Bachelorarbeit

Entwurf und Implementierung von regelmäßigen, randomisierten und bedarfsorientierten Abfahrtsplänen für Züge in einer Verkehrssimulation

Design and Implementation of Regular, Randomized and Demand-Driven Departure Schedules for Trains in a Traffic Simulation

Christian J. Raue

Hasso-Plattner-Institut an der Universität Potsdam

7. Oktober 2023

Bachelorarbeit

Entwurf und Implementierung von regelmäßigen, randomisierten und bedarfsorientierten Abfahrtsplänen für Züge in einer Verkehrssimulation

Design and Implementation of Regular, Randomized and Demand-Driven Departure Schedules for Trains in a Traffic Simulation

von
Christian J. Raue

Betreuung
Prof. Dr. Andreas Polze, Arne Boockmeyer, Lukas Pirl
Professur für Betriebssysteme und Middleware
Henry Huebler, Götz Gassauer
DB Systel GmbH

Hasso-Plattner-Institut an der Universität Potsdam

7. Oktober 2023

Zusammenfassung

Das Projekt *FlexiDug* erforscht die Nachnutzungsmöglichkeiten der Schieneninfrastruktur, die bisher dem Braunkohletransport in der Lausitz diente. Ziel ist es, ein Konzept für eine gemeinsame Nutzung für Personen- und Güterzüge zu erstellen. Diese Arbeit ist Teil eines Softwareprojekts, das die Simulation des Zugverkehrs auf Basis einer Erweiterung des Verkehrssimulators *SUMO* ermöglicht. Die Arbeit beschäftigt sich mit der Erzeugung von Zügen innerhalb der Simulation. Dafür werden drei verschiedene Arten von Abfahrtsplänen benötigt: regelmäßige, zufällige und bedarfsorientierte. Bedarfsorientierte Abfahrtspläne basieren auf dem berechneten Kohlebedarf von Kraftwerken. Die Berechnung basiert auf Stromproduktionsdaten der Bundesnetzagentur. Die Ergebnisse der Simulation zeigen, dass die Abfahrtsperioden der Kohlezüge mit dem Kohlebedarf korrelieren und dass die Software einen realistischen Zugverkehr simulieren kann.

Inhaltsverzeichnis

1 Einleitung

Dieses Projekt beschäftigt sich mit der Entwicklung einer Simulationsumgebung für den Zugverkehr. Die Arbeit an diesem Projekt fand im Team statt. Ziel war es, den gemeinsamen Verkehr von Personen- und Güterzügen auf einem Schienennetz zu simulieren, welches bisher ausschließlich dem Transport von Braunkohle dient. In dieser Arbeit wird die Erzeugung von Zügen innerhalb der Simulation genauer beleuchtet. In der Einleitung werden zunächst die Motivation und die aktuelle Situation vorgestellt. Außerdem wird ein Überblick über das Gesamtprojekt, die Teamarbeit innerhalb des Projektes und die Anforderungen gegeben. Der Grundlagenteil klärt grundlegende Begriffe und Konzepte. Im Hauptteil wird die Umsetzung der Anforderungen beschrieben. Einen großen Teil davon nimmt die Diskussion der entworfenen Softwarearchitektur ein. Abschließend werden die Ergebnisse der Arbeit diskutiert und ein Ausblick auf mögliche Erweiterungen gegeben.

1.1 Strukturwandel in der Lausitz

Die Lausitz, eine Region in Südbrandenburg, steht in naher Zukunft vor tiefgreifenden Veränderungen. Mit rund 8000 Beschäftigten [1] die LEAG AG mit dem Braunkohleabbau ein bedeutender Arbeitgeber in der Region. Braunkohle ist ein wichtiger Energieträger für die Strom- und Fernwärmeproduktion in der Lausitz. Im Rahmen der Energiewende soll jedoch die Energieproduktion aus Braunkohle bis zum Jahr 2038 zugunsten von erneuerbaren Energiequellen vollständig eingestellt werden. Dies wird die Region (und auch ganz Deutschland) vor eine Vielzahl von finanziellen, wirtschaftlichen und gesellschaftlichen Herausforderungen stellen. Es müssen neue Wirtschaftskonzepte für die Region entwickelt werden, um diesen Strukturwandel zu bewältigen. Dazu zählt zum einen eine Alternative für den Bergbau zu finden, um den vorhandenen Arbeitnehmern weiterhin Arbeitsplätze zur Verfügung stellen zu können. Zum anderen soll der Tourismus in der Region stark, durch die Nachnutzung der entstehenden Flächen als Erholungsgebiet, ausgebaut werden. Um das erreichen zu können, ist es notwendig, die Tagebaulandschaft zu renaturieren [2] und die notwendigen Flächen und Infrastrukturen zu erschließen. Dieser Strukturwandel stellt für die Lausitz den größten Wandel seit dem Strukturbruch im Jahr 1990 im Rahmen der Wiedervereinigung dar.

Es ist daher notwendig, die entsprechenden Maßnahmen ausreichend zu planen und mit den jeweiligen Interessenvertretern abzustimmen. Mit einem Teil dieser Planung beschäftigt sich das Projekt *FlexiDug*, welches in den folgenden Abschnitten genauer beschrieben wird. Im Rahmen des Projektes werden Nachnutzungsmöglichkeiten der Schieneninfrastruktur erkundet, welche bisher ausschließlich dem Transport der Braunkohle diente und aktuell in privater Hand liegt.

1.2 Das Schienennetz der LEAG

Essentiell für den Betrieb der Kohlekraftwerke ist die regelmäßige und zuverlässige Belieferung mit Braunkohle. Zu diesem Zweck betreibt die LEAG ein Schienennetz von 391 Kilometern Länge [10]. Es verbindet die Tagebaue *Jänschwalde*, *Welzow-Süd*, *Nochten* und *Reichwalde* mit den Braunkohlekraftwerken *Jänschwalde*, *Boxberg* und *Schwarze Pumpe*. Außerdem ist der Kohleveredelungsbetrieb in *Schwarze Pumpe* angeschlossen [29]. Es fahren gleichzeitig bis zu 25 Kohlezüge auf dem Netz. Sie dienen nicht nur allein dem Transport der Kohle. Ebenso befördern sie die Abfallprodukte der Kohleverstromung, Asche und Gips. Sie haben eine Maximalgeschwindigkeit von 50 km/h. Zum Fuhrpark gehören 61 E-Loks und 14 Diesel-Loks [10]. Abbildung 1.1 zeigt das Schienennetz der LEAG und die Positionen der angeschlossenen Tagebaue und Kraftwerke.

1.3 FlexiDug - Gemeinsam Nutzung durch Personen- und Güterzüge

Nach dem Kohleausstieg würde das Potential des Schienennetzes ohne ein Nachnutzungskonzept ungenutzt bleiben [19]. Das Projekt *FlexiDug* (Flexible, digitale Systeme für den schienengebundenen Verkehr in Wachstumsregionen) beschäftigt sich damit, Nachnutzungsperspektiven zu erstellen und ein solches Konzept bis zum Jahr 2024 zu erstellen [9]. Teil dieses Projektes sind die Analyse von Infrastrukturen, die Erstellung eines digitalen Zwillings des Schienennetzes, die Entwicklung von Sensornetzen zur Infrastrukturüberwachung und die Erforschung digitaler Leit- und Sicherungstechnik. All dies soll einen Personenverkehr auf diesem Schienennetz ermöglichen. Das große Ziel des Projektes ist das „Ergründen von Perspektiven“ für die Nachnutzung der Infrastruktur und die Erforschung der Möglichkeit einer Nachnutzung, als auch einer gemeinsamen Nutzung durch Personen- und Güterzüge. [19] Diese Arbeit im Kontext des gesamten Softwareprojektes leistet einen Beitrag durch die Erweiterung des Verkehrssimulators *SUMO*[1] um ein realistischeres Modell des Zugverkehrs. Damit wird die Simulation von gleichzeitig fahrenden Personen- und Güterzügen auf dem Schienennetz der LEAG ermöglicht.

1.4 Anforderungen

Die Anforderungen an das Softwareprojekt haben wir als Team selbst aufgestellt. Dabei orientierten wir uns an den Bedürfnissen anderer Teilnehmer von *FlexiDug* und bereits existierenden Projekten.[2] Unser Projekt hat das Ziel, eine möglichst realistische Simulationsumgebung für Schienenverkehr bereitzustellen. Konkret soll eine nutzende Person in der Lage sein, ein Szenario zu konfigurieren, es im Anschluss durch Simulation auszuführen und zuletzt produzierte Ergebnisse zu erhalten. Die Konfiguration eines Szenarios sollte folgende Parameter beinhalten:

[1] `https://github.com/eclipse/sumo`

[2] Es handelt sich um Software, die bei der Professur für Betriebssysteme und Middleware am HPI entwickelt wurde.

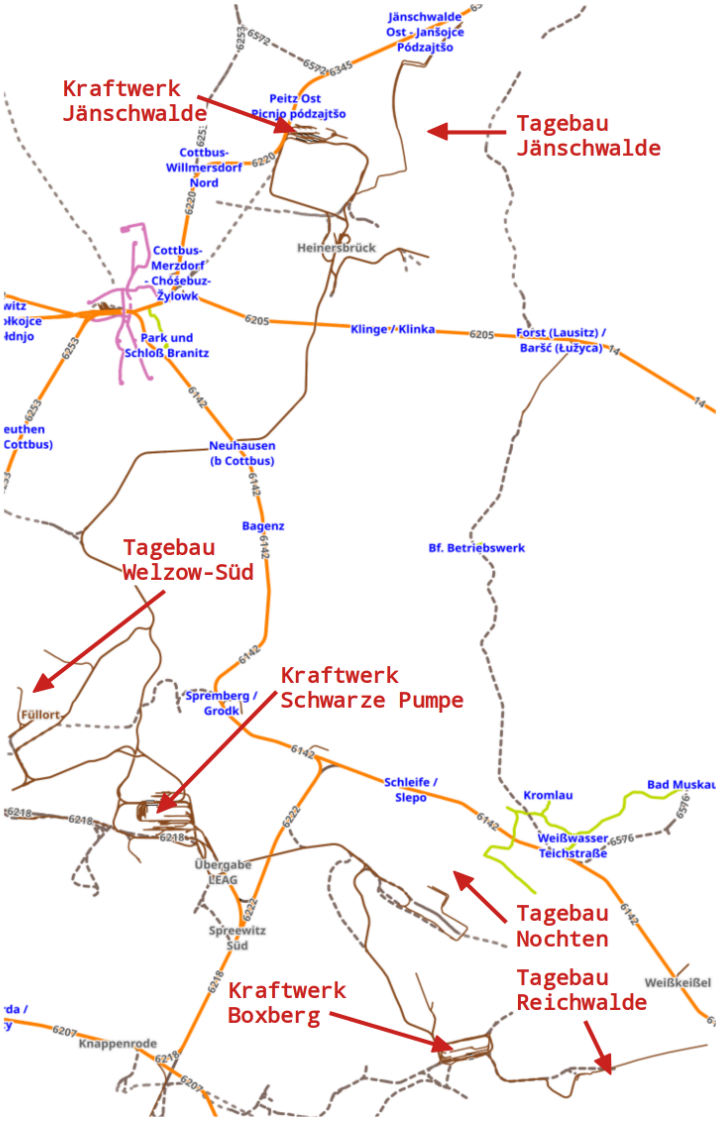

Abbildung 1.1: Das Schienennetz der LEAG für den Kohletransport (braun) und die Positionen der angeschlossenen Tagebaue und Kraftwerke (rot).

- Abfahrtsorte und Zeitpunkte von beliebig vielen Zügen, sowie deren Zwischenstationen
- Den Typ jedes Zuges (z. B. Güter- oder Personenzug)
- Verschiedene Fehler, welche unvorhergesehene Ereignisse simulieren (wie z. B. Weichenstörungen oder ausgefallene Züge)
- Eine textuelle Beschreibung der konfigurierten Simulation

Die Simulation wird von dem Verkehrssimulator *SUMO* [6] durchgeführt. Da dieser den Schienenverkehr jedoch nicht ausreichend realistisch abbilden kann, haben wir ihn um entsprechende Funktionalitäten erweitert. Diese sind:

- Eine eigens entwickelte Stellwerkslogik, die wir von der Professur für Betriebssysteme und Middleware übernommen und angepasst haben. [11]
- Die Schaffung einer objektorientierten Schnittstelle zu den Teilen der prozeduralen *SUMO*-API, welche wir für die Simulation von Schienenverkehr benötigen
- Ein Algorithmus, welcher die Züge durch das Schienennetz leitet, dabei Deadlocks vermeidet und die Stellwerkslogik entsprechend steuert
- Die Beachtung von Zugprioritäten in Abhängigkeit des Zugtyps

Letztendlich sollen folgende Ergebnisse durch die Simulation generiert werden können:

- Die Zeitpunkte, zu denen sich ein Zug an den konfigurierten Zwischenstationen befand
- Die Abweichungen dieser Zeitpunkte bei regelmäßig verkehrenden Zügen
- Die Verkehrsmenge, welche angibt, welche Strecke insgesamt von allen Zügen auf dem Schienennetz zurückgelegt wurde
- Die Verkehrsleistung, die die Verkehrsmenge pro Zeiteinheit betrachtet

Im Kontext unseres Projektes dient die Software außerdem dazu, den Zugverkehr auf dem Schienennetz in der Lausitz zu simulieren. Da wir die Fragestellung untersuchen möchten, ob eine gemeinsame Nutzung durch Kohle- und Personenzüge möglich ist, kommen weitere Anforderungen hinzu, die spezifisch für das untersuchte Schienennetz sind. So besteht die Notwendigkeit, die Züge zu realistischen Zeitpunkten abfahren zu lassen. Die Kohlezüge folgen, anders als die Personenzüge, keinem festgelegten Fahrplan. Vielmehr verkehren sie unregelmäßig und abhängig vom aktuellen Kohlebedarf zu verschiedenen Jahreszeiten und Wetterbedingungen. Dazu soll die Möglichkeit bestehen, die Kohlezüge anhand des Kohlebedarfs innerhalb der Simulation zu erzeugen, welcher aus historischen Daten stammt. Weiterhin sollen dieser Bedarf und die dazu erzeugten Kohlezüge Teil der generierten Ergebnisse sein.

Diese Arbeit beschäftigt sich mit dem Teil der Anwendung, welcher für die Erstellung der Züge zuständig ist. Die spezifischen Anforderungen an diesen Teil der Software sind, wie bereits zuvor erwähnt, die Erzeugung von Zügen zu den korrekten Zeitpunkten an den richtigen Abfahrtsorten. Dazu muss die hier betrachtete Softwarekomponente mit der Schnittstelle zu *SUMO* kommunizieren und dabei auch den Zugtyp und die Zwischenstationen eines jeden Zuges übergeben. Es soll möglich sein, einen Zug von jeder Station über beliebig viele weitere Stationen zu jeder Station fahren zu lassen. Die Zeitpunkte der Abfahrt sollen durch drei verschiedene Mechanismen erzeugbar sein:

- Züge fahren in regelmäßigen Zeitabständen ab.
- Züge fahren in zufälligen (einer Gleichverteilung folgenden) Zeitabständen ab.
- Züge fahren anhand eines Kolebedarfs ab.

1.5 Das Softwareprojekt und die Interaktion im Team

Dieser Abschnitt beschreibt die einzelnen Arbeitspakete des Softwareprojektes und wie die Verantwortlichkeiten innerhalb des Teams verteilt wurden. Weiterhin wird auf gemeinsame Teamarbeit eingegangen.

1.5.1 Arbeitspakete und Verantwortlichkeiten

Das Team besteht aus sechs Personen. Entsprechend wurde das Softwareprojekt in sechs Arbeitspakete geteilt, welche dann getrennt bearbeitet werden konnten. Die zugehörigen Arbeiten und die Pakete, mit denen sie sich befassen sind in Tabelle 1.1 dargestellt.

