sich auch auf vorher erstellte Klone aus.

Ein SVG-Dokument kann in einer eigenen Datei gespeichert sein, es ist aber auch möglich, es direkt innerhalb eines HTML-Dokuments einzubinden, indem das `<svg>`-Tag direkt in den HTML-Code eingefügt wird. Der Code bis zum schließenden `</svg>` wird dann als SVG interpretiert.

CSS *(Cascading Style Sheets)* ist eine Beschreibungssprache, mit der das Aussehen von HTML-, aber auch von SVG-Dokumenten spezifiziert werden kann. Dies geschieht durch Zuweisung von Werten an bestimmte Schlüsselwörter, beispielsweise `fill:red` oder `stroke-width:3px`. Für welche Elemente eine Definition gilt, wird durch verschiedene Selektoren festgelegt, dabei kann u. a. nach Elementnamen oder Klassennamen (durch einen vorangestellten Punkt) selektiert werden, auch Verschachtelungen sind möglich. Genauer definierte Selektionen überschreiben ggf. weniger spezifische.

Beispielsweise kann durch die Anweisung `text { fill: white; }` angegeben werden, dass alle Text-Elemente weiß dargestellt werden sollen, durch die (spezifischere, daher dominierende) Anweisung `.clock text { fill: yellow; }` werden Text-Elemente, die innerhalb eines Elements mit der Klasse `clock` liegen, gelb dargestellt.

3.2.3 JavaScript-Besonderheiten

Asynchronität Im Code müssen an mehreren Stellen Daten vom Backend geladen werden, z. B. Fahrplandaten. Da i. d. R. sämtlicher Code einer Webseite in einem Thread läuft [11], würde eine synchrone Abfrage diesen Thread bis zum Erhalt einer Antwort anhalten und die Seite würde nicht mehr auf Nutzereingaben reagieren.

Codebeispiel 1: Typische Codefolge zum Abruf von Daten vom Server (unter Nutzung der jQuery*-Hilfsfunktion `$.get()`)*

```
function setup () {
    $(document.body).append('<div id="loadingmsg">Lade Daten ...</div>');
    $.get('/data.json', (function(data) {
        $('#loadingmsg').remove();
        $(document.body).append(parse_response(data));
    }));
    // weiterer Code, der von der Abfrage unabhängig ist
}
```

Deshalb bietet JavaScript nur asynchrone Abfragen an[12]. Dazu wird beim Erstellen einer Abfrage eine Callback-Funktion mit angegeben. Sobald eine Antwort eingetroffen ist, wird diese Funktion aufgerufen und ihr die empfangenen Daten übergeben. Die aufrufende Funktion läuft sofort nach dem Absenden der Anfrage weiter, an dieser Stelle kann also nur Code ausgeführt werden, der die vom Server abgefragten Daten nicht benötigt.

Ich nutze die Abfragen etwa in der im Codebeispiel 1 dargestellten Art. Durch die Statusmeldung, die im Normalfall nur Sekundenbruchteile sichtbar ist, findet bei einer längeren Antwortzeit oder einem Fehler automatisch eine Information des_der Nutzer_in statt, so dass eine explizite Fehlerbehandlung nicht zwingend notwendig ist.

Klassen JavaScript hat formell gesehen keine Klassen, stellt aber mit dem Konzept der *Objects* und *Prototypes* ein Sprachmittel zur Verfügung, das sehr ähnlich genutzt werden kann [13]. Ein *Object* ist zunächst nur ein Datentyp, der Werte unter einem Schlüssel speichert[13], aber in Verbindung mit *Prototypes* wie Klassen genutzt werden kann, wenn auch mit einer gewöhnungsbedürftigen Syntax.

Codebeispiel 2: Illustration der Definition und Nutzung einer „Klasse“ in JavaScript

```
var Circle = function(radius) {
    this.radius = radius;
}
Circle.prototype = {
    get_perimeter: function () {
        return this.radius * 2 * Math.PI;
    }
}

var unitcircle = new Circle(1);
unitcircle.get_perimeter() // == 6.28...
```

[12] Tatsächlich sind synchrone Abfragen momentan möglich, wegen der genannten Nachteile für die grafische Oberfläche sind sie allerdings *deprecated* und sollen zukünftig entfallen [12].

[13] In anderen Programmiersprachen z. B. bekannt als assoziatives Array, Map, oder Dictionary.

Eine „Klasse“ in JavaScript ist prinzipiell eine Funktion (die als Initialisierungsfunktion dient) in Verbindung mit einem *Prototype*, einem *Object*, das weitere Attribute und Methoden bereitstellen kann. Der Prototype entspricht also etwa der Klasse, zusätzlich kann auch er wiederum einen Prototype haben, wodurch eine Art Klassenhierarchie implementiert werden kann. Um ein neues „Objekt“ zu erzeugen, wird die Initialisierungsfunktion mit dem Schlüsselwort `new` aufgerufen, dabei erstellt JavaScript intern ein neues, leeres *Object* und weist ihm als `prototype` den der Initialisierungsfunktion zugeordneten zu. Dann wird die Initialisierungsfunktion derart aufgerufen, dass `this` auf das neu erstellte *Object* zeigt. Im Codebeispiel 2 ist eine Nutzung dieser „Klassen“[14] beispielhaft dargestellt.

3.2.4 Strukturierung

Das ZWL-Diagramm besteht aus verschiedenen grafischen Elementen, wie den Bahnhofskürzeln, der Zeitleiste, den einzelnen Zuglinien und ihren Beschriftungen. Es liegt nahe, diese bereits existierende Struktur auch zur Strukturierung des Codes zu benutzen.

Ich habe diese „Klassen“ für die oben erwähnte Strukturierung genutzt. Ich werde im Folgenden diese Aufteilung erläutern und begründen sowie die Arbeitsweise des Frontends anhand dieser Unterteilung erläutern, d. h. die Aufgaben der einzelnen Teile vorstellen und erklären, wie ich sie umgesetzt habe und wie die Teile zusammenarbeiten. In Abbildung 7 ist der Zusammenhang der Klassen grafisch dargestellt.

Gemeinsamkeiten der verschiedenen Klassen. Den meisten Klassen ist gemein, dass ihnen ein `<g>`-Gruppierungselement im SVG-Baum zugeordnet ist; die Gliederung des SVG-Dokuments entspricht also etwa derer des Codes. Des weiteren ist bei den meisten Klassen der Initialisierungscode auf mehrere Stellen verteilt: In der Initialisierungsfunktion werden die SVG-Elemente nur erzeugt, und erst in einer weiteren Methode, `redraw`, positioniert. So kann letztere wiederverwendet werden, z. B. um bei einer Änderung der Fenstergröße die Anzeige anzupassen, so dass sie immer den verfügbaren Raum ausnutzt, oder um nach einer Aktualisierung der Fahrplandaten die Zuglinien neu zu zeichnen.

Ein weiteres Problem wird in allen Klassen gleich gelöst: Würden alle SVG-Elemente absolut (d. h. relativ zum Browserfenster) positioniert, müssten beim Verschieben der Zeitleiste viele Male pro Sekunde sämtliche Elemente neu positioniert werden. Mit SVG ist es jedoch möglich, Elementen (und auch Gruppen) Transformationen zuzuweisen, so können z. B. alle Elemente einer Gruppe „nachträglich“ (d. h. ohne Änderung der eigentlichen Koordinaten der einzelnen Elemente) verschoben werden. Es werden daher alle Elemente relativ zu einem (frei gewählten) Zeit-Nullpunkt positioniert, und die gesamte Gruppe wird durch eine Transformation so nach oben verschoben, dass der gewünschte Zeitraum im Browserfenster zu sehen ist. Außerdem

[14] Der Einfachheit halber verwende ich im Folgenden die Begriffe *Klasse* und *Instanz*.

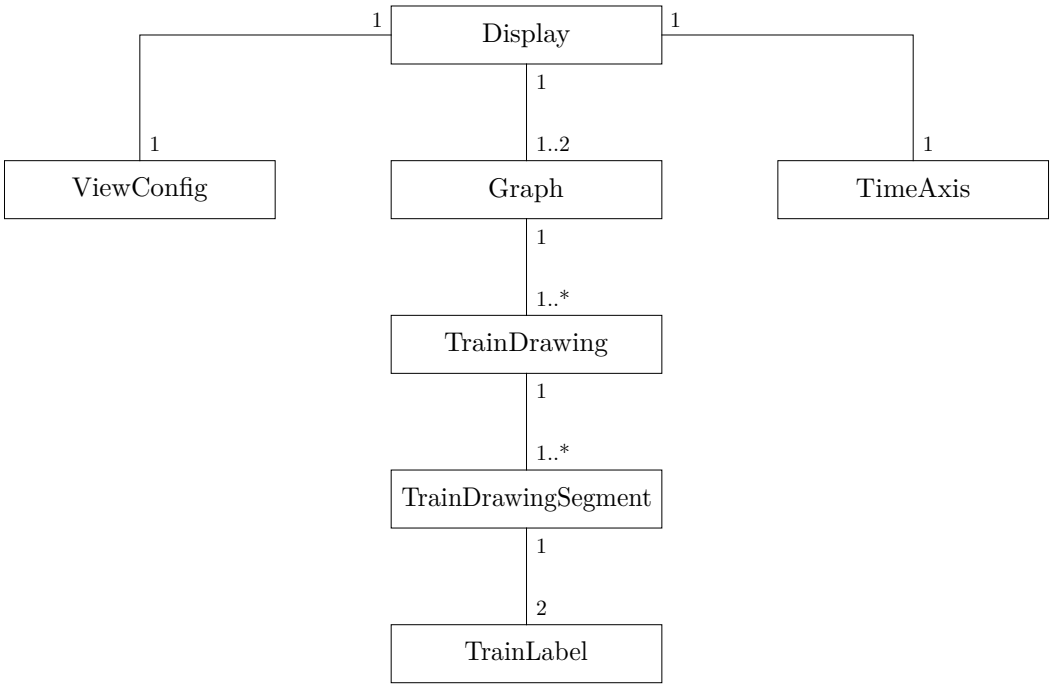

Abbildung 7: UML-Diagramm der Klassen im Frontend

werden die Zuglinien so beschnitten[15], dass der außerhalb des anzuzeigenden Bereichs liegende Teil nicht sichtbar ist. So ist der Code zur Positionierung der Zuglinien unabhängig vom aktuell gewählten Zeitraum und beim Ändern dieses Zeitraums muss nur eine Stelle (die Transformationskoordinaten) geändert werden.