Tabelle 1.1: Die Arbeitspakete dieses Projektes, die Arbeiten, welche sich mit diesen beschäftigen und die Autoren dieser Arbeiten

Arbeitspaket	Titel der Arbeit	Autor
REST-API	„Architektur und Entwicklung einer REST-API zur Simulation von Zugverkehr am Beispiel des LEAG-Schienennetzes in der Lausitz“ [13]	Kamp
die Fehlerinjektion	„Fehlerinjektion als Methode zur Bewertung der Resilienz von simulierten Eisenbahnverkehrssystemen“ [17]	Persitzky
die Routenberechnung	„Evaluierung und Anpassung von Routingalgorithmen für das Finden von nebenläufigen Routen zwischen Betriebsstellen am Beispiel einer Simulation des LEAG-Schienennetzes in der Lausitz“ [15]	Lietze
die *SUMO*-Schnittstelle	„Implementierung und Evaluierung einer API-Abstraktion um eine simulatorunabhängige Zugsimulation zu ermöglichen“ [16]	Ortlam
die Datenauswertung	„Entwurf einer Software zur Protokollierung und Auswertung von Daten aus Simulationen von Zugverkehr am Beispiel des LEAG-Schienennetzes“ [20]	Reisener
die Zugerzeugung	Entwurf und Implementierung von regelmäßigen, randomisierten und bedarfsorientierten Abfahrtsplänen für Züge in einer Verkehrssimulation(diese Arbeit)	Christian J. Raue

Die *REST-API* stellt eine Benutzerschnittstelle für die von uns entwickelte Simulationsumgebung bereit. Die entsprechende Arbeit beschäftigt sich außerdem mit der komponentenübergreifenden Architektur und der Interaktion der einzelnen Softwarekomponenten. Die Fehlerinjektion dient der Erzeugung von Fehlern, welche der Simulation von unvorhergesehenen und unerwünschten Ereignissen im Rahmen des Bahnverkehrs dienen. Bei der Routenberechnung wird die Bewegung der Züge durch das Schienennetz geplant und gesteuert. Sie interagiert mit der Stellwerkslogik und bedient die Schnittstelle zu *SUMO*. Die *SUMO*-Schnittstelle ist ein objektorientierter *Wrapper*,[3] welcher die für uns wichtigen Aspekte der Schnittstelle zu *SUMO* kapselt und für uns die Arbeit mit ihnen erleichtert. Durch die Datenauswertung werden die Ergebnisse der Simulation grafisch dargestellt

[3] Ein *Wrapper* dient der Bereitstellung einer anderen Schnittstelle für bereits existierenden Funktionalitäten.

und die Analyse der Simulationsergebnisse ermöglicht. Die Zugerzeugung wird in dieser Arbeit ausführlich behandelt.

1.5.2 Die Teamarbeit

Um unser Projekt effizient und flexibel zu gestalten, haben wir agile Arbeitsmethoden angewendet. Dabei haben wir uns auf selbstbestimmtes Arbeiten und Gleichberechtigung aller Teammitglieder fokussiert. Um unsere Arbeit zu organisieren und zu koordinieren, haben wir verschiedene Werkzeuge genutzt. Das wichtigste davon war das GitHub-Repositorium[4] das uns die Quelltextverwaltung ermöglicht hat. Außerdem haben wir über GitHub ein Wiki als Wissenssammlung und ein *Kanban*-Board für die Verwaltung unserer Aufgaben erstellt. Das *Kanban*-Board hat uns geholfen, unsere Aufgaben zu visualisieren und transparent zu machen. Wir haben auch die Anzahl der gleichzeitig zu bearbeitenden Aufgaben begrenzt und unsere Arbeitsabläufe iterativ verbessert. Die Kommunikation haben wir durch die Kollaborationsplattform Slack und durch morgendliche Meetings gewährleistet.

[4] https://github.com/BP2022-AP1/bp2022-ap1

2 Grundlagen

Um den Zugverkehr in der Lausitz zuverlässig simulieren zu können, ist die Kenntnis von Konzepten sowohl aus dem Bahnwesen, als auch aus der Softwarearchitektur notwendig. Daher klärt der Grundlagenteil grundlegende Begriffe. Zunächst werden die Begriffe „Fahrplan“ und „Abfahrtsplan“ definiert und voneinander abgegrenzt. Anschließend werden die in der Softwarearchitektur verwendeten Entwurfsmuster[5] erläutert. Im Anschluss werden das Konzept der Objekt-Relationen-Abbildung (ORM[6]) und die Datenschnittstelle der Bundesnetzagentur für Stromproduktionsdaten vorgestellt.

2.1 Fahrpläne

Ein Fahrplan ist die zeitliche und räumliche Festlegung von Zugfahrten auf einem Schienennetz [8]. Er enthält unter anderem

- die Zuggattung,
- die Verkehrstage,
- die Strecke,
- die Ankunfts- Durchfahr- und Abfahrzeiten in allen Halte- und Betriebsstellen,
- und die zulässigen Geschwindigkeiten in den einzelnen Streckenabschnitten.

Der Fahrplan dient einer Reihe von Zwecken. Zum einen hilft der Fahrplan dabei, die vorhandene Infrastruktur auf die verkehrenden Züge zu verteilen und den Verkehr somit zu koordinieren. Damit können mögliche Konflikte bereits bei der Erstellung des Fahrplans lokalisiert werden. Zum anderen dient der Fahrplan als Informationsquelle für den Kunden und hilft diesem bei der Entscheidungsfindung bezüglich der Wahl der geeigneten Verkehrsmittel. Er dient den Triebfahrzeugführern und Fahrdienstleitern als Grundlage für ihre Betriebsplanung und ermöglicht Aussagen über benötigte Ressourcen. So bedeutet ein hoher Takt beispielsweise auch eine hohe Anzahl an benötigten Fahrzeugen und Personal. Zusätzlich ermöglicht ein Fahrplan das Treffen von Aussagen über die Leistungsfähigkeit des Zugverkehrs. [8]

2.2 Abfahrtspläne

Im Gegensatz zur klassischen Planung von Fahrplänen, arbeiten wir nicht mit festgelegten Ankunfts- und Durchfahrtzeiten an den einzelnen Stationen. Stattdessen legen wir ledig-

[5] auch engl. *Design Pattern*
[6] object relational mapping

lich die Abfahrtszeiten an den Startstationen fest und bestimmen dann die Ankunftszeiten an den Zwischenstationen und der Endstation durch Simulation. Daher wird zur Abgrenzung von dem Begriff Fahrplan hier der Begriff *Abfahrtsplan* eingeführt. Im Gegensatz zu einem Fahrplan enthält ein Abfahrtsplan lediglich folgende Informationen:

- Die Zuggattung (im Folgenden „Zugtyp“ genannt)
- Die Abfahrtszeiten an der Starthaltestelle
- Eine Liste der Zwischenhaltestellen und die Endhaltestelle, jedoch keine Ankunfts- oder Abfahrtzeiten

Wie bereits in den Anforderungen angedeutet, werden drei verschiedene Arten von Abfahrtsplänen benötigt. Solche, die zeitlich regelmäßige Zugfahrten beschreiben (regelmäßige Abfahrtspläne), solche, die Zugfahrten mit zufälligen Abfahrtszeiten beschreiben (zufällige Abfahrtspläne) und solche, deren Abfahrtzeiten sich nach einem historischen Kohlebedarf richten (bedarfsorientierte Abfahrtspläne). Jeder Abfahrtsplan hat einen Startzeitpunkt $t_s \in \mathbb{N}$ und einen Endzeitpunkt $t_e \in \mathbb{N}$. Da hier eine diskrete Zeitachse betrachtet wird, handelt es sich bei den Zeitpunkten um natürliche Zahlen. Der Zeitpunkt t_s beschreibt den Zeitpunkt in der Simulation, wenn der Abfahrtsplan aktiv wird, t_e beschreibt den Zeitpunkt, wenn der Abfahrtsplan deaktiviert wird. Für jeden Abfahrtszeitpunkt eines Zuges t_a gilt also $t_s \leq t_a \leq t_e$. Die Details der drei Varianten werden im Folgenden erklärt.

Regelmäßige Abfahrtspläne Der regelmäßige Abfahrtsplan beschreibt die Abfahrt von Zügen in regelmäßigen zeitlichen Abständen und dient der Modellierung von klassischem fahrplan-basiertem Verhalten. Er hat eine Periode $p \in \mathbb{N}$ in Sekunden, welche die Größe des zeitlichen Abstandes angibt. Ein regelmäßiger Abfahrtsplan lässt somit Züge zu den Zeitpunkten $t_a = t_s + np$ mit $n \in \mathbb{N}$, unter der Einschränkung $t_a \leq t_e$, abfahren.

Zufällige Abfahrtspläne Der zufällige Abfahrtsplan beschreibt die Abfahrt von Zügen in zufälligen zeitlichen Abständen. Er dient damit der Modellierung von Zügen, die unplanmäßig oder unvorhergesehen kurzfristig fahren. Dieser Abfahrtsplan hat eine Wahrscheinlichkeitsverteilung P, die angibt, mit welcher Wahrscheinlichkeit zu jedem diskreten Zeitpunkt der Simulation ein Zug erzeugt wird. P folgt einer Gleichverteilung. Die Wahrscheinlichkeit für eine Abfahrt ist also zu jedem Zeitpunkt gleich. Allerdings kann maximal ein Zug zu einem bestimmten Zeitpunkt abfahren.

Bedarfsorientierte Abfahrtspläne Der bedarfsorientierte Abfahrtsplan beschreibt die Abfahrt von Zügen in zeitlichen Abständen, die sich nach einem historischen Kohlebedarf richten. Er dient damit der Modellierung von Kohlezügen. Dieser Abfahrtsplan hat eine Kohlebedarfsfunktion $k : \mathbb{N} \to \mathbb{R}$, die angibt, wie viel Kohle zu einem Zeitpunkt t benötigt wird. Sie ergibt sich aus den historischen Kohlebedarfsdaten. Weiterhin sind der Zeitpunkt $t_0 \in \mathbb{N}$, an welchem die historischen Daten beginnen und die Zeitspanne $t_r \in \mathbb{N}$, welche die zeitliche Auflösung der Daten angibt, definiert. Außerdem ist $c \in \mathbb{R}$ die Menge an Kohle, die ein Kohlezug transportieren kann. Sowohl t_0, als auch t_r und c sind konstant und ergeben sich aus domänenspezifischen Vorgaben. Zur Berechnung der Abfahrtszeiten wird zunächst die Funktion $a : \mathbb{N} \to \mathbb{R}$ aufgestellt, die den akkumulierten

Kohlebedarf zum Zeitpunkt t berechnet (siehe Gleichung 2.1). Dabei ist $n \in \mathbb{N}$ die Anzahl der Datenpunkte.

$$a(t) = \sum_{i=0}^{\left(\frac{t-t_0}{t_r}\right)} k(t_r i + t_0) \tag{2.1}$$

Der Abfahrtsplan soll nun stets dann Züge abfahren lassen, wenn die Menge an Kohle, die transportiert werden muss, die Menge erreicht, die ein Zug transportieren kann. Die transportierte Menge wird dann von der bisher akkumulierten Menge abgezogen. Gleichung 2.2 stellt das dar. Die Funktion $fmod : (\mathbb{R}, \mathbb{R}) \to \mathbb{R}$ ist dabei definiert als $(x, y) \mapsto x - ny$ mit $n \in \mathbb{Z}$, $x < 0 \Leftrightarrow fmod(x, y) < 0$, $fmod(x, y) < |y|$ und $y \neq 0$. Intuitiv formuliert, gibt die Funktion den Rest der Division $\frac{x}{y}$ zurück.

$$a_{mod}(t, c) = fmod(a(t), c) \tag{2.2}$$

Abbildung 2.1 zeigt schematisch das Verhalten eines bedarfsorientierten Abfahrtsplans. Die Funktion k zeigt den (hier zur Vereinfachung konstanten) Kohlebedarf. Die Funktion a stellt den akkumulierten Kohlebedarf dar. Die Funktion a_{mod} zeigt die akkumulierte Kohlemenge abzüglich der transportierten Kohlemenge. Die Abfahrtszeiten sind durch rote Pfeile markiert.

2.3 Entwurfsmuster

Nach dem Architekten Christopher Alexander lässt sich ein Entwurfsmuster folgendermaßen beschreiben:

„Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.“[7] [7]

Obwohl diese Aussage auf Gebäude bezogen war, lässt sie sich ebenso gut auf Softwarearchitekturen anwenden. Auch in der Planung großer Softwaresysteme treten häufig dieselben abstrakten Probleme auf, welche sich mit einer Menge von Entwurfsmusters lösen lassen, welche die Beziehungen und Interaktionen von Objekten spezifizieren.

Ein Entwurfsmuster besteht dabei immer aus vier Elementen, seinem Namen, einer Problembeschreibung, einer Lösung und den sich ergebenden Konsequenzen [7]. Die Problembeschreibung gibt dabei an, in welchen Fällen sich ein Muster anwenden lässt, während die Lösung die Beziehungen und Verantwortlichkeiten der beteiligten Objekte beschreibt. Sie geht dabei nicht auf individuelle Anwendungsfälle ein, sondern stellt lediglich eine abstrakte Herangehensweise an das Problem bereit. Jedes Muster bringt Vor- und Nachteile mit sich und geht Kompromisse zwischen verschiedenen Qualitätsaspekten

[7] dt.: Jedes Muster beschreibt ein Problem, das in unserer Umgebung immer wieder auftritt und beschreibt dann den Kern der Lösung für dieses Problem, und zwar so, dass Sie diese Lösung millionenfach anwenden können, ohne sie jemals zweimal auf die gleiche Weise zu verwirklichen.

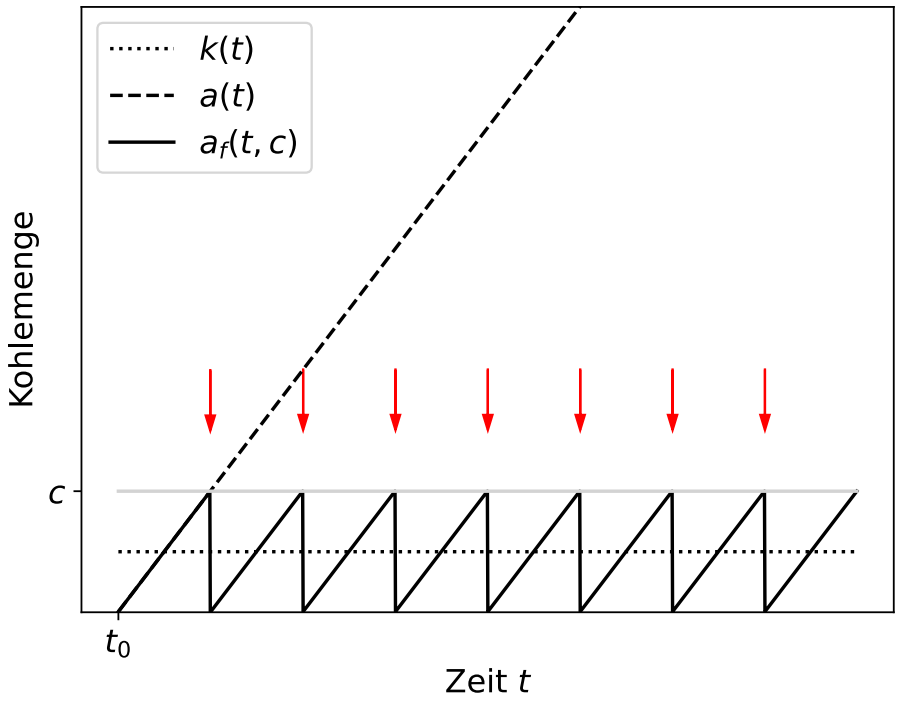

Abbildung 2.1: Schematisches Verhalten eines bedarfsorientierten Abfahrtsplans mit dem Kohlebedarf $k(t)$, dem akkumulierten Kohlebedarf $a(t)$ und der akkumulierten Kohlemenge abzüglich der transportierten Kohlemenge $a_{mod}(t, c)$. Die roten Pfeile markieren die Abfahrtszeiten der Kohlezüge.

ein. Die Konsequenzen beleuchten diese und helfen bei der Argumentation für oder gegen eine konkrete Designentscheidung.

Im Folgenden werden die abstrakten Entwurfsmusters beschrieben, welche in der Softwarearchitektur eine Rolle spielen werden.

2.3.1 Strategy Pattern

Problembeschreibung

Es kann vorkommen, dass mehrere Varianten eines Algorithmus benötigt werden, welche alle eine einheitliche Schnittstelle bereitstellen. Je nach Kontext wird ein anderer Algorithmus benötigt. Die Algorithmen sollen dynamisch austauschbar sein, um deren flexiblen Einsatz zu ermöglichen und eine lose Kopplung zu gewährleisten. Eine Implementierung der Algorithmen-Varianten innerhalb der ausführenden Klasse, würde deren Komplexität deutlich erhöhen. Außerdem soll die Möglichkeit bestehen, in Zukunft weitere Algorithmen hinzuzufügen. Das *Strategy Pattern* kann die Übersichtlichkeit verbessern, wenn viele ähnliche Klassen existieren, die sich lediglich in einem Teil ihres Verhaltens unterscheiden.

Weiterhin sollen Implementierungsdetails und algorithmusspezifische Daten vom Kontext abgekapselt werden. [7]

Lösung

Wie in Abbildung 2.2 abgebildet, besitzt der Kontext (`Context`), welcher einen Algorithmus verwenden soll, eine Referenz auf eine Strategie (`Strategy`). Die konkrete Strategie (`ConcreteStrategy`) wurde dem Kontext zuvor durch den Anwender (`Client`)[8] mithilfe von `setStrategy` zugewiesen. Die konkrete Strategie realisiert die `Strategy`-Schnittstelle, sodass Kontext und konkrete Strategie lediglich lose gekoppelt sind. Jede Strategie stellt die Methode `execute` bereit, um den gekapselten Algorithmus auszuführen.

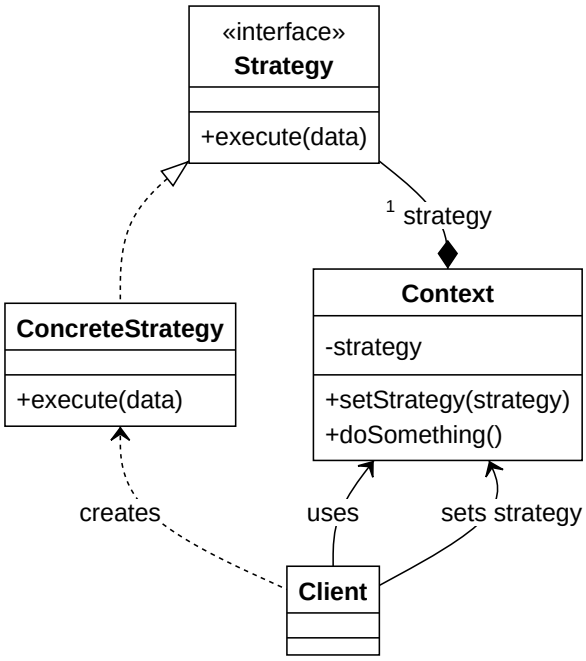

Abbildung 2.2: Klassendiagramm des *Strategy Patterns*. Der `Context` kann durch Nutzung der abstrakten `Strategy`-Schnittstelle eine Menge verschiedener Algorithmen nutzen. Die konkrete Strategie ist austauschbar und kann zur Laufzeit neu zugewiesen werden. [24]

Abbildung 2.3 zeigt die typische Verwendung des *Strategy Patterns*. Der Kontext wird initialisiert, indem ihm vom Anwender eine konkrete Strategie zugewiesen wird (3), welche zuvor vom Anwender erzeugt wurde (1). Der Anwender kann von nun an den Kontext dazu veranlassen, eine Aktion durchzuführen, welche die Verwendung des in der hinterlegten Strategie vorhandenen Algorithmus nach sich zieht (5). Der Kontext kann den Algorithmus ausführen (6), ohne Wissen über die konkrete Strategie zu benötigen.

[8] Der Anwender (`Client`) ist in der Regel eine weitere Klasse, welche den beschriebenen Mechanismus verwendet. Es handelt sich in aller Regel nicht um einen Menschen.

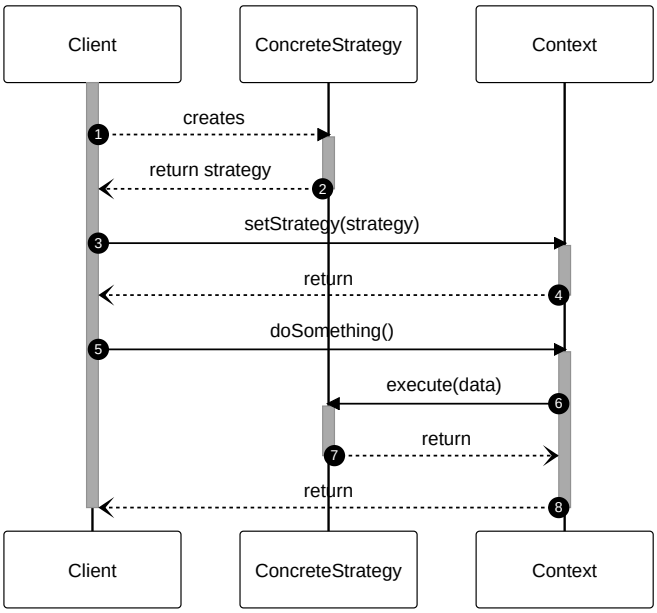

Abbildung 2.3: Sequenzdiagramm des *Strategy Patterns*. Der `Client` erzeugt eine neue konkrete Strategie (1) und weist sie dem `Context` zu (3). Dieser kann den in der Strategie implementierten Algorithmus dann nutzen (6), wenn er dazu aufgefordert wird (5). [24]

Konsequenzen

Das *Strategy Pattern* ermöglicht es, durch die Verwendung von Vererbung, eine Hierarchie von Algorithmen aufzubauen. Das kann hilfreich sein, wenn mehrere Algorithmen sich Teile ihrer Implementierung teilen. Das Muster extrahiert die Algorithmen-Implementierung aus dessen Kontext und verhindert damit die sonst notwendige Bildung von Subklassen des Kontextes. Die Auswahl des auszuführenden Algorithmus wird über Aggregation gesteuert. Damit entfällt die Notwendigkeit von bedingten Sprüngen. Weiterhin können dank des *Strategy Patterns* auch mehrere Algorithmen bereitgestellt werden, deren Verhalten identisch ist und sich beispielsweise nur in der Performance unterscheiden. So kann je nach Laufzeitumgebung eine Entscheidung für eine bessere Laufzeit oder einen effizienteren Umgang mit Speicherressourcen getroffen werden.

Nachteile des *Strategy Patterns* sind zum einen ein erhöhter Mehraufwand für Kommunikation. Unter Umständen benötigt ein Algorithmus nicht die gesamte Schnittstelle, um zu funktionieren. Da der Kontext allerdings nur die abstrakte Schnittstelle der Strategie kennt, muss er stets die gesamte Schnittstelle bedienen. Das bedeutet unnötige Aufrufe von Methoden oder unnötige Übergabe von Parametern. Zum anderen erhöht dieses Muster die Gesamtzahl an Objekten und damit die Komplexität zur Laufzeit. [7]

2.3.2 Template Method

Problembeschreibung

Ein Algorithmus besteht aus mehreren Schritten, die jeweils eine bestimmte Aufgabe erfüllen. Diese Schritte können je nach Situation oder Anforderung unterschiedlich implementiert werden, ohne den Algorithmus zu verändern. Der Algorithmus legt jedoch die Abfolge der Schritte fest und ruft sie in einer vorgegebenen Reihenfolge auf. [7]

Lösung

Jede Klasse, welche einen Algorithmus der selben Struktur implementiert, erbt von einer abstrakten Klassen (`AbstractClass`), welche die `templateMethod` implementiert. Das wird in Abbildung 2.4 gezeigt. Diese gibt das Grundgerüst des Algorithmus vor und ruft darin die einzelnen Teilschritte auf. Diese sind jeweils in eigenen Methoden implementiert. Die Subklassen (`ConcreteClass`) definieren diese Methoden zum Teil neu, wenn eine Veränderung des Verhaltens dieses Teilschrittes notwendig ist.

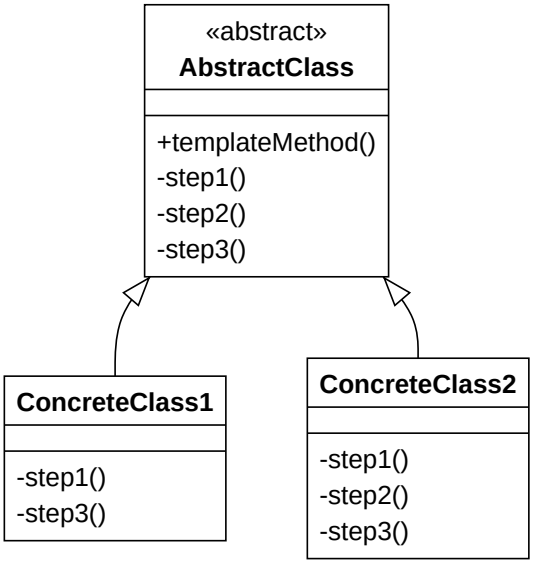

Abbildung 2.4: Klassendiagramm der *Template Method*. Sie gibt ein Gerüst eines Algorithmus in der Methode `templateMethod` vor. Die Details des Algorithmus werden in weiteren Methoden implementiert, sodass sie einzeln durch Spezialisierung der Basisklasse angepasst werden können. [25]

Konsequenzen

Template Methods sind ein Mechanismus, welcher die Wiederverwendung von Quelltext ermöglicht und somit der Codeduplikation entgegen wirken kann. Anstatt Abwandlungen von Algorithmen von Grund auf neu zu implementieren, ist es möglich, sie aus bestehenden Komponenten zusammenzusetzen und je nach Bedarf neuen Code hinzuzufügen.

Das Muster der *Template Method* weist eine umgekehrte Kontrollstruktur auf. Anstatt dass eine Klasse Methoden ihrer Superklasse aufruft, delegiert die *Template Method* die Verantwortlichkeit für die einzelnen Teile des Algorithmus an ihre Subklassen.

Bei der Verwendung der *Template Method* ist jedoch zu beachten, wie die einzelnen Methoden zu verwenden sind. Diese lassen sich grob in zwei Arten einteilen, die *Hook*[9]-Methoden und die abstrakten Methoden. Während die *Hook*-Methoden eine Standard-Implementierung in der abstrakten Basisklasse bereitstellen, ist dies bei den abstrakten Methoden nicht der Fall. Entsprechend müssen die abstrakten Methoden zwingend von einer konkreten Subklasse implementiert werden. Bei den *Hook*-Methoden ist das optional.

Die *Template Method* synergiert mit dem *Strategy Pattern*. Einzelne Schritte eines Algorithmus können in einer Strategie-Klasse implementiert sein. [7]

2.3.3 Observer Pattern

Problembeschreibung

Häufig müssen verschiedene Komponenten eines Systems synchron gehalten werden. Gleichzeitig soll aber auch eine enge Kopplung dieser Komponenten vermieden werden. Es wird eine 1:*n*-Beziehung zwischen den Objekten benötigt, wobei *n* Objekte von einem Objekt abhängen. Wenn das eine Objekt seinen Zustand ändert, so sollen alle abhängigen Objekte benachrichtigt werden, sodass auch sie ihren Zustand aktualisieren können. Ein naiver Lösungsansatz wäre, jedem abhängigen Objekt eine Referenz auf das Objekt zu geben, von welchem es abhängt. Die Objekte könnten dann in regelmäßigen Abständen prüfen, ob eine Zustandsänderung stattgefunden hat (*Polling*). Dieser Ansatz weist jedoch nicht nur eine hohe Kopplung auf, er ist auch wenig performant. Auch wenn keine Zustandsänderung stattgefunden hat, wird auf diese geprüft.

Das *Observer Pattern* kann Anwendung finden, wenn es zwei voneinander getrennte Konzepte gibt und eines von dem anderen abhängig ist. Die Abhängigkeit muss modelliert werden können, ohne die Objekte stark zu koppeln. Weiterhin soll es möglich sein, die Anzahl der abhängigen Objekte variabel zu halten. [7]

Lösung

Das *Observer Pattern* besteht aus einem Sender (`Publisher`) und mehreren Empfängern (`ConcreteSubscriber`). Die Empfänger implementieren die Empfänger-Schnittstelle (`Subscriber`), welches eine Methode `update` zur Aktualisierung des Zustandes bereitstellt. Der Sender hält eine Liste von Referenzen auf Empfänger und verfügt über die Methoden `subscribe` und `unsubscribe`, welche es ermöglichen, der Liste Empfänger hinzuzufügen, oder sie zu entfernen (siehe Abbildung 2.5).

Abbildung 2.6 zeigt, dass der Anwender (`Client`)[8] den Zustand des Sender durch Senden von `updateSate` verändern kann (1). Der Sender ruft draufhin auf sich selbst `notify` auf (2) und beginnt über seine Liste von Empfängern zu iterieren. Jedem Empfänger sendet er dann `update` (3, 5) und übergibt den notwendigen Kontext, sodass der Empfänger seinen Zustand entsprechend aktualisieren kann.

[9]Schnittstelle zur Integration fremder Funktionalität in ein bestehendes System

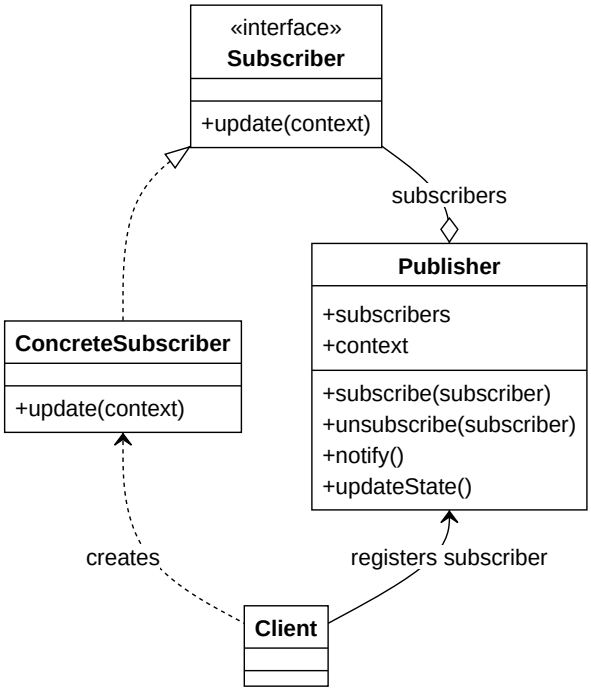

Abbildung 2.5: Klassendiagramm des *Observer Patterns*. `Subscriber` können beim `Publisher` registriert werden, um Nachrichten über Ereignisse zu erhalten. Wird ein Ereignis ausgelöst, so informiert der `Publisher` alle registrierten `Subscriber` darüber. [23]

Konsequenzen

Durch Separation von Sender und Empfänger und durch die Abstraktion der Empfänger-Schnittstelle wird eine lose Kopplung von Sender und Empfänger erreicht. Diese lose Kopplung ermöglicht es, sowohl den Sender, als auch die Empfänger beliebig auszutauschen. Weiterhin können sich der Sender und die Empfänger auf unterschiedlichen Abstraktionsniveaus befinden. Ein Sender auf einem niedrigen Niveau kann einen Empfänger auf einem hohen Niveau benachrichtigen. Wären Empfänger und Sender nicht getrennt, so wäre dafür ein Objekt notwendig, welches mehrere Abstraktionsebenen umfasst, wodurch die Trennung der Abstraktionsschichten beeinträchtigen würde. Ein weiterer Vorteil ist die dynamische Anzahl der Empfänger, mit denen ein Sender interagieren kann. So kann der Anwender über den Sender eine beliebige Zahl an Empfängern erreichen.

Ein Nachteil des *Observer Patterns* ist, dass der Sender stets alle seine Empfänger benachrichtigt. Es kann vorkommen, dass nur eine Teilmenge der Empfänger die Benachrichtigung benötigt, was in unnötigen Methodenaufrufen resultiert. [7]

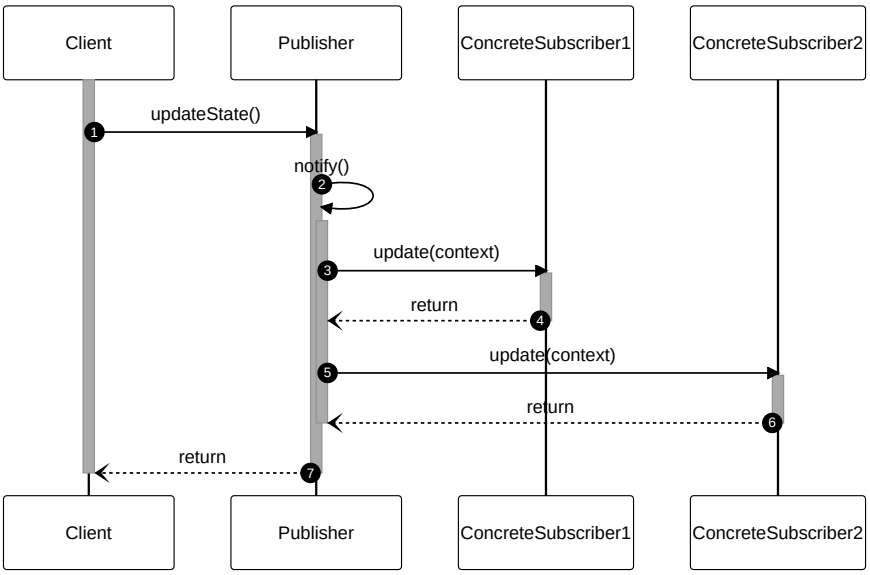

Abbildung 2.6: Sequenzdiagramm des *Observer Patterns*. Nachdem der `Publisher` über ein Ereignis informiert wurde (1), gibt er diese Information durch Selbstaufruf von `notify` (2) an alle registrierten konkreten `Subscriber` weiter (3, 5). [23]

2.3.4 Mediator Pattern

Problembeschreibung

Große Softwareprojekte bestehen meist aus einer großen Anzahl von Klassen bzw. Objekten, welche miteinander interagieren. Ein Ziel ist es, die Kopplung zwischen diesen Komponenten so lose wie möglich zu halten, um die Übersichtlichkeit des Codes zu bewahren. Komplexe Interaktionsmuster zwischen diesen Objekten lassen sich nicht immer verhindern, da sie die inhärente Komplexität des modellierten Problems widerspiegeln. In diesem Fall ist eine Lösung notwendig, die diese Komplexität kapselt. Das *Mediator Pattern* ist in der Lage, solche Fälle von komplexer Interaktion zu vereinfachen. [7]

Lösung

Der konkrete *Mediator* (`ConcreteMediator`) implementiert die *Mediator*-Schnittstelle (`Mediator`), welche die Methode `notify` bereitstellt, um Benachrichtigungen der einzelnen Komponenten (`Component`) entgegenzunehmen. Jede Komponente besitzt eine Referenz auf den *Mediator*, um `notify` an ihn senden zu können. Dabei übergibt die Komponente sich selbst, um dem *Mediator* den Kontext der Benachrichtigung bereitzustellen. In Abhängigkeit von dem übergebenen Sender und dessen Zustand, führt der *Mediator* eine (komplexe) Logik aus, welche vollständig innerhalb des *Mediators* gekapselt ist. Der *Mediator* hält ebenso Referenzen auf alle Komponenten, die er zu beeinflussen in der Lage sein soll. Die gekapselte Logik kann somit die Interfaces der Komponenten verwenden, um diese zu beeinflussen. Dadurch findet eine indirekte Beeinflussung von

Komponenten durch andere Komponenten über den *Mediator* statt. Das Klassendiagramm dieser Architektur ist in Abbildung 2.7 dargestellt.