Zwei Grundsätze gelten ebenfalls für jede der Klassen (mit wenigen Ausnahmen): Die von einer Klasse erstellten SVG-Elemente werden auch nur von dieser Klasse manipuliert. Außerdem kommuniziert ein Objekt nur mit Objekten, die es erstellt hat, bzw. von denen es erstellt wurde.

`ZWL` ist ein *Object*, das nur der Strukturierung dient. Da JavaScript keine Namespaces unterstützt, ist dies ein üblicher Weg, diese zu emulieren. Alle definierten Klassen werden als Attribute von `ZWL` definiert, ihr Name lautet also generell `ZWL.Klassenname`.

`ZWL.Display` ist die Hauptklasse. Sie wird als erste initialisiert, verwaltet alle übrigen Klassen und erzeugt das SVG-Dokument, jedoch keines der grafischen Elemente, da diese alle einer der anderen Klassen zugeordnet sind. `Display` speichert auch die aktuelle Uhrzeit und den aktuell angezeigten Zeitraum[16]. Zur Umsetzung der oben

[15] Mittels des SVG-Attributs `clip-path` kann einem Element ein Ausschneidepfad zugewiesen werden. Der außerhalb dieses Pfades liegende Teil des Elements wird dann nicht angezeigt.

[16] Was nicht heißt, dass diese Daten den anderen Klassen nicht zur Verfügung stünden, sie werden lediglich an nur einer Stelle gespeichert und aktualisiert, da die zeitbezogenen Daten in allen Graphen gleich sind.

beschriebenen zeitrelativen Positionierung stellt `Display` eine Methode `time2y` zur Verfügung, mit der die anderen Klassen ihre Grafiken entsprechend positionieren können.

`ZWL.Graph` setzt die oben beschriebene Trennung des Codes zur Erzeugung der Graphen um und ermöglicht so eine Anzeige von mehreren Strecken auf einem Bildschirm. In diesem Fall gibt es mehrere Instanzen dieser Klasse (siehe z. B. Abbildung 8, dort gibt es zwei `Graph`-Instanzen).

`Graph` wird initialisiert unter Angabe des Bezeichners der anzuzeigenden Strecke und ggf. der Anfangs- und Endpunkte im Falle der Anzeige nur eines Streckenabschnitts, und lädt dann eine Beschreibung der Strecke vom Server. Die Beschreibung enthält alle auf der Strecke liegenden Bahnhöfe, Signale, Abzweigungen etc. und ihre Position auf der Strecke[17], ihr Kürzel und Ähnliches. Diese Daten werden gespeichert und u. a. von der Methode `pos2x` genutzt, diese gibt zu einem Betriebsstellenbezeichner die entsprechende x-Koordinate zurück.

Außerdem können nun die Kürzel der Betriebsstellen sowie die senkrechten Linien im Diagramm gezeichnet werden. Danach[18] werden die Zugdaten abgerufen. Das sind momentan nur die Fahrplandaten, in Zukunft soll das Backend jedoch in der selben Abfrage Streckensperrungen und Kommentare mitliefern. Um keine Daten zu Zügen zu erhalten, die gar nicht im momentan sichtbaren Ausschnitt erscheinen würden, wird bei der Anfrage nicht nur der Bezeichner der angezeigten Strecke, sondern auch Anfang und Ende des angezeigten Streckenabschnitts und der angezeigte Zeitraum mit übermittelt. Das Backend gibt dann nur Züge zurück, die innerhalb dieses Zeitraums auf der angezeigten Strecke fahren[19].

Diese Beschränkung führt dazu, dass beim Verschieben des Anzeigezeitraums kurzzeitig nicht alle Züge angezeigt werden, die im neuen Zeitraum fahren, bis diese Züge nachgeladen wurden (siehe auch Abbildung 9).

Diese Abfrage wird sehr oft ausgeführt, und zwar nach einer Änderung des ausgewählten Zeitabschnitts, außerdem bei der regelmäßigen Aktualisierung der Anzeige.

Nach jeder Abfrage der Fahrplandaten werden die Zuglinien neu gezeichnet, dies ist im Abschnitt zu `ZWL.TrainDrawing` (s. u.) beschrieben.

Eine Anforderung ist, dass es möglich sein soll, nur einen Streckenabschnitt anzuzeigen. Dies ist durch Angabe von Positionswerten in der URL möglich (eine detaillierte Beschreibung dazu findet sich weiter unten im Abschnitt zu `ZWL.ViewConfig`). Um den Code, der die Zuglinien zeichnet, simpel zu halten, wird hierbei der selbe Mechanismus

[17] Die Position ist nicht anhand von Streckenkilometern definiert, sondern als Wert zwischen 0 und 1, was einerseits die Berechnung der Bildschirmposition erleichtert und andererseits die Streckenbeschreibung flexibler macht, so kann z. B. die freie Strecke etwas gestaucht und dafür Bahnhofsbereiche detaillierter angezeigt werden.

[18] Eine eigentlich mögliche Parallelisierung wurde mit Rücksicht auf die Codekomplexität unterlassen: Der Abruf der Zugdaten geschieht mehrmals pro Minute, die Parallelisierung hätte jedoch nur beim Start der Anwendung einen kleinen Vorteil gebracht.

[19] Die Aufbereitung der Daten im Backend wird weiter unten beschrieben.

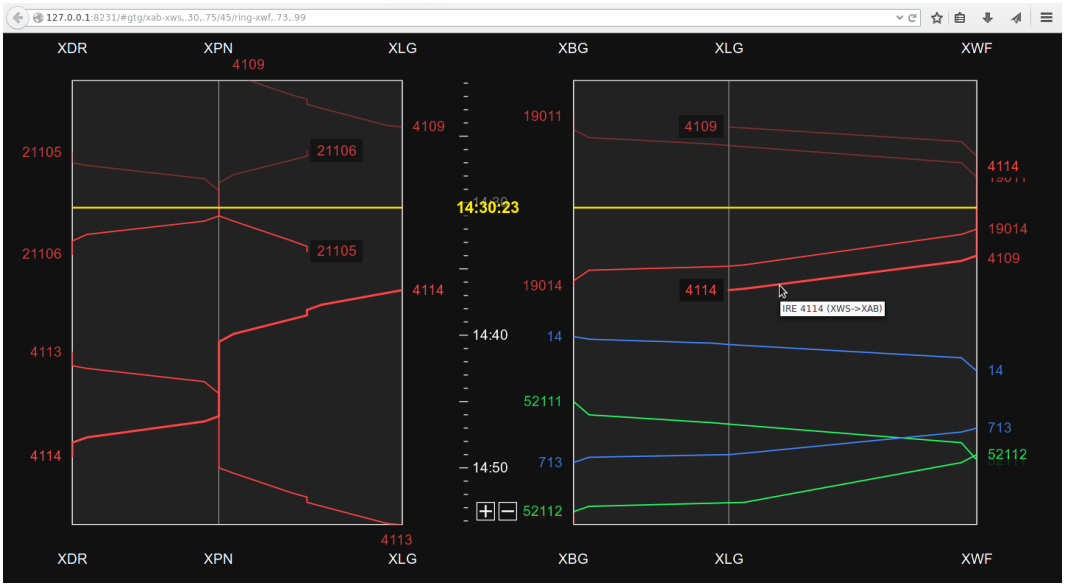

Abbildung 8: Ansicht des ZWL-Diagramms mit zwei angezeigten Strecken und der Zeitleiste in der Mitte. Links ist die eingleisige Strecke dargestellt, die in XLG *auf den Ring (rechts) trifft; es ist leicht zu sehen, dass Zugkreuzungen nur in Bahnhöfen stattfinden. Der Zug 4114 wechselt in* XLG *auf die links angezeigte Strecke. Beim Überfahren seiner Zuglinie mit der Maus wird diese Linie auch im linken Graph hervorgehoben, indem sie etwas breiter dargestellt wird.*

angewandt wie oben beschrieben: Die Zuglinien werden immer auf der gesamten Strecke gezeichnet, jedoch mit einer SVG-Transformation an die gewünschte Stelle verschoben und auf den gewählten Ausschnitt zugeschnitten.

Um in der Vergangenheit liegende Zuglinien blasser darzustellen, wird ebenfalls ein SVG-Mittel benutzt, nämlich die dem Ausschneiden verwandte *Maske* (SVG-Attribut `mask`). Hiermit kann einem Element an verschiedenen Stellen verschiedene Grade von Transparenz zugewiesen werden. So können die Zuglinien im Bereich oberhalb der aktuellen Zeit teilweise transparent und damit blasser gemacht werden. Indem diese Funktionalität rein durch SVG implementiert wird, kann eine aufwendige Trennung der Linien am aktuellen Zeitpunkt in eine blasse und eine normale Linie vermieden werden.

`ZWL.TimeAxis` verwaltet die Zeitachse. Diese dient wie oben beschrieben auch zum Verändern des angezeigten Zeitabschnitts. Sie besteht aus kleinen Strichen zu jeder Minute, zusätzlich wird alle zehn Minuten die Uhrzeit ausgeschrieben, außerdem gibt es noch Buttons, mit denen der zeitliche „Zoom“, also die Anzahl der Pixel pro Minute, stufenweise verstellt werden kann.