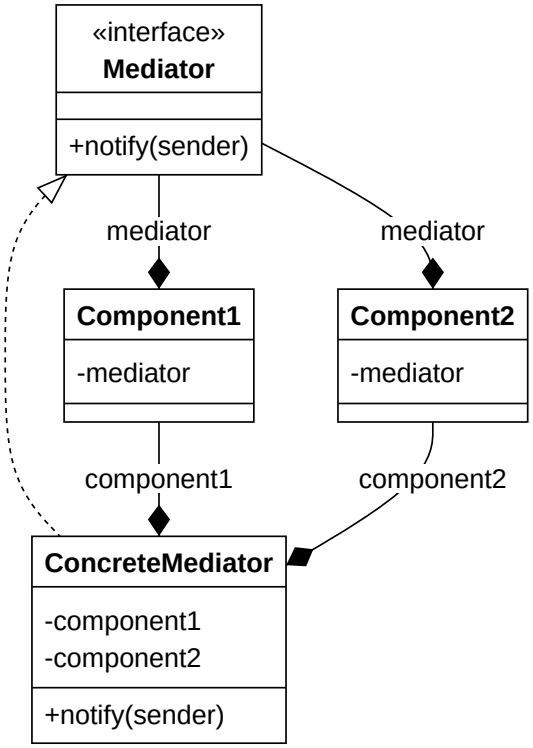

Abbildung 2.7: Klassendiagramm des *Mediator* Patterns. Die Komponenten kommunizieren nur über den konkreten `Mediator` durch Aufruf von `notify` miteinander. Der `Mediator` dirigiert die Interaktion zwischen den Komponenten, zu denen er Referenzen hält. [22]

Konsequenzen

Der *Mediator* kapselt Verhalten, welches ansonsten über mehrere Klassen verteilt wäre. Soll dieses Verhalten spezialisiert werden, so ist nur eine Spezialisierung des *Mediators* notwendig. Weiterhin verhindert der *Mediator* eine starke Kopplung der Komponenten. Komponenten- und *Mediator*-Klassen können bei kompatiblen Interfaces beliebig ausgetauscht werden. Der *Mediator* vereinfacht außerdem die Multiplizitäten von Objektinteraktionen. Er wandelt *m*:*n*- Beziehungen zwischen Objekten in 1:*n*-Beziehungen zwischen den Objekten und dem *Mediator* um.

Der *Mediator* bündelt Kontrolle an einem einzigen Punkt. Dies kann zur Übersichtlichkeit beitragen, kann dieser jedoch bei ausreichend komplexer Logik auch entgegenwirken. Das kann dem *Mediator* eine monolithische Struktur geben, deren Verhinderung seine eigentliche Aufgabe ist.

Sind die Abhängigkeiten zwischen den Komponenten zu komplex, so kann das *Observer Pattern* zur Kommunikation zwischen den Komponenten und dem *Mediator* verwendet

werden. Dadurch lassen sich die Abhängigkeiten außerdem flexibler gestalten, sie können also zur Laufzeit einfacher geändert werden. [7]

2.3.5 Visitor Pattern und Double Dispatch

Problembeschreibung

Gelegentlich muss eine Operation auf einer Menge von Objekten durchgeführt, die alle Teil einer Objekthierarchie aber unterschiedlich sind. Diese Objekte besitzen daher unter Umständen voneinander abweichende Schnittstellen. Das *Visitor Pattern* erlaubt es, solche Operationen außerhalb der Objekte und für alle betroffenen Objekte innerhalb einer separaten Klasse zu definieren. [7]

Lösung

Der konkrete *Visitor*[10] (`ConcreteVisitor`) realisiert die *Visitor*-Schnittstelle (`Visitor`), welche die Methode `visit` bereitstellt, wie in Abbildung 2.8 zu erkennen ist. Diese erlaubt es dem *Visitor*, ein Element (`Element`) zu „besuchen“ und auf ihm eine Operation durchzuführen. Die Elemente „akzeptieren“ den „Besuch“ des *Visitors* mithilfe der Methode `accept`, welche als Argument den besuchenden *Visitor* erhält.

Abbildung 2.9 zeigt das Verhalten des *Visitor Patterns*. Der Anwender[8] trägt dem *Visitor* auf, eine Operation auf einem Element oder einer Menge von Elementen durchzuführen (1). Der *Visitor* sendet daraufhin `visit` an alle Elemente, auf die er eine Referenz hält (2). Jedes Element ruft daraufhin eine `accept`-Methode auf dem *Visitor* auf (3). Zu beachten ist hierbei, dass es für jede Element-Klasse eine eigene Methode im *Visitor* gibt. Dies kann durch das Bereitstellen von Methoden mit unterschiedlichen, zu den aufrufenden Klassen korrespondierenden, Namen oder durch Multimethoden realisiert werden. Multimethoden sind Methoden, welche je nach Typ der übergebenen Argumente unterschiedliche Implementierungen ausführen. Somit kann der *Visitor* nach Erhalt von `accept` die zum Typ des sendenden Elements passende Operation ausführen. Dieser Mechanismus nennt sich *Double Dispatch*.

Konsequenzen

Durch die Kapselung der Operation in einem *Visitor*, ist es sehr einfach, neue Operationen hinzuzufügen. Es bedarf dazu lediglich eines weiteren *Visitors*. Außerdem kapselt ein *Visitor* die Menge an Operationen auf den Elementen. Zusammengehörige Operationen werden in einer Klasse gesammelt. Nicht zueinander gehörende Operationen befinden sich in unterschiedlichen *Visitors*. Ein weiterer Vorteil eines *Visitors* ist dessen Fähigkeit, während des „Besuchens“ mehrerer Elemente Informationen über diese zu akkumulieren und im Anschluss gebündelt zu repräsentieren.

Der *Visitor* weist jedoch auch Nachteile auf. Zum einen ist es schwer, weitere konkrete Element-Klassen zu einem System hinzuzufügen, welches bereits eine Reihe an *Visitors* besitzt. Da ein *Visitor* für jeden Typ von Element eine Methode bereitstellen muss, kann ein weiteres Element einen erhöhten Implementierungsaufwand bedeuten. Das *Visitor*

[10] Die deutsche Übersetzung „Besucher“ ist in diesem Kontext eher unüblich.

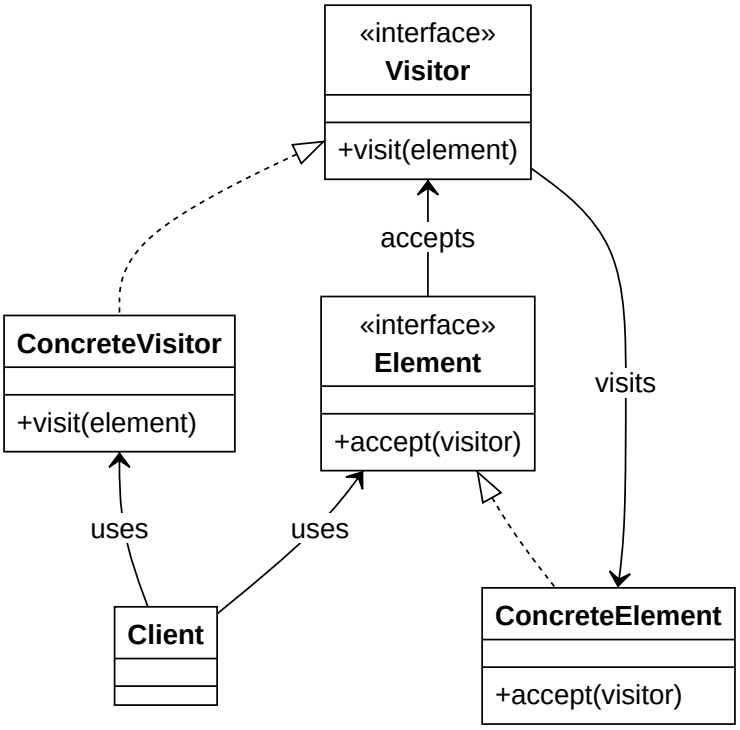

Abbildung 2.8: Klassendiagramm des *Visitor Patterns*. Der `Visitor` kapselt eine Operation auf einer Menge von Elementen in seinen `visit`-Methoden. Die Elemente werden von dem `Visitor` „besucht“ und rufen die zu ihrem Typ korrespondierende Methode auf dem Visitor auf. [26]

Pattern sollte daher nur verwendet werden, wenn entweder die Menge an Elementklassen abgeschlossen oder die Menge an *Visitor*-Klassen übersichtlich ist. Zum anderen müssen die Elemente dem *Visitor* eine Schnittstelle bereitstellen, welche es dem *Visitor* ermöglicht, seine Operation ausführen zu können. Dies kann dazu führen, dass das Element einen großen Teil seines internen Zustands preisgeben muss, welcher bei nicht-Verwendung dieses Musters gekapselt geblieben wäre. [7]

2.3.6 Factory Method

Problembeschreibung

Es wird eine Schnittstelle benötigt, um eine Reihe von Objekten erzeugen zu können. Jedes Objekt hat jedoch andere Anforderungen an seine Erzeugung. Eine *Factory-Method* kann eingesetzt werden, wenn eine Klasse kein Wissen darüber besitzt oder besitzen soll, welches konkrete Objekt sie zu erzeugen hat oder wenn eine Klasse die Verantwortlichkeit über diese Entscheidung ihren Subklassen überlassen soll. [7]

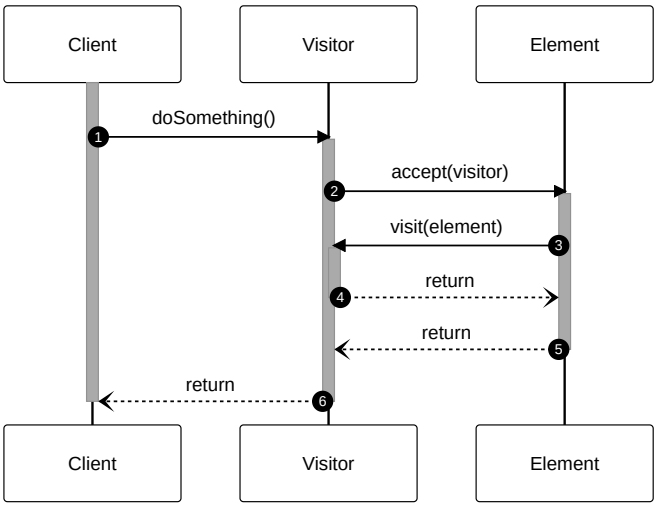

Abbildung 2.9: Sequenzdiagramm des *Visitor Patterns*. Das Entwurfsmuster nutzt einen *Double Dispatch*, um die Ausführung der korrekten Methode im `Visitor`-Objekt zu gewährleisten. Dazu ruft der Visitor auf dem Element die Methode `accept` auf (2). Diese ruft dann die zum Element passende `visit`-Methode auf (3). [26]

Lösung

Es existiert ein abstrakter Erzeuger (`Creator`), welcher eine Schnittstelle bereitstellt, um Produkte (`Product`) zu erzeugen. Die Details der Erzeugung dieser Produkte sind in den konkreten Subklassen der Erzeuger-Klasse implementiert. Jeder konkrete Erzeuger kann somit einen Typ von konkretem Produkt (`ConcreteProduct`) erschaffen. Das entsprechende Klassendiagramm ist in Abbildung 2.10 dargestellt.

Konsequenzen

Ein Objekt über eine *Factory-Method* zu erzeugen ist flexibler, als das Objekt direkt über den Konstruktor der Klasse zu instanziieren. Die erzeugende Klasse braucht nur das Interface des abstrakten Erzeugers zu kennen und ist somit in der Lage beliebige konkrete Produkte über deren korrespondierende konkrete Erzeuger zu instanziieren. Hierbei fällt auf, dass die Erzeuger-Klassenhierarchie die Produkt-Klassenhierarchie spiegelt. Für jeden Produkttyp existiert also auch eine Erzeuger-Klasse. Daraus kann sich jedoch auch ein Nachteil ergeben. Zur Nutzung eines Produktes müssen nun stets zwei Subklassen definiert und zur Laufzeit ein weiteres Objekt erstellt werden. Das erhöht die Komplexität. [7]

2.4 Object-Relational-Mapping

Für die dauerhafte Speicherung von Daten wird eine Datenbank benötigt. Die Interaktion von objektorientiertem Quelltext mit einer relationalen Datenbank kann den Program-

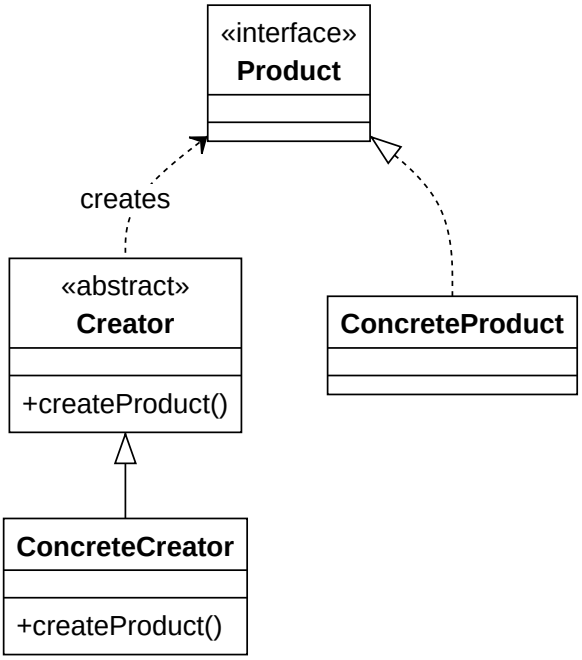

Abbildung 2.10: Klassendiagramm des *Factory-Method*-Musters. Für jedes konkrete Produkt existiert ein konkreter *Creator*, welcher eine *Factory Method* `createProduct` besitzt. Diese erzeugt das korrespondierende Produkt. Dabei spiegelt die Vererbungshierarchie der *Creator* die der Produkte. [21]

mierer vor Herausforungen stellen. Diese und mögliche Lösungen werden in diesem Abschnitt vorgestellt.

2.4.1 Objekt-relationale Systeme

In der Praxis haben sich für die Beschreibung von Datenbanken und Software vor allem zwei verschiedene Paradigmen etabliert. Datenbanken, welche der persistenten Speicherung von Daten dienen, folgen häufig dem relationalen Paradigma. Softwaresysteme hingegen, deren Verhalten und Struktur durch Programmiersprachen beschrieben werden, werden meist objektorientiert modelliert. Da diese beiden Technologien, breite Verwendung finden, ist ihre Kombination innerhalb desselben Systems in der häufig unumgänglich. Solche Systeme nennt man objekt-relationale Systeme. [12]

2.4.2 Object-Relational Mapper (ORM)

Die Aufgabe eines ORM ist es, die Persistenz von Objekten zu gewährleisten [30] und dabei gleichzeitig die darunterliegende relationale Datenbank zu abstrahieren. So kann der Programmierer, welcher eine objektorientierte Programmiersprache verwendet, vollständig in einem objektorientierten Kontext arbeiten. Er ist nicht mehr gezwungen, direkt mit der relationalen Datenbank über *SQL* zu kommunizieren. Das ORM stellt dabei eine

bidirektionale Abbildung zwischen dem objektorientierten und dem relationalen Modell bereit. Dabei muss sichergestellt werden, dass sowohl die Struktur, als auch die Mechanismen beider Paradigmen korrekt aufeinander abgebildet werden. Die Abbildung der Struktur beschäftigt sich mit der Zuordnung von Tabellen zu Klassen. Die Abbildung der Mechanismen behandelt unter anderem die Navigation durch Objektreferenzen und das Schreiben und Lesen von Daten. [12]

2.4.3 Hindernisse

Die Unterschiede zwischen beiden Paradigmen rufen eine Reihe von Problemen hervor, welche das ORM lösen muss. Diese Probleme sind nach [12] die folgenden:

- Es muss eine Abbildung zwischen den Strukturen beider Paradigmen gefunden werden. Eine relationale Datenbank unterstützt weder Klassenhierarchien noch Spalten, die mehrere Elemente beinhalten können.
- Zeilen innerhalb einer Relation haben eine festgeschriebene Struktur. Objekte hingegen können eine dynamische Struktur besitzen. Die Frage ist, wie sich die Objektstruktur in der Datenbank abbilden lässt.
- Während der Zustand eines Objektes durch Kapselung geschützt ist, sind alle Daten einer relationalen Datenbank öffentlich. Das Problem besteht in der Abbildung dieser Kapselung.
- „Identität“ hat innerhalb der beiden Paradigmen unterschiedliche Bedeutungen. Objekte gelten als identisch, wenn sie die gleiche Speicheradresse besitzen. Identische Zeilen hingegen zeichnen sich durch einen gleichen Primärschlüssel aus. Das kann zu Problemen führen, wenn zwei nicht-identische Objekte mit gleichem Primärschlüssel erzeugt werden.
- Referenzen in objektorientierten und relationalen Modellen besitzen unterschiedliche Richtungen. Entsprechend muss die Navigation durch die Modelle abgebildet werden.
- In der Praxis kann es vorkommen, dass die Datenbank und die darauf aufbauende Software von unterschiedlichen Teams gepflegt werden. Außerdem besteht die Möglichkeit, dass eine Datenbank von mehreren Softwaresystemen genutzt wird. Dabei gilt es, eine einheitliche Kommunikation zu gewährleisten.

Jedes ORM konzentriert sich auf die Aspekte der zu leistenden Abbildung unterschiedlich stark. So kann der Fokus beispielsweise mehr auf der strukturellen Abbildung zwischen Klassen und Tabellen liegen oder auf der korrekten Abbildung der Mechanismen. Es gibt daher keine einheitliche oder richtige Lösung. Entsprechend ist die Trennung zwischen objektorientiertem und relationalem Paradigma nie vollständig und hängt vom Einzelfall und den Bedürfnissen des Anwenders ab.

2.4.4 peewee

Für unser Softwareprojekt haben wir das ORM *peewee*[11] [14] ausgewählt. Da das ORM einer der Grundsteine unseres Projektes darstellt, stand die Auswahl des ORM am An-

[11] `https://github.com/coleifer/peewee`

fang Planungsprozesses. Daher war es uns nicht möglich, das ORM anhand der Anforderungen unserer Softwarearchitektur auszuwählen. Diese stand zu diesem Zeitpunkt, aufgrund der agilen Arbeitsweise unseres Teams, noch nicht in ausreichendem Detailgrad zur Verfügung. Wir haben die Entscheidung stattdessen anhand der allgemeinen Funktionsübersicht und der Benutzerfreundlichkeit getroffen. Zur Auswahl stand alternativ das ORM *SQLAlchemy* [28]. Obwohl *SQLAlchemy* einen größeren Funktionsumfang bietet, ist *peewee* einfacher zu bedienen und man erreicht viele der meistbenutzten Funktionen mit weniger Code. Da dies zugunsten der Entwicklungsgeschwindigkeit geht, haben wir uns letztendlich für *peewee* entschieden. Außerdem sinkt die Wahrscheinlichkeit für das Begehen von Fehlern mit weniger Code.

Im Folgenden wird eine Reihe von Grundfunktionalitäten von *peewee* vorgestellt, welche in unserem Projekt häufig Verwendung fanden und die auch im Rahmen dieser Arbeit eine Rolle spielen.

Definition von Modellen

Als ein Modell wird im Kontext von *peewee* eine Klasse bezeichnet, deren Objekte sich in der verknüpften Datenbank ablegen lassen. Wie in Quelltext 2.1 zu erkennen ist, lassen sich Modelle definieren, indem die betroffene Klasse von `peewee.Model` erbt. Die Verknüpfung zur Datenbank lässt sich durch die Definition des Feldes `db` in der Klasse `Meta` herstellen. Diese Verknüpfung muss innerhalb einer Klassenhierarchie nur einmal in der obersten Superklasse definiert werden.

Quelltext 2.1: Python-Code zur Definition eines *peewee*-Modells. Es modelliert eine Person mit einem Namen und einem Geburtstag. Weiterhin besitzt die Klasse das nicht-persistente Feld `age`. [18]

```
class Person(Model):
    class Meta:
        db = SqliteDatabase('people.db')

    name = CharField()
    birthday = DateField()
    age: int
```

Der Klasse können dann beliebige Attribute und auch Methoden zugewiesen werden. Soll ein Attribut persistent sein, soll es also in der Datenbank abgelegt werden, so muss ihm in der Klassendefinition ein Feld vom Typ `peewee.Field` zugewiesen werden. Über den Subtyp dieses Feldes wird der Typ der korrespondierenden Spalte in der Datenbanktabelle festgelegt. Andere Attribute, wie `age` in Quelltext 2.1 sind Teil des Objektes, aber nicht persistent.

Speichern von Objekten

Objekte lassen sich auf mittels zwei verschiedener Methoden in der Datenbank ablegen. Jede Modell-Klasse erbt die Methode `create` von `peewee.Model`. Diese nimmt die gleichen Argumente entgegen wie der eigentlich Konstruktor der Klasse. Wird `create` aufgerufen, so wird eine Instanz der Klasse erstellt, in die Datenbank geschrieben und im Anschluss zurückgegeben, wie in Quelltext 2.2 ersichtlich wird.

Quelltext 2.2: Python-Code zum Erzeugen eines Personen-Objektes und zur Speicherung in der Datenbank in einem Schritt. [18]

```
uncle_bob = Person.create(name='Bob', birthday=date(1960, 1, 15))
```

Alternativ dazu kann das Objekt auch direkt über den Konstruktor erzeugt werden, wie in Quelltext 2.3 zu erkennen. Durch anschließenden Aufruf von `save` wird das Objekt dann in die Datenbank gelegt. Diese Methode bietet den Vorteil, dass das Objekt nicht zwangsläufig gespeichert werden muss. Auch erfüllt `save` die Funktion der Aktualisierung eines Objektes in der Datenbank, nachdem ein bereits existierendes Objekt verändert wurde.

Quelltext 2.3: Python-Code zum Erzeugen eines Personen-Objektes und zur anschließenden Speicherung in der Datenbank. [18]

```
uncle_bob = Person(name='Bob', birthday=date(1960, 1, 15))
uncle_bob.save()
```

Datenabfragen

Das Abfragen von Daten orientiert sich in *peewee* an den Konzepten von SQL, bildet die *SQL*-Schlüsselwörter jedoch auch Methoden ab. Quelltext 2.4 zeigt eine einfache Datenabfrage die zu dem *SQL*-Statement in Quelltext 2.5 äquivalent ist.

Quelltext 2.4: Python-Code zum Abfragen eines Objektes aus der Datenbank. [18]

```
grandma = Person.select().where(Person.name == 'Grandma .L').get()
```

Quelltext 2.5: *SQL*-Abfrage mit SELECT und WHERE.

```
SELECT * FROM Person WHERE name = 'Grandma L.';
```

2.5 Die Smard-API

Für die Ermittlung von historischen Energieproduktionsdaten, auf welchen die von uns verwendeten Kohlebedarfsdaten basieren, haben wir die *Smard-API*[12] der Bundesnetzagentur verwendet [3]. Sie liefert die Menge produzierten Stromes in Abhängigkeit von der Zeit. Die Daten werden in verschiedenen zeitlichen Auflösungen, von Jahresdaten bis hin zu 15-Minuten-Intervallen, bereitgestellt. Räumlich lassen sich die Daten bis auf eine Regelzone[13] eingrenzen. Weiterhin lassen sich die Daten nach dem Energieträger filtern. So besteht die Möglichkeit, Daten speziell für Braunkohle zu erhalten.

Bei der *Smard-API* handelt es sich um eine statische API. Das bedeutet, die Antwort auf eine Anfrage wird nicht für jeden Benutzer individuell zusammengestellt. Stattdessen existiert eine Menge von JSON-Dateien, von denen je nach Anfrage eine ausgeliefert wird.

[12] `https://github.com/bundesAPI/smard-api`

[13] „Als Regelzone wird ein räumlich abgegrenztes Netzgebiet bezeichnet für das ein Übertragungsnetzbetreiber verantwortlich ist.“ [27]

Die API stellt zwei Arten von Dateien zur Verfügung. Für jede Kombination aus zeitlicher und räumlicher Auflösung und Filter gibt es eine Index-Datei. Diese beinhaltet eine Liste von Zeitstempeln. Jeder Zeitstempel ist mit einer Daten-Datei assoziiert. Durch Verwendung des Zeitstempels kann die zugehörige Daten-Datei heruntergeladen werden. Diese enthält eine Liste von Paaren aus Zeitpunkt und produzierter Energiemenge in MWh. Die Paare sind aufsteigend nach Zeitpunkten sortiert und der erste Zeitpunkt entspricht dem verwendeten Zeitstempel.

3 Entwurf und Ergebnisse

Der Hauptteil beschreibt die Umsetzung der in der Einleitung definierten Anforderungen. Es werden die Berechnung des Kohlebedarfs, die Softwarearchitektur und einige Implementierungsdetails der Software vorgestellt. Abschließend werden die Ergebnisse präsentiert, welche durch das entwickelte System entstanden sind.

3.1 Die Berechnung des Kohlebedarfs

Im Folgenden wird erläutert, wie aus Stromproduktionsdaten der Kohlebedarf für ein Kraftwerk berechnet wird. Außerdem wird erklärt, wie daraus Abfahrtszeiten für Kohlezüge bestimmt werden können. Viele Informationen dieses Abschnittes, welche mit dem Kohleverkehr in der Lausitz in Zusammenhang stehen, stammen aus einem Interview mit Sascha Lesche von der LEAG (siehe Anhang). Abbildung 3.1 zeigt den Ablauf der Berechnung.

Die Bundesnetzagentur stellt Kenndaten zu allen Kraftwerken Deutschlands bereit [4]. Die für uns wichtigen Kraftwerksdaten sind der Name des Kraftwerkes, seine elektrische Leistung in MW, die thermische Leistung in MW, der verwendete Energieträger und das Bundesland, ich welchem sich das Kraftwerk befindet. Diese Daten wurden einmalig in Form einer CSV-Datei heruntergeladen. Im Anschluss wurden die Daten gefiltert, sodass nur noch Kraftwerke mit dem Energieträger Braunkohle innerhalb der *50Hertz*-Regelzone übrige blieben. Diese Zone umfasst die neuen Bundesländer und Hamburg. Aus diesen gefilterten Kraftwerksdaten wurde dann die Gesamtleistung P_T aller Braunkohlekraftwerke in der Regelzone durch Aufsummieren berechnet. Diese Gesamtleistung beträgt rund 10.600 MW. Außerdem wurden für die weitere Berechnung nur noch die drei Lausitzer Kraftwerke betrachtet, welche über das Schienennetz der LEAG mit Braunkohle versorgt werden. Diese Kraftwerke sind *Boxberg*, *Jänschwalde* und *Schwarze Pumpe*.

Die produzierte Energiemenge E aus Braunkohle in der *50Hertz*-Regelzone für einen bestimmten Zeitraum kann von der *Smard-API* abgerufen werden. Unter der Annahme, dass alle Kraftwerke dieser Zone gleichmäßig ausgelastet werden, benötigen sie eine Zeitspanne von Δt, um die Energiemenge E zu produzieren. Δt ergibt sich, wie in Gleichung 3.1 dargestellt, aus dem Quotienten aus der produzierten Energie und der Gesamtleistung aller Kraftwerke.

$$\Delta t = \frac{E}{P_T} \tag{3.1}$$

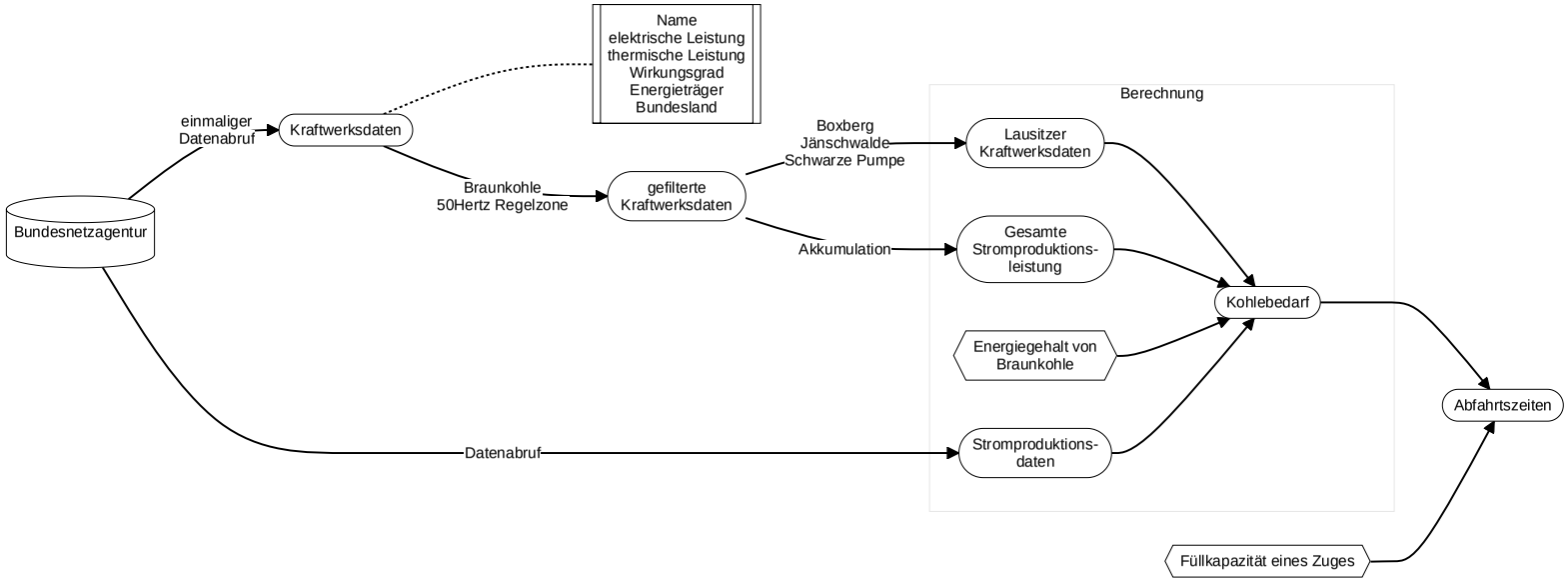

Abbildung 3.1: Schematische Abbildung der Berechnung des Kohlebedarfs eines Kraftwerkes und der benötigten Anzahl an Kohlezügen aus Stromproduktionsdaten und Kennzahlen von Kraftwerken. Der Zylinder stellt eine Datenquelle dar, die abgerundeten Felder Daten und die Hexagone Konstanten. Die Bundesnetzagentur stellt Daten über historische Stromproduktionsdaten und über Kraftwerke bereit. Aus diesen kann der Kohlebedarf berechnet werden, auf welchem die Abfahrtszeiten der Züge basieren.

Die benötigte Kohlemenge m_k, die ein Kraftwerk benötigt, um Δt lang bei voller Leistung Strom zu produzieren, kann mithilfe von Gleichung 3.2 berechnet werden. Dabei sind P_e die elektrische Leistung und P_t die thermische Leistung des betrachteten Kraftwerkes in MW. Weiterhin sind ρ die mittlere Energiedichte von Braunkohle mit 2,46 MWh/t[14] und η der Wirkungsgrad des Kraftwerkes.

$$m_k = \frac{\Delta t(P_e + P_t)}{\rho\eta} \tag{3.2}$$

Zusammen mit der Füllkapazität eines Kohlezuges c von 960 t lässt sich durch Gleichung 3.3 die Anzahl der Züge n_z berechnen, die benötigt wird, um die Masse an Kohle zu transportieren.

$$n_z = \frac{m_k}{c} \tag{3.3}$$

Die Anzahl an Zügen kann dann über die Zeit akkumuliert werden, sodass immer, wenn dieser Wert eine ganze Zahl überschreitet, ein Zug abfahren kann.

Die für diese Berechnung verwendeten Annahmen sind die Folgenden:

- Alle Kraftwerke der *50Hertz*-Regelzone werden gleichmäßig (gewichtet mit ihrer jeweiligen Maximalleistung) ausgelastet.
- Das Verhältnis zwischen elektrischer Leistung P_e und thermischer Leistung P_t eines Kraftwerkes ist konstant.
- Der Wirkungsgrad eines Kraftwerkes ist eine Konstante und nicht abhängig von der momentan bereitgestellten Leistung.

Diese Annahmen mussten getroffen werden, da keine Daten mit höherer räumlicher Auflösung zur Verfügung standen und das Leistungsverhalten eines Kohlekraftwerkes für unsere Zwecke vereinfacht wurde. Die Güte bzw. der Realismus dieser Annahmen wird sich letztendlich aus dem Vergleich der Simulationsergebnisse mit Referenzdaten ergeben.

3.2 Architektur

In diesem Abschnitt wird die Architektur der Datenbank, der Zugerzeugung und des Eventbusses diskutiert. Anhand von Klassen- und Sequenzdiagrammen wird die Funktionsweise der einzelnen Komponenten erläutert und Entwurfsentscheidungen werden begründet.