Für die Veränderung der Zeit wurde das svg.js-Plugin *svg.draggable.js* benutzt. Es abstrahiert das Verschieben (*drag&drop*) der Zeitachse, so dass nur drei Funktionen definiert werden müssen, die zu Beginn des Ziehens, bei jedem Schritt (`dragmove`) bzw. nach Loslassen der Maustaste (`dragend`) aufgerufen werden. Außerdem kön-

nen Bedingungen für die Bewegung definiert werden, so wird verhindert, dass die Zeitachse horizontal verschoben werden kann. Durch `dragmove` wird die Bewegung an die Klasse `Display` weitergegeben, die sie wiederrum an die `Graph`-Instanzen weitergibt[20], so dass diese sich mitbewegen. Am Ende wird durch `dragend` die Methode `Display.redraw` aufgerufen, diese veranlasst z. B. ein Neuladen der Fahrplandaten aller `Graph`-Instanzen.

Im Gegensatz zu den Zuglinien wird die Zeitachse nicht nur im sichtbaren Bereich gezeichnet, sondern auch jeweils eine Bildschirmhöhe nach oben und unten, da man sonst beim Ziehen der Zeitleiste nicht sehen würde, in welchem Zeitraum man sich befindet.

Auch Teil der Zeitleiste ist die Anzeige der aktuellen Simulations-Uhrzeit. Sie wird immer an der Stelle in der Zeitachse eingeblendet, die der aktuellen Uhrzeit entspricht, wenn diese im sichtbaren Bereich ist. Ansonsten wird sie, um sicher zu stellen, dass die Uhrzeit immer zu sehen ist, am oberen bzw. unteren Rand der Zeitleiste positioniert.

`ZWL.TrainInfo` und `ZWL.LineConfiguration` dienen der Speicherung der vom Server zurückgesandten Daten. Dafür könnte auch ein normales JavaScript-Object benutzt werden, jedoch können mittels der Klassen Hilfsfunktionen angeboten werden, beispielsweise im Falle von `LineConfiguration` eine Funktion, die eine Betriebsstelle der Strecke anhand ihres Kürzels zurück gibt (diese sind eigentlich in einer Liste gespeichert, da ihre Ordnung relevant ist).

`ZWL.ViewConfig.` Die Möglichkeit, einen oder mehrere Graphen gleichzeitig anzuzeigen, macht die Positionierung der einzelnen Diagramm-Elemente wesentlich komplexer, daher wurde diese aus der Klasse `Display` herausgelöst und hier implementiert.

`ViewConfig` wird initialisiert mit der Ansichtskonfiguration, die der_die Nutzer_in über das *URL-Fragment* (den Teil der URL nach dem `#`) übergeben kann[21]. Diese wird geparst und eine entsprechende Anzahl von `Graph`-Instanzen erzeugt und der aufrufenden `Display`-Instanz zugewiesen. Danach (sowie bei jeder Änderung der Fenstergröße) berechnet `ViewConfig` passende Positionen und Größen für die Diagramm-Elemente (Graph(en) und Zeitachse) und ruft deren `redraw`-Methode auf, so dass sie ihrerseits die zu ihnen gehörenden SVG-Elemente entsprechend positionieren.

Durch die Ansichtskonfiguration kann angegeben werden, welche und wie viele Strecken angezeigt werden sollen, sowie an welcher Stelle die Zeitachse erscheinen soll. Sie besteht aus einem Schlüsselwort, das eine Positionierungsmethode definiert, einem Schrägstrich und einer von dieser Methode abhängigen Zahl von Parametern, die wiederum voneinander durch Schrägstriche getrennt sind. Bisher sind folgende Positionierungsmethoden implementiert, weitere können aber leicht hinzugefügt

[20] entsprechend des Prinzips, dass einander unbekannte Klassen nicht direkt miteinander kommunizieren

[21] Es soll zukünftig für die Stellwerke jeweils eine passende Konfiguration vorgegeben werden und diese alle an geeigneter Stelle verlinkt werden, so dass die Nutzer_innen die Syntax nicht kennen müssen.

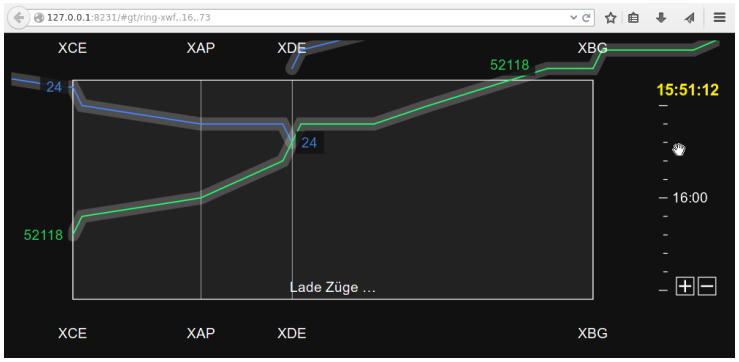

Abbildung 9: Screenshot mit Manipulationen zur Demonstration einiger Interna. Es wurde gerade der angezeigte Zeitabschnitt verschoben, die neuen Zuglinien werden gerade geladen. Es werden also nur die Züge angezeigt, die schon vorher geladen wurden, weil sie auch im vorher angezeigten Zeitabschnitt sichtbar waren. Des Weiteren wurde der Zuschnitt der Zuglinien deaktiviert und die „doppelte" Zuglinie sichtbar gemacht. (Um den Screenshot erstellen zu können, habe ich die Antwort des Servers künstlich verzögert.)

werden:

- eine Strecke: Schlüsselwörter `gt` und `tg` (die Position des `t` bestimmt, ob die Zeitachse links oder rechts des Graphs ist). Als Parameter folgt nur der Bezeichner der Strecke. Beispiel: `#gt/ring-xwf`
- zwei Strecken: `tgg`, `gtg` und `ggt`, als Parameter folgen die Bezeichner der beiden Strecken, dazwischen eine Zahl, die die relative Breite (in Prozent) des linken Graphen angibt. Beispiel: `#gtg/xab-xws/30/ring-xwf` (der linke Graph nimmt also etwa 30% der Fensterbreite ein, Zeitachse in der Mitte)

Zusätzlich zum Bezeichner der anzuzeigenden Strecke kann ein Ausschnitt der Strecke gewählt werden, indem an ihn, mit Komma getrennt, zwei Zahlen angehängt werden. Diese definieren Anfang und Ende des Streckenabschnitts. Wie oben beschrieben, hat jede Betriebsstelle intern eine Position zwischen 0 und 1, diese Zahlen müssen also auch zwischen 0 und 1 liegen; sind sie nicht gegeben, wird die gesamte Strecke angezeigt (entsprechend der Angabe `0,1`). Die Ansichtskonfiguration in Abbildung 8 (Seite 21, `#gtg/xab-xws,.30,.75/45/ring-xwf,.73,.99`) bedeutet also: Die linke Strecke soll etwas schmäler als die andere sein, von ihr soll nur der mittlere Teil angezeigt werden, von der rechten nur etwa ein Viertel.

Diese Syntax muss jedoch von den gewöhnlichen Nutzer_innen nicht beherrscht werden, es ist geplant, für die verschiedenen Stellwerke jeweils eine passende Ansicht vorzukonfigurieren und zu verlinken.

`ZWL.TrainDrawing` und `ZWL.TrainDrawingSegment` sind für das Zeichnen der Zuglinien zuständig. Jede `Graph`-Instanz erstellt eine Instanz von `TrainDrawing` für jeden Zug, zu dem sie vom Backend Fahrplandaten erhalten hat (d. h. für jeden Zug, der im gewählten Zeitraum auf der gewählten Strecke fährt). Es ist jedoch möglich, dass ein Zug mehrfach in einem Graphen auftaucht, insbesondere beim Ring, vgl. Zug 52111

in Abbildung 6 (Seite 15). In diesem Fall besteht der vom Server zurückgegebene Fahrplan für diesen Zug aus mehreren Segmenten. In jedem Fall erstellt `TrainDrawing` eine Instanz von `TrainDrawingSegment` pro Fahrplansegment, letztere Klasse ist für das Zeichnen der Zuglinien verantwortlich.

Von `Graph` werden zwei SVG-Gruppen erstellt, jeweils eine für die Zugnummern und die Zuglinien. Diese Trennung ist nötig, damit die Zugnummern aller Züge immer oberhalb aller Zuglinien dargestellt werden[22].