3.2.1 Die Datenbank

Das in unserem System verwendete Datenbank-Management-System ist *Postgres*. Es läuft in einem eigenen *Docker*-Container. Als Datenbankschnittstelle verwendeten wir das ORM *peewee*, welches Tabellen einer relationalen Datenbank auf sogenannte Modell-Klassen abbildet. Da jede unserer Modell-Klassen weitere Funktionalität benötigt, haben wir eine

[14] Die Energiedichte der Rohbraunkohle in der Lausitz schwankt zwischen 2,14 und 2,78 MWh/t. [5]

abstrakte Modell-Klasse (`BaseModel`) definiert, welche als Superklasse für jede weitere Modellklasse dient, und welche selbst von `peewee.Model` erbt, wie in Abbildung 3.2 zu erkennen ist. Diese Funktionalitäten sind

- die explizite Definition eines Primärschlüssels (`id`) mit dem Typ einer Zeichenkette, welcher einen *Universally unique identifier* (UUID) enthält,
- Daten, wann ein Objekt erzeugt und wann es zuletzt aktualisiert wurde
- und einen menschen-lesbaren Bezeichner (`readable_id`), welcher prozedural erzeugt wird und einer verbesserten Nutzerinteraktion dient.

Weiterhin wurde die Methode `save` überschrieben aus `peewee.Model` überschrieben, um darin den Feldern `created_at`, `updated_at` und `readable_id` ihre Werte zuzuweisen.

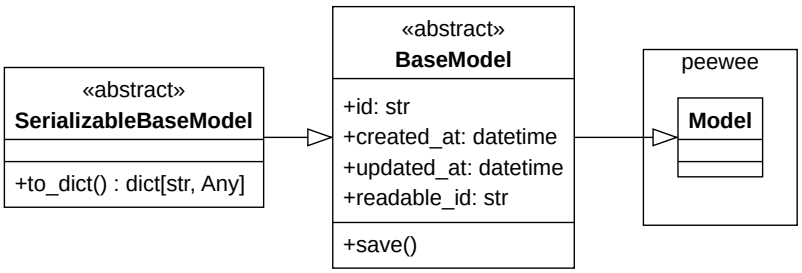

Abbildung 3.2: Klassendiagramm der abstrakten Modell-Klassen des ORM. `BaseModel` erbt von `peewee.Model`, um weitere Funktionalität hinzuzufügen. Dazu gehören Felder für `datetime`-Objekte, welche angeben, wann das Objekt erstellt und aktualisiert wurde. Weiterhin besitzt jedes Objekt eine `readable_id`, welche die Identifizierung durch einen Menschen vereinfachen soll. Davon erbt `SerializableBaseModel`, um die Serialisierung von Objekten zu ermöglichen.

Von `BaseModel` erbt `SerializableBaseModel`, welches die zusätzliche Methode `to_dict` implementiert. Damit lässt sich das Objekt JSON-serialisieren, um es bei Bedarf an das *Frontend* der Anwendung senden zu können.

Da die hinzugefügten Funktionalitäten in der überschriebenen Methode `save` implementiert wurden, sind diese gut in das Modell-System von *peewee* integriert. Da `save` auch in `create` aufgerufen wird, wie in Abbildung 3.3 zu erkennen, ist die Funktionalität damit automatisch auch bei der direkten Erzeugung eines neuen Objekts gegeben.

Alle im Rahmen unseres Projektes definierten Modell-Klassen erben von `BaseModel` oder `SerializableBaseModel` und erhalten damit die hinzugefügten Attribute.

3.2.2 Der Spawner

Das zentrale Element der Zugerzeugung ist die Klasse `Spawner`. Die Architektur aller an der Zugerzeugung beteiligten Klassen ist in Abbildung 3.4 dargestellt. Die Klasse `Spawner` erbt, wie die anderen zentralen Klassen des Systems, von der Klasse `Component` und erhält damit Zugriff auf den `EventBus`, welcher in einem folgenden Abschnitt beschrieben wird, und auf die Methode `next_tick`. Diese Methode wird in jedem Zeitintervall der Simulation einmal aufgerufen und gibt der Komponente die Möglichkeit, auf die Simulation zu

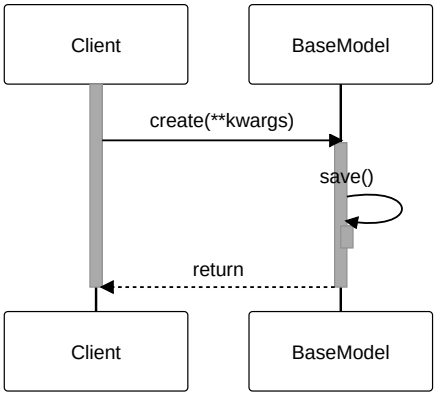

Abbildung 3.3: Sequenzdiagramm der Erstellung eines Objektes über die Methode `create`. In `create` wird sowohl das Objekt initialisiert, als auch Selbstaufruf von `save` in der Datenbank abgelegt.

reagieren. Weiterhin realisiert die Klasse `Spawner` die `ISpawnerDisruptor`-Schnittstelle, welche eine Fehlerinjektion [17] ermöglicht. Der `Spawner` besitzt die Attribute `configuration`, welches im Folgenden Abschnitt näher beleuchtet wird, und `schedules`, bei dem es sich um eine Liste Abfahrtsplänen (`Schedule`-Objekte) handelt.

Die Klasse `Schedule` ist abstrakt und bietet damit die Möglichkeit der Spezialisierung. Die einzige bisher implementierte Spezialisierung ist der `TrainSchedule`, welcher speziell für die Erzeugung von Zügen verantwortlich ist. Wir haben uns für eine abstrakte `Schedule`-Klasse entschieden, um die Möglichkeit zu haben, das System in Zukunft um Abfahrtspläne anderer Verkehrsteilnehmer zu erweitern. Denkbar wäre beispielsweise die Simulation von Fußgängern, die gleichzeitig als Passagiere für Personenzüge fungieren. Diese Funktionalität könnte in einer Klasse `PedestrianSchedule` implementiert werden, welche ebenfalls von `Schedule` erbt. Eine Instanz eines `TrainSchedule` Attribute, welche den Zugtyp (`train_type`) und die Liste der anzufahrenden Haltestellen (`platform_ids`) enthalten.

Um eine zukünftige Erweiterbarkeit zu gewährleisten, wurde die Klasse `Schedule` so implementiert, dass der Algorithmus, welcher die Abfahrzeiten bestimmt, von der eigentlichen `Schedule`-Klasse unabhängig ist. Ein `Schedule`-Objekt hält eine Referenz auf ein `ScheduleStrategy`-Objekt, welches einen Algorithmus zur Bestimmung der Abfahrtszeiten implementiert. Diese Klasse stellt eine Start- und eine Endzeit (`start_time` und `end_time`) bereit, welche die Zeitspanne festlegen, in der die Abfahrtszeiten bestimmt werden. Mit der Methode `should_spawn` kann für eine bestimmte Zeit abgefragt werden, ob ein Zug zu erzeugen ist. Von der abstrakten Klasse `ScheduleStrategy` erben die Klassen `RegularScheduleStrategy`, `RandomScheduleStrategy` und `DemandScheduleStrategy`, welche das Verhalten der drei zuvor beschriebenen Abfahrtspläne implementieren.

Die Klasse `RegularScheduleStrategy` implementiert die regulären Abfahrtspläne und besitzt das Attribut `frequency`, welches die Häufigkeit der regelmäßig abfahrenden Züge angibt. Die `RandomScheduleStrategy`-Klasse ist für die randomisierten Abfahrtspläne

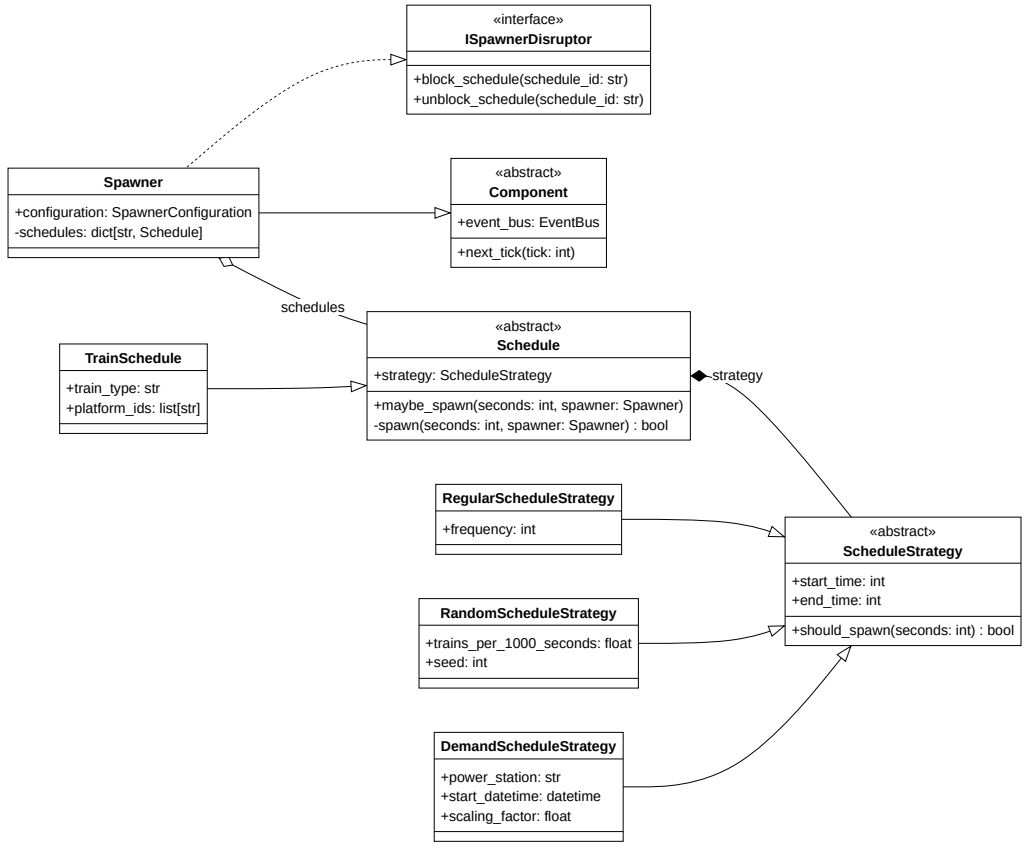

Abbildung 3.4: Klassendiagramm der Zugerzeugung. Der `Spawner` erbt von `Component`, um in jedem *Tick* der Simulation benachrichtigt werden zu können und Zugriff auf den Eventbus zu erhalten. Außerdem realisiert er die Schnittstelle `ISpawnerDisruptor`, um eine Beeinflussung durch die Fehlerinjektion zu ermöglichen. Der `Spawner` besitzt eine Liste von `Schedule`-Objekten, welche die Abfahrtspläne repräsentieren. Jedes `Schedule`-Objekt besitzt eine Referenz auf ein `ScheduleStrategy`-Objekt, welches den Algorithmus zur Bestimmung der Abfahrtszeiten implementiert.

zuständig und besitzt dafür die Attribute `trains_per_1000_seconds` und `seed`, welche die Wahrscheinlichkeit einer Abfahrt und einen Startwert für den Zufallszahlengenerator beinhalten. Letzteres Attribut dient der Reproduzierbarkeit von Simulationsergebnissen. Die `DemandScheduleStrategy`-Klasse implementiert die Abfahrtspläne, welche auf dem Kohlebedarf basieren. Die hält die Attribute `power_station` für das betrachtete Kraftwerk, `start_datetime` für das Startdatum der historischen Datenbasis und `scaling_factor`, womit der Kohlebedarf skaliert werden kann, um beispielsweise eine unvollständige Auslastung eines Kraftwerks zu simulieren.

Abbildung 3.5 zeigt das Verhalten der Zugerzeugung innerhalb eines Zeitintervalls der Simulation. Ein `Communicator`-Objekt[15] sendet `next_tick` an das `Spawner`-Objekt (1). Dieser sendet daraufhin `maybe_spawn` an jedes Element in der Liste `schedules` und

[15] beschrieben in der Arbeit von Kamp [13]

übergibt sich dabei selbst (2). Durch Aufruf von `should_spawn` wird überprüft, ob ein Zug zu erzeugen ist (3). Ist dies der Fall, ruft das `Schedule`-Objekt auf sich selbst `spawn` auf (5), was dazu führt, dass `spawn_train` an die übergebene `Spawner`-Instanz gesendet wird (6). Dort wird letztendlich die Schnittstelle zu *SUMO* angesprochen, um einen Zug in die Simulation zu setzen.

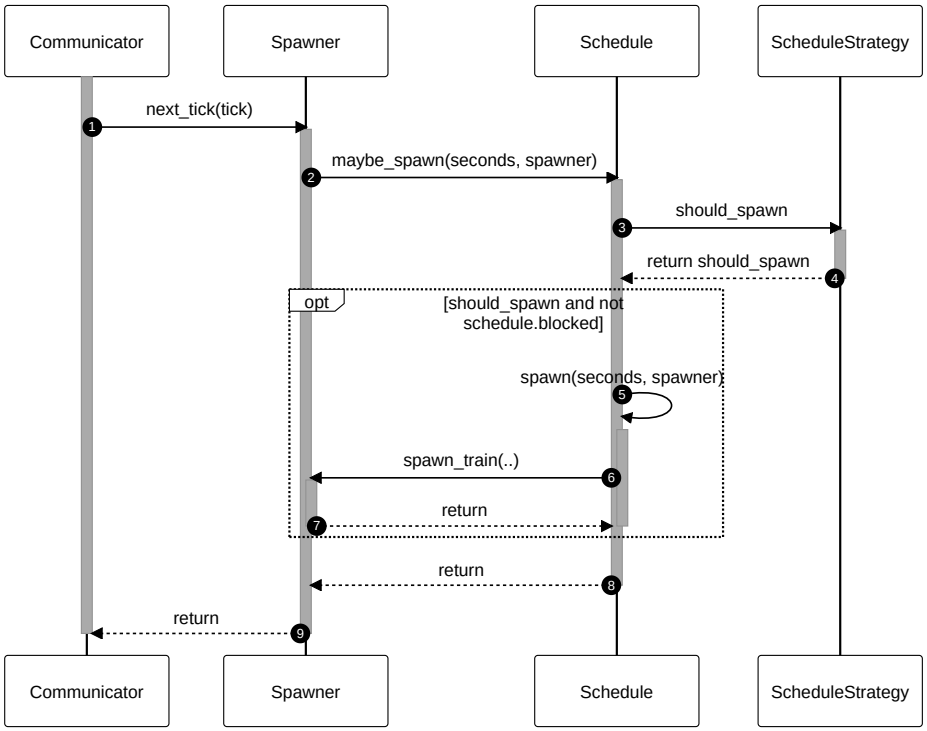

Abbildung 3.5: Sequenzdiagramm der Zugerzeugung. Zu jedem Simulations-*Tick* (1), veranlasst der `Spawner` jedes seiner `Schedule`-Objekte, zu prüfen, ob es einen Zug zu erzeugen gibt (2). Die `Schedule`-Objekte führen dazu den Algorithmus aus, welcher in der `ScheduleStrategy`-Instanz implementiert ist (3). Gibt der Algorithmus ein positives Ergebnis zurück und wurde das `Schedule`-Objekt nicht durch die Fehlerinjektion blockiert, wird die Erzeugung eines Zuges veranlasst (5, 6).

Bei diesem Mechanismus finden zwei Entwurfsmuster Anwendung. Der `Spawner` fungiert and *Visitor* und besucht jedes Element in der Liste von `Schedule`-Objekten. Über einen *Double-Dispatch* mit den Methoden `maybe_spawn` und `spawn_train` wird dabei erreicht, dass die Verantwortlichkeit der Implementierung zur Erzeugung von Zügen (und zukünftig evtl. weiterer Verkehrsteilnehmer) bei der Klasse `Spawner` liegt. Die Entscheidung hingegen wird von den `Schedule`-Objekten getroffen. Dadurch lassen sich in Zukunft relativ einfach weitere Arten von `Schedule`-Klassen hinzufügen. Die Klasse `Spawner` muss dazu nur um jeweils eine weitere Methode erweitert werden. Bei der Methode `maybe_spawn` handelt es sich weiterhin um eine im abstrakten `Schedule` implementierte *Template-Method*. Sie gibt den Ablauf der Entscheidung über die Zugerzeugug vor. Die konkreten Implementierungen für `spawn` und `should_spawn` liegen jedoch in den

konkreten Subklassen von `Schedule` bzw. `ScheduleStrategy`. Auch hier ist der Vorteil, dass sich neue Algorithmen zur Bestimmung der Abfahrtszeiten relativ einfach hinzufügen lassen, ohne dass die Klasse `Spawner` angepasst werden muss.