Innerhalb dieser beiden Gruppen wird von `TrainDrawing` für jeden Zug jeweils eine Untergruppe erstellt, in die wiederum die `TrainDrawingSegment`-Instanzen die von ihnen erzeugten Zuglinien bzw. Zugnummern einfügen. Letzteren beiden Gruppen werden jeweils folgende Eigenschaften zugewiesen:

- im Attribut `title` werden Informationen zum Zug wie Zuggattung, Zugnummer und Start/Ziel der Fahrt gesetzt. Beim Überfahren einer der Zuglinien oder Beschriftungen (Zugnummer) wird dieser `title` vom Browser angezeigt.
- dem Attribut `class` wird ein von der Verkehrsart abhängiger Wert zugewiesen, z. B. bei einer Regionalbahn `category_nv` (Nahverkehr). So kann der Zuglinie und -nummer flexibel per CSS eine Farbe zugewiesen werden, die auch vom Theme[23] abhängig sein kann, ohne dass der JavaScript-Code vom Theme abhängig ist.
- wenn die Gruppe mit der Maus überfahren wird, wird die gesamte Gruppe in den Vordergrund geholt. Dies dient dazu, dass ggf. von einer anderen Zugnummer überdeckte Zugnummern sichtbar gemacht werden (vgl. Züge 52111/52112 rechts unten in Abbildung 6, Seite 15). Außerdem wird die Gruppe als ausgewählt markiert, so dass sie per CSS hervorgehoben werden kann. Dies wird über die JavaScript-Events `mouseenter` und `mouseleave` erreicht; dabei werden diese Änderungen nicht direkt an der Gruppe durchgeführt, sondern eine Methode von `Display` aufgerufen, die die genannten Änderungen in allen `Graph`-Instanzen veranlasst, so dass der Zug überall hervorgehoben wird (vgl. Abbildung 8, Seite 21)

`TrainDrawingSegment` analysiert das ihm übergebene Fahrplansegment und erstellt daraus eine Liste von zu zeichnenden Elementen, das sind abwechselnd Stops an einer Betriebsstelle (mit Ankunfts- und Abfahrtszeit), also senkrechte Striche, und Fahrten zwischen zwei Betriebsstellen. In einem zweiten Schritt wird für jedes dieser Elemente eine SVG-Element (`<line>`) erzeugt[24]. SVG unterstützt zwar über das Element `<polyline>` auch Polygonzüge, womit die gesamte Zuglinie in einem Element gezeichnet werden könnte. Damit wäre es aber nicht möglich, Besonderheiten, die nur auf einem Teil der Fahrt gelten (z. B. Fahrt im Gegengleis), hervorzuheben.

Um die Selektion der relativ dünnen Zuglinien mit der Maus zu vereinfachen, wird zu-

[22] Wie bereits erwähnt, überdecken in SVG später definierte Elemente frühere.

[23] noch nicht implementiert, aber vorbereitet

[24] Wobei bei Betriebsstellen, an denen nicht gehalten wird, das entsprechende Element übersprungen wird, da die Linie eine Länge von 0 hätte.

sätzlich zu der sichtbaren Linie eine zweite Zuglinie mit identischem Verlauf gezeichnet, die wesentlich breiter ist und eine Transparenz von 100% hat.

`ZWL.TrainLabel` ist für die Anzeige der Zugnummern zuständig. Von jeder `TrainDrawingSegment`-Instanz werden zwei `TrainLabel` erzeugt, die am Beginn und Ende der Zuglinie eingeblendet werden; falls diese außerhalb des sichtbaren Bereichs beginnt bzw. endet, erscheint die Zugnummer an dessen Rand, an der Stelle, an der die Linie das Diagramm verlässt.

Da Überlappungen der Zugnummern nicht ausgeschlossen werden können, wird zu jeder Zugnummer ein Rechteck erzeugt, das evtl. darunter liegende Zugnummern überdeckt[25]. Diese beiden Elemente sind in einer Gruppe zusammengefasst, die einmalig von `TrainDrawing` erstellt und außerhalb des sichtbaren Bereichs positioniert wird, die einzelnen `TrainLabel`-Instanzen erstellen lediglich einen Klon (`<use>`) und verschieben diesen an den gewünschten Ort.

Da sich die Stelle, an der die Linie den sichtbaren Bereich des Diagramms verlässt, beim Verschieben des angezeigen Zeitraums jederzeit verändern kann, muss diese Positionsberechnung während des Ziehens ständig neu durchgeführt werden (dazu wird die hierfür zuständige Funktion innerhalb des Ereignisses `dragmove` der Zeitachse aufgerufen).

3.3 Backend

3.3.1 zu den benutzten Bibliotheken

Flask ist ein Framework zur Entwicklung von Webapplikationen, das die typischen Aufgaben übernimmt, die für den Betrieb einer Webanwendung nötig sind, z. B. die Kommunikation mit dem Webserver[26], die Verwaltung und gegenseitige Isolierung von Threads zur gleichzeitigen Beantwortung mehrerer Anfragen sowie Parsen der Anfragedaten. Außerdem bietet es viel Hilfsfunktionalität, ich nutze beispielsweise das Einlesen von Konfigurationsdateien, Abbilden von URLs auf Funktionsaufrufe und die automatische Erzeugung von JSON-Ausgaben (siehe Codebeispiel 3).

SQLAlchemy ist einerseits eine Bibliothek, die es erlaubt, SQL-Abfragen abstrakt in Python-Code zu formulieren, und andererseits ein Object Relational Mapper, d. h. es bildet Datenbanktabellen auf Python-Klassen ab (eine solche Klasse nennt man dann *Model*). Dabei werden, wo immer möglich, Datenbanktypen in entsprechende Python-Typen abgebildet, so dass z. B. bei dem Model im Codebeispiel 4 dem Attribut `arr_plan` ein Objekt vom Typ `datetime.time`[27] zugewiesen werden kann. Die Models

[25] Wie oben beschrieben, kann durch Überfahren der Zuglinie mit der Maus dessen Zugnummer in den Vordergrund geholt werden.

[26] Für die Entwicklung hat Flask auch einen einfachen Webserver eingebaut, im Produktivbetrieb sollte es im Hintergrund eines echten Webservers betrieben werden.

[27] üblicher Datentyp für Zeitangaben, aus der Python-Standardbibliothek

Codebeispiel 3: Vereinfachtes Beispiel zur von Flask gebotenen Abstraktion. Wenn eine auf den angegebenen String passende URL aufgerufen wird, ruft Flask die Funktion `graphdata` *auf, wobei* `linename` *der in der URL stehende Text zugewiesen ist. Die Hilfsfunktion* `get_graphdata` *gibt Python-Datentypen zurück (*`list` *und* `dict`*),* `jsonify` *wandelt diese in JSON um. Die Funktionen* `app.route`*,* `jsonify` *und* `abort` *stammen von Flask.*

```
@app.route('/graphdata/<linename>.json')
def graphdata(linename):
    if linename ins all_lines:
        result = get_graphdata(linename)
    else:
        abort(404)
    return jsonify({
        'graphdata': result,
    })
```

Codebeispiel 4: Definition eines Model *mit SQLAlchemy. Wie man sieht, können die Namen der Attribute von denen der Tabellenspalten abweichen. Das Attribut* `train` *liefert direkt das zum Fahrplaneintrag gehörende* `Train`*-Objekt (mit der per* `train_id` *angegebenen* `id`*).*

```
class Train(db.Model):
    # ...
class TimetableEntry(db.Model):
    __tablename__ = 'fahrplan_sessionfahrplan'
    id = db.Column(db.Integer, primary_key=True)
    train_id = db.Column('zug_id', db.Integer, db.ForeignKey(Train.id))
    loc = db.Column('betriebsstelle', db.String(10))
    arr_plan = db.Column('ankunft_plan', db.Time)
    dep_plan = db.Column('abfahrt_plan', db.Time)
    train = db.relationship(Train)
```

Codebeispiel 5: Beispiele für mittels SQLAlchemy formulierte Datenbankanfragen

```
# alle Fahrplaneinträge an einer bestimmten Betriebsstelle
TimetableEntry.query.filter(TimetableEntry.loc == 'XWF')

# alle Fahrplaneinträge eines bestimmten Zuges nach 15 Uhr
# 'trainobj' ist ein Objekt vom Typ Train (s.o.)
TimetableEntry.query.filter(
    (TimetableEntry.dep_plan > datetime.time(15,0)) &
    (TimetableEntry.train == trainobj)
)
```

können wie gewöhnliche Klassen benutzt werden, z. B. kann man eigene Methoden definieren. Auch Abfragen können mit Hilfe dieser Models durchgeführt werden, die Syntax dafür integriert sich sehr natürlich in Python-Code (siehe Codebeispiel 5).

3.3.2 Umsetzung

Aufbereitung der Fahrplandaten Die wichtigste Funktion des Backends ist die Aufbereitung der Fahrplandaten für das Frontend. Dabei werden die Informationen aus den Tabellen `fahrplan_sessionzuege`, `fahrplan_sessionfahrplan` und `zuege_zuggattungen` extrahiert und in einem Eintrag pro Zug zusammengefasst. Dieser Eintrag enthält allgemeine Informationen zum Zug (Zugnummer, Zuggattung, Verkehrsart) und zum Fahrtverlauf (Start- und Zielbahnhof, ggf. Zugnummer, unter welcher der Zug am Zielbahnhof weiterfährt bzw. vor dem Startbahnhof fuhr[28]) sowie den Fahrplan. Letzterer besteht, wie im Abschnitt zum Frontend beschrieben, aus mehreren Segmenten, wenn der Zug die Strecke mehrfach befährt[29], ansonsten aus einem. Jedes dieser Segmente enthält die befahrenen Betriebsstellen (in der Reihenfolge, in der sie befahren werden) mit der zugehörigen Ankunfts- und Abfahrtszeit sowie die Angabe, in welcher Richtung die Strecke befahren wird[30].

Diese Aufteilung der Fahrplandaten ist nötig, da bei einer direkten Nutzung der Fahrplandaten für das Zeichnen der Zuglinien Probleme entstehen, z. B. würde bei einem Zug, der die Strecke verlässt und sie vom anderen Ende wieder befährt, eine Linie quer über den Graph gezeichnet werden, auch eine Anzeige der Zugnummer an den Stellen, an denen er den Graph verlässt, wäre so nur schwer zu realisieren. Die Aufteilung könnte zwar im Browser geschehen, allerdings ist das aufgrund des kleineren Funktionsumfangs von JavaScript dort schwieriger möglich, auch wäre es bei serverseitiger Implementierung möglich, die aufbereiteten Daten zu cachen, falls später die Serverlast zu groß wird.

Der Algorithmus zur Aufteilung ist im Abschnitt 3.4.2 beschrieben.

Die Streckenkonfiguration Für die Anzeige des Streckengraphen im Frontend und die o. g. Aufbereitung der Fahrplandaten ist eine Auflistung der an der Strecke liegenden Betriebsstellen nötig, diese wird im Code Streckenkonfiguration genannt und in der Klasse `LineConfiguration` implementiert.

Die Streckenkonfiguration ist nicht eindeutig einer Strecke zugeordnet: Beispielsweise gibt es zwei Streckenkonfigurationen, die den Ring abdecken: Neben der in Abbildung 6 (Seite 15) gezeigten, bei der der Ring im Bahnhof *XWF* „aufgetrennt“ ist, existiert eine zweite, bei der dieser Bahnhof in der Mitte ist, da mit ersterer keine für dort arbeitende Fahrdienstleiter_innen sinnvolle Ansicht möglich ist.

Ein weiterer Unterschied zwischen Strecke und Streckenkonfiguration ist, dass in letzterer Betriebsstellen mehrfach vorkommen können (z. B. XWF beim Ring). Daher werden die Betriebsstellenkürzel zur eindeutigen Identifikation um eine Zahl ergänzt (z. B. `XWF#1`). Die aufbereiteten Fahrplandaten enthalten die Betriebsstellen in dieser

[28] Diese Übergänge werden im ZWL-Diagramm noch nicht dargestellt, dies ist aber geplant. Aufgrund der Ringstruktur und der Bedingung, dass jede Betriebsstelle von einer Zugnummer nur ein Mal befahren werden darf, finden im EBuEf sehr oft Zugnummernwechsel statt, bei denen der Zug nach kurzem Aufenthalt in die selbe Richtung weiterfährt.

[29] In der Praxis kommt das nur am Ring vor, theoretisch sind aber weitere Möglichkeiten vorstellbar.

[30] Diese Information wird vom Algorithmus, der die Zugnummern positioniert, benötigt.

Form, so dass sie vom Frontend eindeutig einem Ort im Graphen zugeordnet werden können.

3.4 Algorithmen

Ein Großteil des Codes insbesondere im Frontend führt nur kleine, unzusammenhängende Aufgaben aus. Dieser Code lässt sich schwer losgelöst betrachten, um eine algorithmische Betrachtung anzustellen. Es gibt jedoch einige wenige Stellen, an denen ein größeres Problem gelöst wird, wo eine solche Analyse möglich ist, dies werde ich im Folgenden durchführen.

3.4.1 Prognose

Der zugehörige Quellcode findet sich in der Datei `backend/zwl/predict.py`.

Die Prognose ist innerhalb des Backends implementiert, also in Python und nutzt ebenfalls SQLAlchemy, die Model-Definitionen und einige Hilfsfunktionen mit, ist ansonsten aber relativ unabhängig. Sie interagiert nur über die Datenbank mit den anderen Teilen des Backends, indem sie die Plan-, Soll- und Ist-Fahrplandaten ausliest und prognostizierte Fahrzeiten in die Prognose-Spalte einträgt. Geplant ist, die Prognose regelmäßig (etwa alle 15 Sekunden) laufen zu lassen, so dass im Frontend immer aktuelle Prognosedaten zur Verfügung stehen.

Eingabe des Algorithmus sind bisher nur die Fahrplandaten. Damit ist das Verhalten auf der freien Strecke schon gut prognostizierbar. Zukünftig ist auch eine Einbeziehung von Daten zu den möglichen Fahrstraßen geplant[31], um auch innerhalb von Bahnhöfen gute Prognosen bereitstellen zu können (z. B. im Falle von sich kreuzenden Fahrten im Bahnhof, was bisher nicht als Konflikt erkannt wird).

Als Eingabe sind alle drei Gruppen von Fahrplandaten[32] nötig: Die Soll-Fahrplandaten geben den Tages-Fahrplan vor; am Anfang einer Sitzung, wenn noch keine Ist-Daten vorliegen, entsprechen die Prognosedaten den Soll-Daten. Wenn der Betrieb begonnen hat und die ersten Ist-Daten vorliegen, werden auch diese mit in die Prognose einbezogen, indem jeweils der spätere der beiden Werte angenommen wird. Die Plan-Daten schließlich sind nötig, um die planmäßige Fahrzeit zwischen zwei Betriebsstellen herauszufinden.

Die Berechnung der Prognosezeiten erfolgt in zeitlich aufsteigender Reihenfolge, d. h. der Algorithmus arbeitet sich nach und nach weiter in die Zukunft vor. Dabei werden auch die weiter zurückliegenden Prognosezeiten benutzt, sie sind jedoch keine Eingabedaten des Algorithmus: Die Prognosedaten eines vorherigen Laufs des Algorithmus werden nicht ausgelesen, da sie auf evtl. veralteten Ist-Zeiten beruhen.

[31] Diese existieren bereits weitgehend, da sie für die Anzeige der elektronischen Stellwerke gebraucht werden.

[32] Zu den Gruppen siehe die Erläuterungen zu `fahrplan_sessionfahrplan` auf Seite 9.

Ausgabe des Algorithmus sind die berechneten Prognosezeiten, die anschließend in die Datenbank eingetragen werden (dabei werden ggf. die eines älteren Prognoselaufs überschrieben).

Die Grundidee des Algorithmus ist eine Simulation des Betriebsablaufs. Die Züge melden in zeitlicher Reihenfolge ihre Fahrtwünsche an, die dafür benötigten Gleisabschnitte werden als belegt markiert. Wird dabei festgestellt, dass einer der benötigten Gleisabschnitte bereits belegt ist, muss der Zug warten, bis der Gleisabschnitt wieder freigegeben wurde.

Basis der Implementierung ist die Klasse `Manager`. Sie verwaltet die Streckenelemente und die Züge. Die Züge sind repräsentiert durch die Klasse `Journey`[33], von der pro Zug eine Instanz erstellt wird. Zur Anmeldung einses Fahrtwunschs senden die `Journey`-Objekte eine Instanz einer Unterklasse von `Action`, die vom `Manager` mit einer Instanz einer Unterklasse von `Response` beantwortet wird. Die einzelnen Klassen werden im Folgenden vorgestellt, davor erläutere ich noch ein verwendetes Sprachmittel.

Koroutinen Python bietet mit dem `yield`-Ausdruck die Möglichkeit, mittels Funktionen sogenannte Generatoren zu implementieren. Sobald der Interpreter auf einen solchen Ausdruck stößt, unterbricht er die Ausführung der Funktion, anstelle eines Rückgabewerts wird ein spezielles Objekt zurückgegeben, der Generator. Dieser be-

Codebeispiel 6: Einfaches Beispiel eines Generators und einer Koroutine in Python

```
1  def zweierpotenzen(i):
2      while True:
3          i = i*2
4          yield i
5
6  generator = zweierpotenzen(1)
7  generator.next()  # -> 2
8  generator.next()  # -> 4
9  generator.next()  # -> 8
10
11
12 def summe():
13     sum = 0
14     while True:
15         val = (yield sum)
16         if val is not None:
17             sum += val
18
19 koroutine = summe()
20 koroutine.next()  # -> 0
21 koroutine.send(4) # -> 4
22 koroutine.send(3) # -> 7
```

[33] Der eigentlich passendere Name `Train` wird nicht verwendet, da bereits ein Datenbank-Model so heißt.

sitzt eine Methode `next`, bei deren Aufruf der `yield` übergebene Wert zurückgegeben wird und der Funktionscode weiter ausgeführt wird, bis zum nächsten `yield`-Ausdruck, wo die Ausführung wieder unterbrochen wird.

In Python 2.5 wurde diese Funktionalität um einen Rückkanal erweitert [14]. Seitdem ist es in Ergänzung zur Methode `next` auch möglich, mittels der Methode `send` Werte an den Generator zu senden. In diesem Fall hat der `yield`-Ausdruck einen Rückgabewert[34], der innerhalb des Generators benutzt werden kann. Mit diesem Mittel ist es möglich, Koroutinen zu implementieren. Im Codebeispiel 6 auf der vorherigen Seite findet sich je ein Beispiel zu einem einfachen Generator und einem solchen, der als Koroutine genutzt wird.

`Journey` repräsentiert eine Zugfahrt. Nach Initialisierung wird zunächst der Fahrplan des Zuges aus der Datenbank geladen und dort nach dem spätesten Fahrplaneintrag gesucht, zu dem Ist-Daten existieren, und dieser als aktuelle Position gespeichert.

Die Methode `run()` wird zu Beginn der Simulation von `Manager` aufgerufen. Sie ist eine Koroutine, der Manager erhält also als Rückgabewert einen Generator.

`run` gibt bei Aufruf der `next`- bzw. `send`-Methode dieses Generators ein `Ride`-Objekt zurück, das den nächsten Fahrtwunsch des Zuges angibt (Abfahrtszeit, Start- und Zielbetriebsstelle). Der `Manager` prüft diesen Wunsch und sendet mittels der `send`-Methode eine Antwort zurück, entweder eine Bestätigung, dass diese Fahrt so durchgeführt werden kann, falls die dafür benötigten Gleisabschnitte frei sind, ansonsten antwortet er mit einer Fehlermeldung und der Angabe, wann die Gleisabschnitte vorraussichtlich wieder frei sind. In letzterem Fall meldet `Journey` den selben Fahrtwunsch erneut, jedoch mit der angegebenen, späteren Uhrzeit. Wenn die Fahrt schließlich erfolgreich durchgeführt werden konnte, wird die erfolgreiche Uhrzeit als prognostizierte Abfahrtszeit an der Startbetriebsstelle gespeichert.