3.2.3 Die Konfiguration des Spawners

Die Konfiguration des *Spawners* legt fest, wann und wo in der Simulation Züge erzeugt werden. Eine der Anforderungen an das System ist, dass der Benutzer in der Lage sein muss, diese Konfiguration durchzuführen. Weiterhin ist es wünschenswert, dass die Konfigurationen gespeichert werden können, um die zugehörigen Simulationen erneut oder in abgeänderter Form durchführen zu können. Zu Beginn des Projektes erbten die Klassen `Spawner`, `Schedule` und `ScheduleStrategy` von `BaseModel`. Somit ließen sich Instanzen dieser Klassen und ihrer Subklassen direkt in die Datenbank schreiben. Dieser Ansatz wurde jedoch verworfen, da das verwendete ORM *peewee* nicht in der Lage ist, objektorientierte Vererbungshierarchien ausreichend gut abzubilden. *peewee* ordnet jeder Klasse eine Tabelle in der Datenbank zu. Es spielt dabei keine Rolle, ob es sich um eine abstrakte oder eine konkrete Klasse handelt. Ein `Schedule`-Objekt kann beispielsweise ein `RegularScheduleStrategy`-Objekt enthalten. Die Klasse `Schedule` hat jedoch nur Kenntnis von der abstrakten Klasse `ScheduleStrategy`. Die Abbildung durch *peewee* erzeugt daher für die Tabelle `Schedule` lediglich einen Fremdschlüssel, der auf die „abstrakte“ Tabelle `ScheduleStrategy` verweist. Referenzen auf abstrakte Klassen lassen sich also mit *peewee* nicht abbilden.

Wir haben uns stattdessen für die in Abbildung 3.6 dargestellte Architektur entschieden. Dem `Spawner` wird nun ein Konfigurationsobjekt (`SpawnerConfiguration`) zugewiesen. Über eine *m*:*n*-Beziehung, die durch die Klasse `SpawnerConfigurationXSchedule` bereitgestellt wird, können einem `SpawnerConfiguration`-Objekt mehrere `ScheduleConfiguration`-Objekte zugewiesen werden, welches alle Informationen für ein `Schedule`-Objekt und ein dazugehöriges konkretes `ScheduleStrategy`-Objekt beinhaltet. Die Liste der anzufahrenden Haltestellen wird über die Klasse `ScheduleConfigurationXSimulationPlatform` referenziert. Über das Attribut `index` kann die Reihenfolge der Haltestellen abgebildet werden. Bei diesem Vorgehen ist zu beachten, dass die *m*:*n*-Beziehungen aus der Datenbank nicht vor dem Programmierer versteckt wurden. Stattdessen wurden sie 1:1 in das objektorientierte Paradigma übernommen. Weiterhin gibt es keine Subklassen von `Schedule` und `ScheduleStrategy` mehr. Sämtliche Attribute der Subklassen finden sich in `ScheduleConfiguration` wieder. Entsprechend wird stets nur eine Teilmenge dieser Attribute verwendet. Um die Subklassen dennoch abzubilden, existieren die Felder `schedule_type` und `strategy_type`. Das hier beschriebene Problem wurde bei der Auswahl des ORM nicht vorhergesehen. Um allerdings nicht ein neues ORM in das System integrieren zu müssen, haben wir uns für diese Lösung entschieden. Sie bot einen Zeitvorteil und funktionierte ausreichend gut.

Alle zur Konfiguration gehörenden Klassen werden in JSON-serialisierter Form vom Benutzer über die *REST-API* übertragen [13]. Sie werden daraufhin deserialisiert und in der Datenbank abgelegt. Aus diesen Konfigurationsobjekten können dann ein `Spawner`, `Schedule`-Objekte und `ScheduleStrategy`-Objekte erzeugt werden. Dafür wurde das Entwurfsmuster der *Factory-Method* verwendet. Abbildung 3.7 zeigt, dass die Klassen

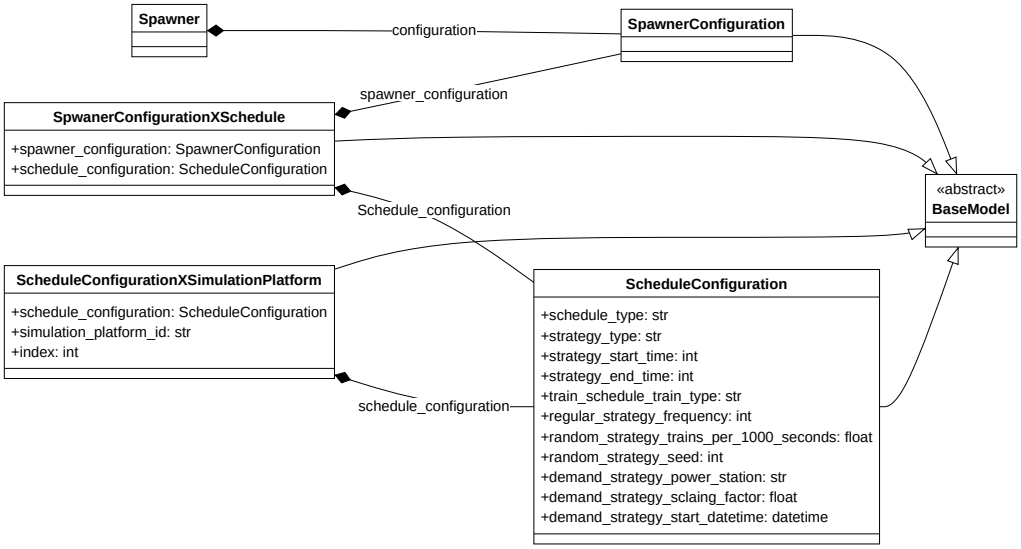

Abbildung 3.6: Klassendiagramm der *Spawner*-Konfiguration. Die Klasse `Spawner` besitzt eine Referenz auf ein `SpawnerConfiguration`-Objekt. Dieses ermöglicht über `SpawnerConfigurationXSchedule`-Objekte die Referenzierung von `ScheduleConfiguration`-Objekten, welche die Informationen für die Abfahrtspläne enthalten. Die Haltestellen werden über `ScheduleConfigurationXSimulationPlatform`-Objekte referenziert. Alle Klassen, bis auf `Spawner` erben von `BaseModel` und sind damit in der Datenbank abgebildet.

`Schedule` und `ScheduleStrategy` bzw. ihre Subklassen ihre eigenen *Factories* sind. Sie besitzen zu diesem Zweck jeweils eine *Klassen*-Methode `from_schedule_configuration`, welche als *Factory-Method* fungiert.

Abbildung 3.8 zeigt, wie aus einer Konfiguration, entsprechende Objekte erzeugt werden. Das `Spawner`-Objekt ruft für jedes referenzierte `ScheduleConfiguration`-Objekt die *Factory-Method* der Klasse `Schedule` auf (1). Diese instanziiert daraufhin ein dem Wert des Attributes `ScheduleType` entsprechendes `Schedule`-Objekt und ein `ScheduleStrategy`-Objekt durch Aufruf der *Factory-Method* der Klasse `ScheduleStrategy` und erneute Übergabe des Konfigurationsobjektes (2). Beide *Factory-Methods* lesen die benötigten Werte für die Attribute der zu erzeugenden Objekte aus dem Konfigurationsobjekt und geben dann das fertige Objekt zurück. Jedes `Schedule`-Objekt kann das erzeugte `ScheduleStrategy`-Objekt im Attribut `strategy` speichern. Das `Spawner`-Objekt kann die erzeugten `Schedule`-Objekte in der Liste `schedules` speichern.

3.2.4 Der Eventbus

Der Eventbus stellt für Softwarekomponenten die Möglichkeit bereit, Ereignisse untereinander auszutauschen. Für jeden Typ von Ereignis stellt er eine *m*:*n*-Beziehung zwischen den Komponenten, welche das Ereignis emittieren und denen, die es empfangen her. Der Eventbus wurde in Zusammenarbeit mit Persitzky entwickelt. Wir haben uns dazu entschieden, das Entwurfsmuster *Mediator* auf den Eventbus anzuwenden, um zu verhindern, dass die voneinander Abhängigen Komponenten direkt miteinander kommunizieren

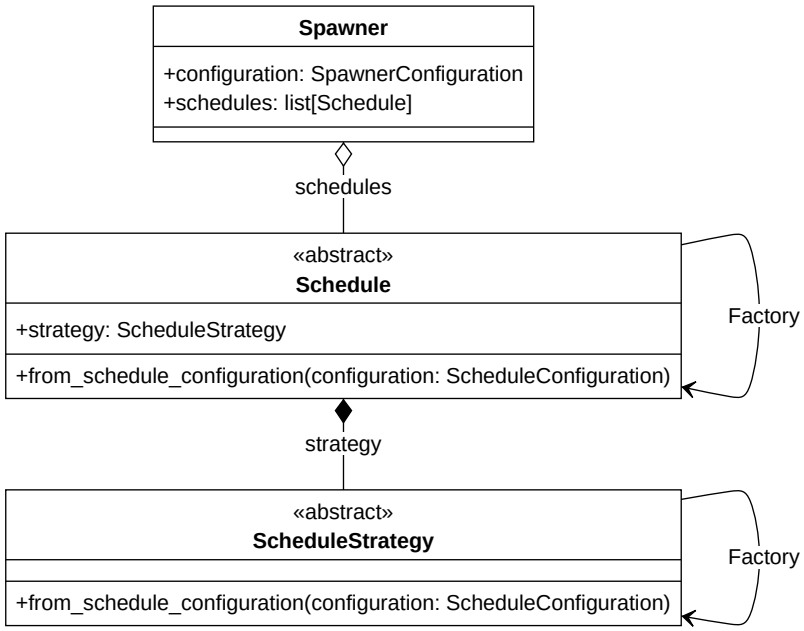

Abbildung 3.7: Klassendiagramm der *Spawner*-Konstruktion. Die Klasse `Spawner` besitzt eine Liste von `Schedule`-Objekten. Die Klasse `Schedule` ist ihre eigene *Factory* und kann aus `ScheduleConfiguration`-Objekten Instanzen ihrer selbst erstellen. Analog dazu besitzt jedes `Schedule`-Objekt eine Referenz auf ein `ScheduleStrategy`-Objekt. Auch die Klasse `ScheduleStrategy` kann sich selbst mithilfe eines `ScheduleConfiguration`-Objektes instanziieren.

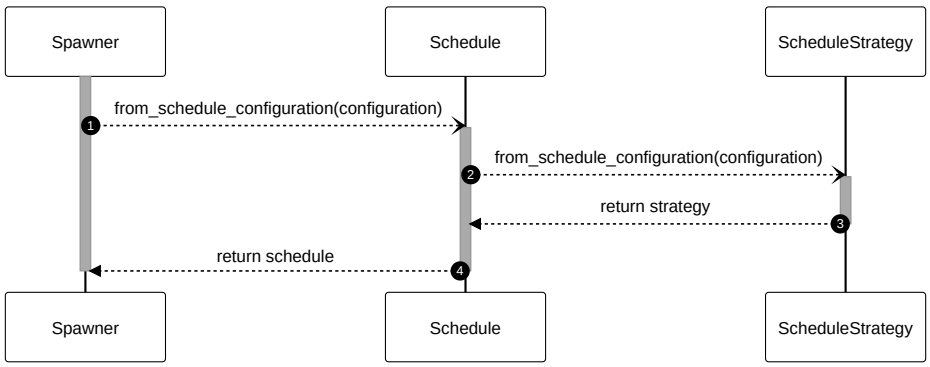

Abbildung 3.8: Sequenzdiagramm der *Spawner*-Konstruktion. Das `Spawner`-Objekt ruft für jedes referenzierte `ScheduleConfiguration`-Objekt die *Factory-Method* der Klasse `Schedule` auf (1). Die Klasse `Schedule` tut daraufhin dasselbe für die Klasse `ScheduleStrategy` (2). Beide Methoden erzeugen dann eine Instanz ihrer jeweiligen Klassen und geben diese zurück (3, 4).

müssen, was die Kopplung senkt. Weiterhin kapselt der *Mediator* das kommunikationsspezifische Verhalten, was es an einer zentralen Stelle verfügbar macht und den einzelnen Komponenten die Verantwortung abnimmt, Kommunikationsdetails zu kennen.

Um eine zu große Komplexität des Eventbus selbst und eine monolithische Struktur zu verhindern, wird als Kommunikationsmechanismus das Entwurfsmuster *Observer* verwendet. Abhängigkeiten zwischen den Komponenten lassen sich so flexibler ändern. Weiterhin ermöglicht dieses Entwurfsmuster, Komponenten auf verschiedenen Abstraktionsniveaus zu verbinden. Wir haben den Nachteil des *Observers* umgangen, dass er stets alle Empfänger benachrichtigt. Dazu wird jeder registrierte Empfänger mit einem Ereignistyp verknüpft. Der Empfänger wird so nur benachrichtigt, wenn ein Ereignis des verknüpften Typs eingegangen ist.

Um die Kopplung weiter zu verringern, werden beim Eventbus nicht die Empfänger selbst registriert, sondern nur *Callbacks* (Funktionen, welche als Argument übergeben werden, um zu einem späteren Zeitpunkt vom Empfänger des Arguments aufgerufen werden zu können), bei welchen es sich jeweils um Methoden der Empfänger handelt. Dies schränkt die Funktionalität nicht ein, da der Empfänger das Ereignis nach wie vor erhält. Der Eventbus muss nun jedoch nur noch Pythons `Callable`-Schnittstelle verwenden und benötigt keine Kenntnis mehr über die kommunizierenden Komponenten. Die Registrierung und die Deregistrierung von Empfängern ist in Abbildung 3.9 und Abbildung 3.10 dargestellt. Für die Registrierung wird das *Callback* und der Ereignistyp (`event_type`) übergeben. Der Eventbus gibt daraufhin ein *Handle* zurück, mit welchem das registrierte *Callback* identifiziert werden kann. Um ein *Callback* zu deregistrieren, lediglich das *Handle* and den Eventbus übergeben werden.

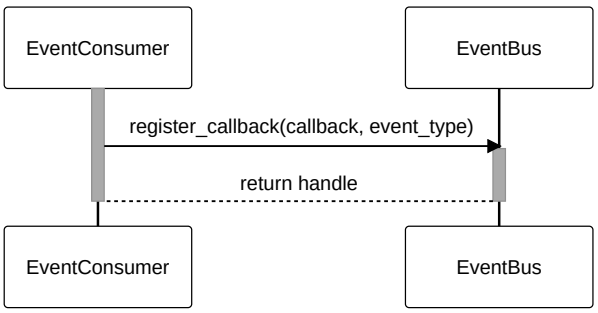

Abbildung 3.9: Sequenzdiagramm der Registrierung eines Empfängers beim Eventbus. Ein `EvenConsumer`-Objekt registriert eine *Callback*-Funktion und einen zugehörigen Ereignistyp beim Eventbus. Der Eventbus gibt ein *Handle* zur Referenzierung des *Callbacks* zurück.

Die Implementierung des Eventbus erfolgte vergleichsweise spät im Projektverlauf. Die Codebasis war bereits entsprechend groß, was uns vor die Herausforderung stellte, wie der Eventbus am besten zu integrieren sei. Auch spielte die Zeitplanung dabei eine Rolle.

Da der *Logger* [20] bereits die Aufgabe übernahm, Ereignisse zu sammeln, lag der Gedanke nahe, ihn entsprechend zu erweitern. Diese Möglichkeit ist im Folgenden unter „Variante 1 – Erweiterung des *Loggers*“ beschrieben.

Da diese Variante gewisse Nachteile mit sich bringt, wird unter „Variante 2 – Eigenstän-

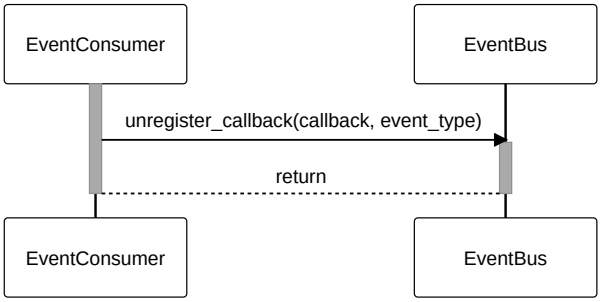

Abbildung 3.10: Sequenzdiagramm der Deregistrierung eines Empfängers beim Eventbus. Unter Verwendung des zuvor erhaltenen *Handles* deregistriert der `EventConsumer` die referenzierte *Callback*-Funktion.

dige Komponente“ eine architektonisch sauberere Lösung vorgestellt. Für diese zweite Lösung haben wir uns aus unten genannten Gründen letztendlich entschieden.

Variante 1 – Erweiterung des *Loggers*

Bei dieser Variante wird das System des *Loggers* erweitert, um die Aufgaben des Eventbus zu übernehmen. Wie Abbildung 3.11 zeigt, erstellt der *Logger* für jedes Ereignis ein `LogEntry`-Objekt, welches in die Datenbank geschrieben wird. Das `LogEntry`-Objekt erbt daher von `BaseModel` und besitzt eine `save`-Methode, welche bei Erzeugung des Objektes stets ausgeführt wird. Darin wird der Eventbus benachrichtigt, welcher dann die registrierten Empfänger über das neue Ereignis informiert. Die Empfänger realisieren dazu die `IEventConsumer`-Schnittstelle, welche eine `on_event`-Methode zur Ereignisbehandlung und ein `event_handle`-Attribut zur Referenzierung der Registrierung beim `EventBus` bereitstellt. Die registrierten *Callbacks* werden im Attribut `callbacks` gehalten. Es handelt sich dabei um eine Liste von Paaren, welche das *Callback* und den Ereignistyp enthalten. Ein *Callback* nimmt dabei ein `LogEntry`-Objekt entgegen und hat keinen Rückgabewert.