Anschließend wird die aktuelle Position auf die Zielbetriebsstelle gesetzt und berechnet, wann der Zug dort ankommen wird. Beim nächsten `send`-Aufruf gibt `Journey` ein Objekt vom Typ `Arrival` zurück, das diese Information enthält. Es dient dazu, die belegten Abschnitte wieder freizugeben. Da hierbei keine weiteren Gleisabschnitte belegt werden, ist diese Aktion immer erfolgreich. Die Zeit wird als prognostizierte Ankunftszeit gespeichert. Anschließend wird wieder ein `Ride`-Objekt für die Fahrt zur nächsten Betriebsstelle zurückgegeben, wie oben beschrieben, und so weiter, bis zum Ende der Fahrt.

Für die Berechnung der jeweiligen Abfahrts- und Ankunftszeiten werden alle vier Gruppen von Fahrplandaten benutzt. Anhand der Ist- und Prognose-Zeiten[35] wird der frühestmögliche Zeitpunkt berechnet, zu dem der Zug weiterfahren bzw. an der nächsten Betriebsstelle ankommen kann. Liegt die Soll-Abfahrtszeit später, wird diese

[34] Bei Aufruf von `next`, also der klassischen Generator-Nutzung ohne Rückkanal, wird `None` zurückgegeben.

[35] Es existiert jeweils immer nur für einen der beiden ein Wert, da keine Prognosezeit berechnet wird, wo schon eine Ist-Zeit existiert.

verwendet. Zur Ermittlung der Zeit, die ein Zug für eine Strecke bzw. einen Halt fahrplanmäßig benötigt, werden die Plan-Zeiten herangezogen.

Falls der Zug Verspätung hat, werden hier jedoch Kürzungen vorgenommen. Bei der Erstellung des Fahrplans wird die Zeit berechnet, in der der Zug die Strecke zwischen zwei Betriebsstellen zurücklegen kann. Darauf wird im derzeitigen Fahrplan eine Pufferzeit zwischen 3 und 4 % addiert, was dem Regelwerk der DB (mindestens zur Zeit des Entwurfs des Fahrplankonzepts im EBuEf, 2010) entspricht [15]. Die tatsächlichen Haltezeiten können wesentlich größer sein als die mindestens notwendige Haltezeit, z. B. wenn ein Anschluss oder eine Überholung abgewartet werden muss.

Wenn ein Zug Verspätung hat, wird die Fahrzeit um die Pufferzeit gekürzt; für die Haltezeit wird der Wert angenommen, der für diesen Bahnhof und Zugtyp in der Tabelle `fahrplan_mindesthaltezeiten`[36] hinterlegt ist, wenn dieser kleiner ist als die laut Plan-Fahrplan vorgesehene Haltezeit.

`Response` ist die Basisklasse der Antworten, die `Manager` den `Journey`-Instanzen sendet. Wie beschrieben gibt es zwei mögliche Antworten, `Admitted`, wenn ein Fahrtwunsch möglich ist und immer bei `Arrival`; sowie `NotFree`, wenn benötigte Gleisabschnitte belegt sind. Die Klassen enthalten keinen Code, sondern dienen nur der Unterscheidung der Antworten; außerdem enthält `NotFree` als Zusatzinformation die Zeit, zu der erwartet wird, dass die Gleisabschnitte wieder frei sind, so dass `Journey` für diese Zeit einen erneuten Fahrtwunsch anmelden kann.

`Action` ist die Basisklasse für Aktionen, die Züge durchführen können. Es gibt Fahrtwunsch (`Ride`), Ankunftmeldung (`Arrive`) sowie Ende einer Fahrt (`EndJourney`, dient zur Freigabe sämtlicher belegten Abschnitte). Für eine Verfeinerung der Prognose sind aber zukünftig noch weitere Unterklassen denkbar, beispielsweise die Information, dass ein Zug auf einem Gleis stehen bleiben und unter anderer Zugnummer weiterfahren wird. Die Instanzen von `Journey` teilen dem `Manager` durch Objekte eines dieser Klassen den Fahrtverlauf des Zuges mit.

Kern dieser Klassen ist die Methode `allocate`, die von `Manager` aufgerufen wird und die für die Aktion benötigten Gleisabschnitte als belegt markiert.

Dafür werden zunächst die benötigten Gleisabschnitte gesucht. Diese Funktionalität ist in jeder Klasse getrennt implementiert, bei `Arrive` ist der benötigte Gleisabschnitt nur das Gleis, an dem der Zug stehen bleibt und bei `EndJourney` gar keines. Bei `Ride` ist es wesentlich komplizierter:

Aus Zeitgründen und da bisher nicht für die gesamte Anlage Fahrstraßentabellen vorliegen, mit denen exakt die benötigten Gleisabschnitte herausgefunden werden können, wird hier vorerst ein primitiver Algorithmus angewandt: Es wird lediglich der Start- und Zielpunkt der Fahrt (jeweils die Betriebsstelle und Gleisnummer) sowie die Strecke zwischen den beiden Betriebsstellen als benötigter Gleisabschnitt zurückgegeben. Diese Vorgehensweise funktioniert für einander folgende Züge recht gut,

[36] Siehe die Erläuterungen auf Seite 10.

Konflikte bei entgegenkommenden Zügen auf eingleisigen Strecken sowie Kreuzungen zweier Zugfahrten werden jedoch nicht erkannt. Auch Sonderfälle wie die Benutzung des Gegengleises[37] werden davon nicht erfasst. Ein besseres Verfahren kann jedoch sehr leicht hinzugefügt werden, da nur an dieser Stelle Code verändert werden muss, die anderen Klassen können unverändert bleiben.

Der Rest der Handlungen ist in allen Fällen gleich und daher in der Basisklasse `Action` implementiert. Zunächst wird geprüft, ob die alle benötigten Gleisabschnitte frei sind. Sind sie das nicht, wird eine Instanz von `NotFree` zurückgegeben. Andernfalls werden alle benötigten Gleisabschnitte als belegt markiert sowie die Gleisabschnitte, die der Zug vorher belegt hatte, aber nun nicht mehr benötigt, freigegeben, und eine Instanz von `Admitted` zurückgegeben. In beiden Fällen reicht `Manager` das zurückgegebene `Response`-Objekt an `Journey` weiter.

`Manager` ist die Klasse, die die Simulation steuert. Sie wird mit allen zu Zügen, deren Fahrzeit zu simulieren ist, sowie der aktuellen Simulationsuhrzeit initialisiert. Die Uhrzeit wird benötigt, da ein Zug, dessen Abfahrt sich verzögert, keinen Eintrag in den Ist-Daten generiert; eine solche Verspätung ist folglich nur daran zu erkennen, dass die Soll-Abfahrtszeit bereits abgelaufen ist, ohne dass ein entsprechender Ist-Eintrag existiert.

Zu Beginn ruft `Manager` die Methode `run()` jedes Zuges auf, und speichert den dabei zurückgegebenen Generator. Anschließend wird die `next`-Methode jedes Generators aufgerufen, und so die erste Aktion jedes Zuges ermittelt. Alle diese Aktionen werden in einer Warteliste gespeichert. Diese Warteliste wird nach der Uhrzeit der Aktion sortiert[38]. Die Zeit der frühesten Aktion in der Warteliste entspricht dem Zeitpunkt, der gerade simuliert wird, d. h. die Gleisbelegungszustände, wie sie im `Manager` gespeichert sind, entspricht der Gleisbelegung, die zu diesem Zeitpunkt auf der Anlage zu erwarten ist.

Die früheste Aktion wird entnommen und ihre `allocate`-Methode aufgerufen (s. o.). Deren Rückgabewert, eine Instanz einer Unterklasse von `Response`, wird über die `send`-Methode des entsprechenden Generators an das zugehörige `Journey`-Objekt zurückgegeben. Dessen Code wird daraufhin (als Koroutine) weiter ausgeführt. Beim nächsten `yield`-Ausdruck wird die Ausführung wieder unterbrochen. Der mit `yield` zurückgegebene Wert (der auch der Rückgabewert des `send`-Aufrufs ist) ist die nächste Aktion des Zuges, diese wird in die Warteliste eingefügt. Die in diesem Absatz beschriebene Prozedur wird wiederholt, bis die Warteliste keine Einträge mehr hat.

Laufzeitanalyse Die Worst-Case-Komplexität des Algorithmus ist ungefährt proportional zum Produkt aus der Anzahl der simulierten Züge, der Anzahl der Fahrplaneinträge eines Zuges, der Anzahl der insgesamt vorhandenen Gleisabschnitte und der Anzahl der gleichzeitig durch einen Zug belegten Abschnitte. Dies gilt unter der Annahme, dass negative Antworten auf Fahrtwünsche (und damit ein erneuter, späterer

[37] und damit Behinderung von Zügen der Gegenrichtung

[38] Diese Uhrzeit ist bei `Ride` die Abfahrts-, bei `Arrival` und `EndJourney` die Ankunftszeit.

Versuch) eher selten vorkommen, was jedoch realistisch ist. Die zu den einzelnen Eingabegrößen proportionale Komplexität beruht darauf, dass die Python-Datenstrukturen `list` und `dict` häufig benötigte Operation sehr effizient durchführen können [16], insbesondere benötigt der intern verwendete Sortieralgorithmus bei weitgehend sortierten Listen[39] nur $O(n)$ Vergleiche [17].

Die Zahl der Züge und Gleisabschnitte ist jeweils niedrig dreistellig, die beiden anderen Werte sind niedrig zweistellig. Aufgrund dieser niedrigen Werte ist der konstante Faktor vermutlich nicht vernachlässigbar, so dass die Angabe einer Komplexitätsklasse nicht sehr aussagekräftig ist. Außerdem liegt die tatsächliche Laufzeit durch eine Vielzahl von Datenbankabfragen[40] vermutlich wesentlich über der reinen Rechenzeit.

Eine einfache Laufzeitmessung mit dem aktuellen Algorithmus ergibt eine Laufzeit von durchschnittlich 3.8 Sekunden bei etwa 120 Zügen. Eine Reduktion der Anzahl der prognostizierten Züge bestätigt die ermittelte lineare Laufzeit (60 Züge: 1.7 s, 30 Züge: 0.9 s).

Bewertung Die Art der Implementierung durch eine Simulation des Betriebsablaufs[41] bietet verschiedene Vorteile. So können durch die naturgemäße Nähe zum realen Betrieb alle denkbaren Fahrzeugbewegungen und sonstigen Umstände in der Prognose berücksichtigt werden, ohne dass die gesamte Prognose neu programmiert werden müsste. Hierzu zählt beispielsweise eine Verbesserung des provisorischen, primitiven Algorithmus, der die belegten Gleisabschnitte herausfindet (wofür nur der Code der Klasse `Ride` angepasst werden muss), oder dass ein Gleis nach Ende einer Zugfahrt besetzt bleibt, bis der Zug in ein Abstellgleis rangiert wird oder unter einer anderen Zugnummer weiterfährt (wofür nur neue `Action`-Unterklassen nötig sind).

Hierzu trägt auch die Implementierung mittels Koroutinen bei, wodurch die verschiedenen Aufgaben sauber voneinander getrennt werden konnten sowie der Code der `Manager`-Klasse auf ein Minimum reduziert werden konnte.

Erwähnenswert ist weiter ein kleiner Bruch des Koroutinen-Modells: Der Zeitpunkt, an dem ein Zug einen Gleisabschnitt wieder freigeben wird, ist noch nicht bekannt, wenn der Fahrtwunsch abgegeben wird, sondern kann erst endgültig berechnet werden, wenn dieser angenommen wurde. Daher wird dieser Zeitpunkt nicht über den ansonsten genutzten Kommunikationsweg (das `yield`-Statement) mitgeteilt, sondern über eine nachträgliche Manipulation an dem gesendeten `Action`-Objekt. Dies ist technisch irrelevant, da es direkt nach dem Senden erfolgt, bevor im `Manager` weiterer Code ausgeführt wird und Koroutinen in Python nicht in eigenen Threads ausgeführt werden; es stellt lediglich eine Verletzung des verwendeten Programmierprinzips dar.

Die mit einigen Sekunden recht lange Laufzeit ist nicht kritisch, da die ZWL-Bilder

[39] Dies trifft für das (recht oft erfolgende) Sortieren der Warteliste in `Manager` zu.

[40] Besonders fällt hier die Abfrage der Mindesthaltezeit ins Gewicht, die für jeden Fahrplaneintrag durchgeführt werden muss.

[41] Ein anderes denkbares Verfahren wäre z. B. gewesen, für verschiedene Typen von Konfliktpunkten (Bahnhofseinfahrt, freie Strecke, eingleisige Strecke) jeweils eine eigene Konflikterkennung und Fahrzeitprognose durchzuführen.

etwa alle 15 Sekunden aktualisiert werden sollen, die Prognose folglich auch nur in diesem Abstand ausgeführt werden muss. Da die neuen Prognosedaten auch erst nach dem vollständigen Durchlauf der Simulation in die Datenbank geschrieben werden, kommt es auch nicht zu Problemen durch inkonsistente oder fehlende Daten. Dennoch wäre eine Optimierung zur Reduktion der Serverlast sehr wünschenswert.

3.4.2 Aufbereitung des Fahrplans

Der zugehörige Quellcode findet sich in der Datei `backend/zwl/trains.py`.

Die Notwendigkeit des Algorithmus wurde bereits oben beschrieben (siehe Abschnitt 3.3.2 auf Seite 27). Eingabe ist der Fahrplan eines Zuges und die Streckenkonfiguration, für die der Fahrplan aufbereitet werden soll. Ausgegeben werden sollen ein oder mehrere Fahrplansegmente, in der Form, in der sie vom Frontend benötigt werden.

Im Codebeispiel 7 beispielhaft ein Eingabefahrplan dargestellt[42]. Die Streckenkonfiguration ist die, der die Anzeige in Abbildung 6 (Seite 15) zugrunde liegt, sie enthält – in dieser Reihenfolge – die Betriebsstellen `XWF#1`, `XCE#2`, `XAP#2`, `XDE#2`, `XBG#2`, `XLG#2` und `XWF#3`[43]. Der Zug im Beispiel wird also im Graph auf der linken Seite in XCE beginnen, in XWF den sichtbaren Bereich verlassen und auf der rechten Seite wieder erscheinen, um schließlich in XBG zu enden. Aufgabe des Algorithmus ist, den chronologisch sortierten Fahrplan auf diese Betriebsstellen abzubilden, das Ergebnis ist in Codebeispiel 8 dargestellt.

Der Algorithmus macht sich die Regel zu Nutze, dass (sowohl im EBuEf als auch bei der DB [3]) pro Zugnummer eine Betriebsstelle nur ein Mal durchfahren werden darf. Dadurch ist es möglich, die in der Streckenkonfiguration enthaltenen Betriebsstellen durchzugehen und zu überprüfen, ob diese Betriebsstelle auch im Fahrplan enthalten ist. Wenn eine übereinstimmende Betriebsstelle gefunden ist, wird für die in der Streckenkonfiguration folgende Betriebsstelle nach einem entsprechenden Fahrplaneintrag gesucht (die Suche ist dabei auf Fahrplaneinträge in zeitlicher Nachbarschaft zum ersten Eintrag beschränkt). Der Fundort dieses Fahrplaneintrags gibt die Richtung an, in die der Zug fährt[44].

Ab der dritten Betriebsstelle wird nur noch in dieser Richtung nach Übereinstimmungen gesucht. Wird keine Übereinstimmung gefunden, ist die Zugfahrt beendet oder der Zug hat die Strecke verlassen. In diesem Fall wird das bisher gefundene Segment ausgegeben und der Vorgang mit den verbliebenen Betriebsstellen wiederholt, so dass ggf. weitere Segmente ausgegeben werden.

Die Suche ist dabei etwas tolerant gegenüber fehlenden Fahrplaneinträgen. Dies ist u. a. nötig, da die Streckenkonfiguration die für beide Fahrtrichtungen gültigen Signale

[42] Sowohl dieser Fahrplan als auch die Streckenkonfiguration sind vereinfacht, tatsächlich enthalten beide nicht nur die Bahnhöfe, sondern auch Einträge für sämtliche Signale (in den Screenshots des ZWL-Diagramms erkennbar an den Knicken in den Zuglinien).

[43] Die Auswahl der Zahlen ist dabei willkürlich, sie muss lediglich eindeutig sein.

[44] D. h. ob die Zuglinie im Diagramm von links oben nach rechts unten oder andersherum verläuft.

Codebeispiel 7: Eingabe des Algorithmus. Die Tabelle enthält tatsächlich noch weitere Spalten, die hier aber nicht relevant sind, sowie sekundengenaue Angaben.

```
Betriebsstelle | arr_plan | dep_plan |
XCE            |          | 15:22    |
XWF            | 15:24    | 15:25    |
XLG            | 15:28    | 15:28    |
XBG            | 15:29    |          |
```

Codebeispiel 8: Ausgabe des Algorithmus. Die Ausgabe wird noch um Informationen zum Zug ergänzt. Der Lesbarkeit halber sind die Zeiten hier als String angegeben, tatsächlich sind es Unix-Timestamps.

```
'segments': [
    {'direction':'left',
     'timetable':[
        {'loc':'XCE#2', 'arr_plan':null,    'dep_plan':'15:22'},
        {'loc':'XWF#1', 'arr_plan':'15:24', 'dep_plan':'15:25'}
    ]},
    {'direction':'left',
     'timetable':[
        {'loc':'XWF#3', 'arr_plan':'15:24', 'dep_plan':'15:25'},
        {'loc':'XLG#2', 'arr_plan':'15:28', 'dep_plan':'15:28'},
        {'loc':'XCE#2', 'arr_plan':'15:24', 'dep_plan':null}
    ]}
]
```

enthält; Im Fahrplan sind jedoch nur die Signale enthalten, die für die Fahrtrichtung des Zuges relevant sind. Daher wird die Suche nicht sofort beim ersten Fehlschlag abgebrochen, sondern in begrenztem Rahmen weiter gesucht.

Laufzeit und Bewertung Bei diesem Algorithmus ist eine kurze Laufzeit wesentlich wichtiger, einerseits, weil er bei einer Anfrage vom Frontend ausgeführt wird, d. h. eine kurze Antwortzeit gewünscht wird, andererseits, weil er sehr oft läuft: Es müssen bei jeder Aktualisierung eines Graphen die Fahrpläne mehrerer (ca. 10–20) Züge aufbereitet werden. Diese Aktualisierung findet regelmäßig mehrmals pro Minute und außerdem bei jedem Verschieben der Ansicht statt. Im vollen Betrieb wird das ZWL-Bild an etwa zehn Arbeitsplätzen gleichzeitig angezeigt werden.

Die Komplexität ist proportional zur Anzahl der Fahrplaneinträge. Da diese jedoch i. d. R. niedrig zweistellig ist, ist eine asymptotische Betrachtung hier wenig zielführend. Die gemessene Laufzeit liegt bei etwa 1 ms pro Lauf (d. h. pro Zug), was völlig zufriedenstellend ist. Hierbei ist die Datenbankabfrage der Fahrplandaten nicht enthalten, da jedoch die Fahrplandaten aller Züge gesammelt abgefragt werden, ist die hierfür benötigte Zeit vernachlässigbar.

4 Fazit

Im Zuge dieser Arbeit wurde eine Anwendung zur Anzeige von ZWL-Diagrammen („Frontend“) sowie zur dafür nötigen Aufbereitung der Fahrplandaten („Backend“) entwickelt.

Im Frontend ist es möglich, sich die Fahrzeiten der Züge und elementare Informationen zu den Zügen übersichtlich auf einem Bildschirm anzeigen zu lassen. Dabei ist eine freie Auswahl des anzuzeigenden Streckenabschnitts möglich und es können je nach Bedarf mehrere Strecken angezeigt werden. Auch der anzuzeigende Zeitraum ist wählbar.

Im Backend werden die Daten aus der Datenbank abgerufen, aufbereitet und dem Frontend in geeigneter Form zur Verfügung gestellt. Außerdem können anhand der in der Datenbank hinterlegten Ist-Fahrplanzeiten die zu erwartenden Fahrzeiten prognostiziert werden, dabei werden auch teilweise zu erwartende Konflikte zwischen den Zügen mit einbezogen.

Wie bereits beschrieben, war es aus Zeitgründen nicht Ziel der Arbeit, alle in Kapitel 2.3 beschriebenen Anforderungen vollständig umzusetzen, sondern z. T. nur vorzubereiten. Die nicht oder nicht vollständig umgesetzten Anforderungen sind:

- Die Ist-Zeiten werden von einer anderen, unabhängig entwickelten Anwendung in die Datenbank geschrieben. Hier gibt es jedoch kleine Lücken (nicht zu allen Betriebsstellen können Ist-Zeiten ermittelt werden). Daher zeigt das ZWL-Diagramm in der momentanen Version nur die Plan-Fahrzeiten an. Mangels Notwendigkeit ist daher auch die automatische Aktualisierung noch nicht vollständig implementiert.
- Die Prognose ist zwar in sich funktionsfähig, aber noch nicht so eingerichtet, dass sie regelmäßig ausgeführt wird und Prognosedaten in die Datenbank schreibt.
- Die Prognose ist, wie bereits beschrieben, bisher erst mit einem grundlegenden, primitiven Algorithmus implementiert, der nicht alle möglichen Konflikte erkennt.
- Es werden beim Überfahren der Zuglinie nur ein Teil der möglichen und wünschenswerten Daten zum Zug angezeigt.
- Filterung und Suche nach Zugnummern sind noch nicht implementiert.
- Sonderfälle, z. B. die Benutzung des Gleises der Gegenrichtung oder Lademaßüberschreitungen (Lü), werden im Frontend noch nicht gesondert hervorgehoben.
- Sperrungen von Gleisen und Kommentare werden noch nicht unterstützt.

4.1 Probleme

Mir war während der Erstellung der Anwendung nicht klar, dass es bei den Ist-Zeiten Lücken geben würde[1], daher habe ich die Anwendung nicht darauf ausgelegt. Dies legt ein Kommunikations- bzw. Spezifikationsproblem offen. Der Grund hierfür ist ein systematischer: Auf der Modellbahnanlage ist es leicht möglich, die genaue Position jedes Zuges zu kennen, da die Züge zentral gesteuert werden. Bei der echten Bahn kann jedoch nur durch spezielle Einrichtungen punktuell die Position eines Zuges ermittelt werden. Um die Anlage im EBuEf möglichst realitätsnah zu halten, sollen nur diese Daten verwendet werden.

Die erwähnte sehr landesspezifische Ausrichtung des Bahn-Regelwerks hat das Finden von prägnanten Bezeichnern für Code-Elemente erschwert. Ich habe mich entschieden, alle Klassen, Funktionen u. dgl. auf englisch zu benennen, um eine Inkohärenz zu den von der Sprache und den verwendeten Bibliotheken vorgegebenen (englischen) Bezeichnern zu vermeiden[2]. Es war jedoch teilweise unmöglich, passende und prägnante englische Begriffe für deutsche Fachbegriffe zu finden, was den Code schwerer verständlich macht. Es ist zu überlegen, ob es in einem solchen speziellen Fall nicht besser gewesen wäre, die o. g. Inkohärenz hinzunehmen, und zumindest die bahnbetrieblichen Begriffe auch im Code auf deutsch zu verwenden.

4.2 Ausblick

Als nächster Arbeitsschritt liegt nahe, die Anwendung im EBuEf zu installieren und zunächst in der aktuellen Version (also mit Anzeige der Plan-Fahrzeiten) den Fahrdienstleiter_innen verfügbar zu machen, da sie auch so bereits einen Nutzen darstellt. Außerdem können so bereits erste Rückmeldungen zur weiteren Verbesserung der Software gesammelt werden.

Weiter sollten die einfach zu implementierenden, noch fehlenden Anforderungen implementiert werden, z. B. die vollständige Anzeige aller Informationen zum Zug, Suche nach Zugnummern oder Anzeige von Fahrten im Gegengleis etc.

Auch die Eingabe und Anzeige von Streckensperrungen und Kommentaren kann wahrscheinlich mit überschaubarem Aufwand implementiert werden. Hier ist zu überprüfen, inwiefern eine Mitnutzung dieser Daten durch andere Anwendungen im EBuEf möglich sein soll, und die Art der Speicherung in der Datenbank darauf abzustimmen.

Parallel dazu sollte eine Lösung des geschilderten Problems mit den Ist-Fahrzeiten erarbeitet werden. Hier sind zwei prinzipielle Möglichkeiten denkbar. Einerseits könnte die Software, die die Ist-Zeiten ermittelt, erweitert werden, so dass sie diese Lücken

[1] Das Programm, das diese Daten in die Datenbank schreibt, wurde etwa parallel zu meiner Anwendung entwickelt.

[2] Im Allgemeinen ist ein weiterer Grund, dass so eine Mitarbeit von nicht-deutschsprachigen Entwickler_innen möglich ist. Aufgrund starken Ausrichtung auf das EBuEf ist dies hier ohnehin nicht zu erwarten.

per Interpolation füllt, sobald an späterer Stelle Ist-Zeiten vorliegen. Das würde nur kleinere Anpassungen an der Prognose erfordern, da sie teilweise auch in der Vergangenheit Fahrzeiten prognostizieren müsste[3]. Andererseits wäre es möglich, die Prognose so zu erweitern, dass sie auch den Verlauf von Fahrten in den Lücken prognostiziert und so geeignete Ist-Daten für die Lücken erzeugt. Dies würde sehr große Anpassungen der Prognose erfordern.

Die beiden angedachten und (soweit möglich) bei der Konzeption mit bedachten Erweiterungen, nämlich die Eingabe von Dispositionsentscheidungen (Veränderung von Fahrzeiten und Gleisen) sowie der Anzeige von Sperrzeitentreppen, stellen vermutlich einen größeren Aufwand dar. Dies könnte Gegenstand einer weiteren Arbeit sein. Mit meiner Arbeit liegt eine ausführliche Dokumentation des existierenden Codes vor, die einer solchen Weiterentwicklung den Weg bereitet.

[3] In Fällen, in denen Fahrten bereits stattgefunden haben, aber diese dem System noch nicht bekannt sind, weil der Zug noch nicht an einem Ist-Messpunkt vorbeigefahren ist.

Quellenverzeichnis

[1] Jürgen Siegmann und Christian Blome: *Unterlagen der Vorlesung „Grundlagen des Schienenverkehrs“.* Technische Universität Berlin, Sommersemester 2013.

[2] Jörn Pachl: *Systemtechnik des Schienenverkehrs.* Springer, 2013.

[3] Christian Blome: *Auskünfte im Rahmen der Betreuung dieser Arbeit*, 2014 und 2015.

[4] *Besuch in der Betriebszentrale Berlin Pankow der DB Netz AG und Gespräch mit Mitarbeitenden dort*, Dezember 2014.

[5] DB Netz AG: *Produktbeschreibung LeiDis-NK Basisversion.* http://fahrweg.dbnetze.com/fahrweg-de/start/produkte/nebenleistungen/produkte/leitsystem_leidis_nk.html (abgerufen: 29.03.2015).

[6] *Auskünfte eines ehemaligen Fahrdienstleiters und Mitarbeiters im EBuEf*, 2015.

[7] *Webseite des EBuEf.* http://www.ebuef.de/das-ebuef/ (abgerufen: 14.03.2015).

[8] DB Netz AG: *Richtlinie 408: Züge fahren und Rangieren*, 2012.

[9] Python Wiki: *Should I use Python 2 or Python 3 for my development activity?* https://wiki.python.org/moin/Python2orPython3 (abgerufen: 04.04.2015).

[10] Flask Documentation: *The Status of Python 3.* http://flask.pocoo.org/docs/0.10/advanced_foreword/ (abgerufen: 04.04.2015).

[11] Erin Swenson-Healey: *The JavaScript Event Loop: Explained.* http://blog.carbonfive.com/2013/10/27/the-javascript-event-loop-explained/ (abgerufen: 24.03.2015).

[12] Web Hypertext Application Technology Working Group (WHATWG): *XMLHttpRequest Standard.* https://xhr.spec.whatwg.org/ (abgerufen: 24.03.2015), Abschnitt *The open() method.*

[13] Yehuda Katz: *Understanding “Prototypes” in JavaScript.* http://yehudakatz.com/2011/08/12/understanding-prototypes-in-javascript/ (abgerufen: 17.03.2015).

[14] Guido van Rossum und Phillip J. Eby: *PEP 342 – Coroutines via Enhanced Generators*, 2005. https://www.python.org/dev/peps/pep-0342/ (abgerufen: 06.04.2015).

[15] Armin Emde: *Entwicklung eines skalierbaren Betriebskonzeptes für das Eisenbahn-Betriebs- und Experimentierfeld Berlin.* Bachelorarbeit am Fachgebiet Schienenfahrwege und Bahnbetrieb, TU Berlin, Oktober 2010. Zitiert nach [3].

[16] Python Wiki: *TimeComplexity.* https://wiki.python.org/moin/TimeComplexity (abgerufen: 14.04.2015).

[17] Tim Peters: *listsort.txt.* https://hg.python.org/cpython/file/1589203ff116/Objects/listsort.txt (abgerufen: 14.04.2015).