Abbildung 3.12 zeigt den Vorgang der Meldung eines Ereignisses an den *Logger* bis hin zur Benachrichtigung der abhängigen Empfänger. Eine Komponenten (`Component`) sendet ein Ereignis durch Aufruf von `log` an den `Logger` (1). Dieser erzeugt durch `create` ein `LogEntry`-Objekt (2) und ruft dessen `save`-Methode auf (3). Diese benachrichtigt den Eventbus durch Senden von `notify` und übergibt dabei das `LogEntry`-Objekt selbst (4). Der Eventbus iteriert über die Liste der registrierten *Callbacks* und ruft für jeden Eintrag das *Callback* auf, sofern der Ereignistyp übereinstimmt (5).

Der Vorteil dieser Variante ist die einfache Integration in das bestehende System. Die Änderungen am bestehenden Quelltext sind gering. Damit kann die Implementierung des Eventbus schnell erfolgen, was einen positiven Einfluss auf die Zeitplanung hat. Von Nachteil ist jedoch, dass das *Logging*-System nun auch Teilaufgaben des Eventbus übernimmt. Dies widerspricht dem Prinzip der *Single Responsibility*[16] und kann zu einer unübersichtlichen Codebasis führen.

[16] Eine Komponente, Klasse oder Funktion soll nur ein Konzept implementieren oder nur eine Aufgabe erledigen.

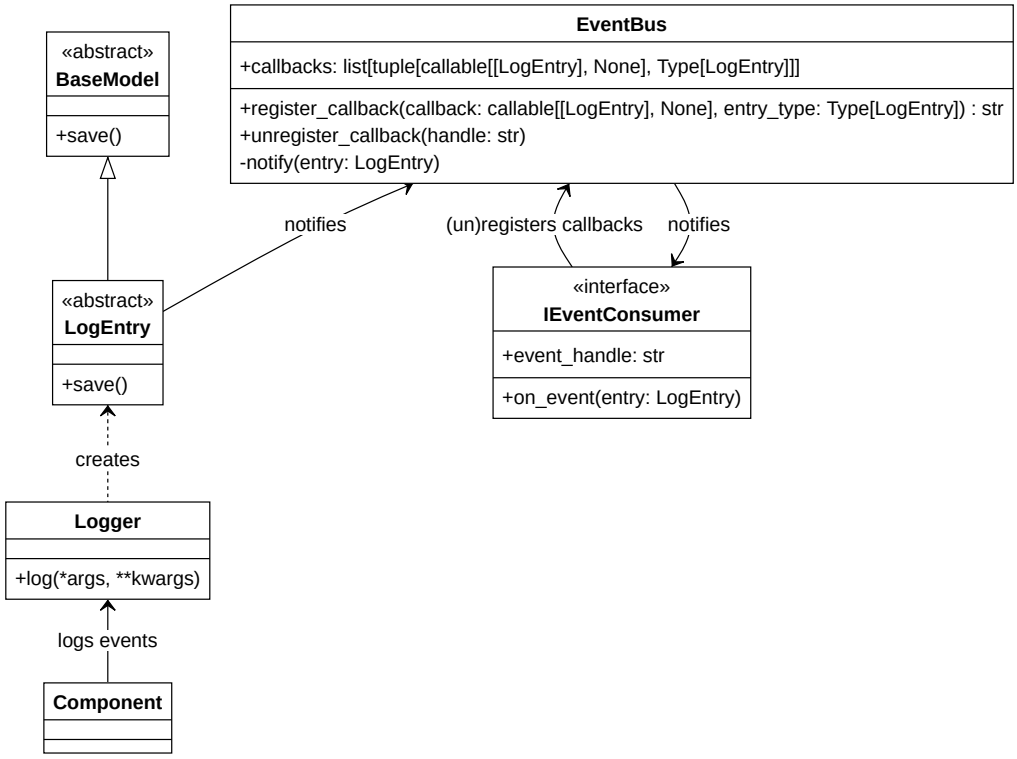

Abbildung 3.11: Klassendiagramm der ersten Variante des Eventbus. Der `Logger` nimmt Ereignisse entgegen. Für jedes Ereignis wird ein `LogEntry`-Objekt erzeugt und die von `BaseModel` geerbte und überschriebene Methode `save` aufgerufen. Diese benachrichtigt den Eventbus, welcher die registrierten Empfänger über das neue Ereignis informiert.

Variante 2 – Eigenständige Komponente

Bei dieser Variante wird der Eventbus als eigenständige Komponente implementiert. Abbildung 3.13 zeigt das Klassendiagramm. Jede Komponente hält nun eine Referenz auf den Eventbus, um sich als Empfänger registrieren zu können. Die Registrierung und Deregistrierung, sowie die Verwaltung der registrieren *Callbacks* erfolgt wie bei der „Erweiterung des *Loggers*“. Der Unterschied besteht darin, dass die Ereignisse nun nicht mehr durch `LogEntry`-Objekte, sondern durch `Event`-Objekte repräsentiert werden. Komponenten benachrichtigen hier selbst den `EventBus`. Der *Logger* implementiert die `IEventConsumer`-Schnittstelle und registriert sich ebenfalls beim Eventbus, um seine Aufgabe erfüllen zu können.

Abbildung 3.14 zeigt den Ablauf der Meldung eines Ereignisses an den Eventbus. Eine Komponente (`Component`) sendet die Parameter eines Ereignisses, durch Aufruf von `log`, an den `EventBus` (1). Dieser instanziiert ein `Event`-Objekt mit den übergebenen Parametern (2). Anschließend iteriert er über die Liste der registrierten *Callbacks* und ruft diese auf, sofern der Ereignistyp übereinstimmt (3).

Der Vorteil dieser Variante ist die saubere Trennung der Aufgaben des *Loggers* und des Eventbus und die klare Zuordnung der Verantwortlichkeiten. Dafür ist die Implementie-

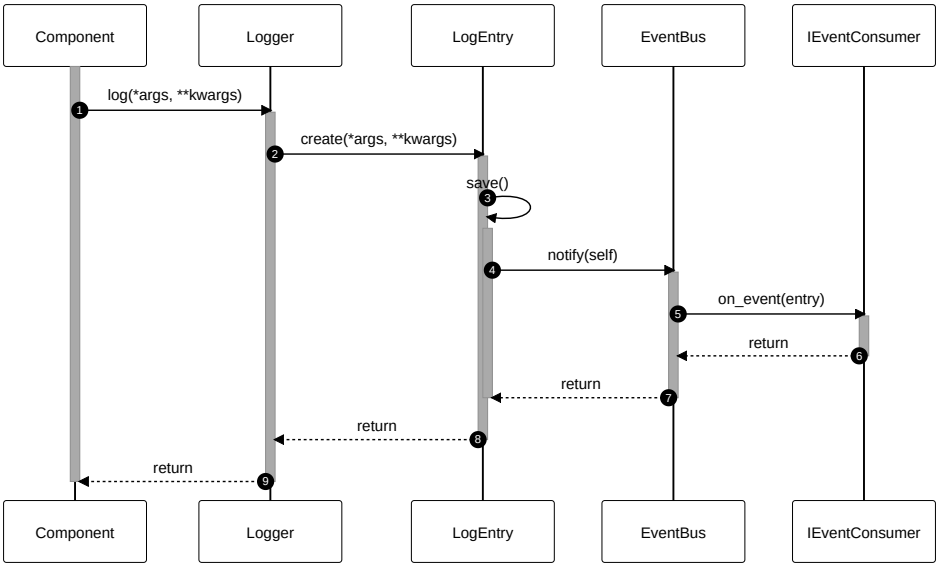

Abbildung 3.12: Sequenzdiagramm der ersten Variante des Eventbus. Eine Komponente sendet ein Ereignis an den *Logger* (1). Dieser erzeugt ein `LogEntry`-Objekt (2)auf welchem `save` aufgerufen wird (3). Dabei wird der Eventbus benachrichtigt (4), welcher die registrierten Empfänger über das neue Ereignis informiert (5).

rung aufwendiger, da mehr Änderungen am bestehenden Quelltext notwendig sind. Das bezieht sich vor allem auf den Umbau des *Loggers*, welcher mit `Event`-Objekten statt mit direkt übergebenen Argumenten arbeiten muss. Trotz des Mehraufwandes haben wir uns letztendlich zugunsten der Codequalität für diese Variante entschieden.

Eine Verwendung findet der Eventbus im aktuellen System nicht. Aus Zeitgründen wurder der Eventbus zwar implementiert, jedoch nicht mehr verwendet. Mögliche Verwendungen sind im Ausblick dieser Arbeit beschrieben.

3.3 Implementierungsdetails

In diesem Abschnitt wird eine Auswahl Implementierungsdetails auf Quelltext-Ebene vorgestellt. Da das grundlegende Verhalten der Komponenten bereits beschrieben wurde, konzentriert sich dieser Teil auf Details, die sich in Sequenzdiagrammen nicht gut darstellen lassen aber für das Verständnis der Komponenten von Bedeutung sind.

3.3.1 Template-Method in Schedule

Quelltext 3.1 zeigt einen Ausschnitt aus der abstrakten Klasse `Schedule`. Gezeigt wird die bereits beschriebene *Template-Method* `maybe_spawn`, welche das Grundgerüst des Algorithmus für die Zugerzeugung bereitstellt.

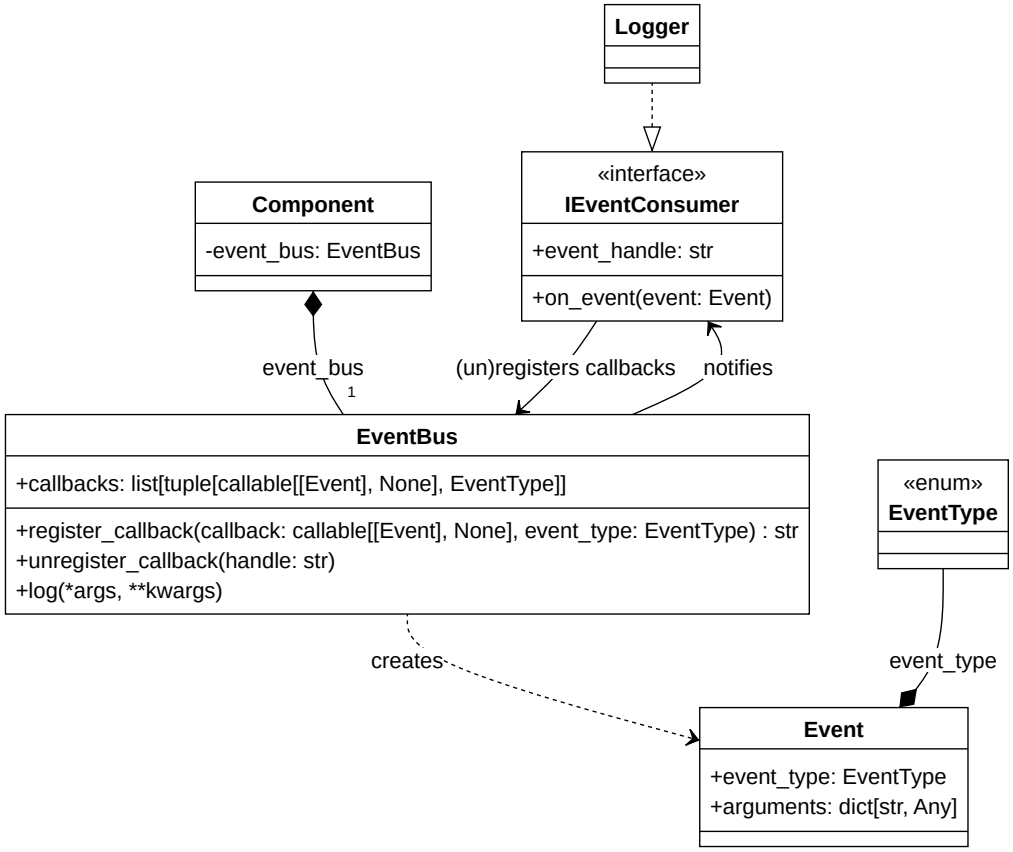

Abbildung 3.13: Klassendiagramm der zweiten Variante des Eventbus. Die Komponenten benachrichtigen den Eventbus direkt über neue Ereignisse. Der Eventbus leitet die Ereignisse dann an die registrierten Empfänger weiter. Die ereignis-spezifischen Informationen sind in `Event`-Objekten gekapselt.

Zuerst wird geprüft, ob der Abfahrtsplan durch einen injizierten Fehler deaktiviert wurde (`blocked`). Im Anschluss wird `should_spawn` auf der konkreten Strategie (`strategy`) aufgerufen, um zu prüfen, ob zum aktuellen Zeitpunkt (`seconds`) ein Zug erzeugt werden soll. Ein bisher nicht diskutiertes Problem, welches es zu lösen galt, war die Tatsache, dass es möglich war, dass ein Zug in einem Abschnitt erzeugt wird, der bereits durch einen anderen Zug belegt ist. Dieses Problem wird ebenfalls in `maybe_spawn` gelöst. Zeigen beide zuvor geprüften Bedingungen an, dass ein Zug erzeugt werden soll, wird dessen Erzeugungszeitpunkt zunächst einer Warteschlange (`seconds_to_be_spawned`) hinzugefügt. Wenn die Warteschlange im Anschluss nicht leer ist, wird versucht, durch Aufruf von `spawn`, einen Zug zu erzeugen. `spawn` wird in den Subklassen von `Schedule` implementiert und gibt einen Wahrheitswert zurück, der angibt, ob die Erzeugung erfolgreich war. In `spawn` findet auch die Prüfung auf belegte Abschnitte statt. War die Erzeugung erfolgreich, wird der entsprechende Zeitpunkt aus der Warteschlange entfernt.

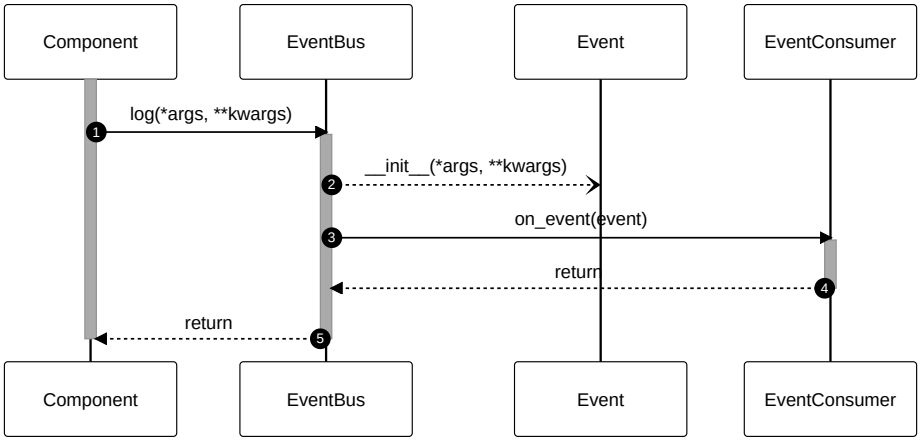

Abbildung 3.14: Sequenzdiagramm der zweiten Variante des Eventbus. Eine Komponente sendet die Parameter eines Ereignisses an den Eventbus (1). Der Eventbus erzeugt daraufhin ein `Event`-Objekt (2) und benachrichtigt die registrierten Empfänger (3).

Dieses Vorgehen hat zur Folge, dass Züge, deren Erzeugung im Moment nicht möglich ist, zwischengespeichert werden, um zum nächstmöglichen Zeitpunkt erzeugt werden zu können.

Quelltext 3.1: Ausschnitt aus der abstrakten Klasse `Schedule`, welcher die *Template-Method* `maybe_spawn` und weitere relevante Attribute zeigt. `maybe_spawn` prüft, ob zum Zeitpunkt `seconds` ein Zug erzeugt werden soll und erzeugt ihn dann ggf.

```
class Schedule(ABC):

    strategy: ScheduleStrategy
    seconds_to_be_spawned: list[int]
    blocked: bool

    def maybe_spawn(self, seconds: int, spawner: Spawner):
        if not self.blocked and self.strategy.should_spawn(seconds):
            self.seconds_to_be_spawned.append(seconds)

        if len(self.seconds_to_be_spawned) > 0:
            if self.spawn(spawner, self.seconds_to_be_spawned[-1]):
                self.seconds_to_be_spawned.pop()

    @abstractmethod
    def spawn(self, spawner: Spawner, seconds: int) -> bool:
        raise NotImplementedError()
```

3.3.2 Algorithmen zur Zugerzeugung

Im Folgenden wird die Implementierung der Algorithmen für die verschiedenen Arten der Zugerzeugung vorgestellt. Der entsprechende Code dafür befindet sich in der Klasse `ScheduleStrategy` und ihren Subklassen, jeweils in der Methode `should_spawn`. Diese

Methode nimmt den aktuellen Zeitpunkt `seconds` entgegen und gibt einen Wahrheitswert zurück, der angibt, ob zum aktuellen Zeitpunkt ein Zug erzeugt werden muss.

Quelltext 3.2 zeigt die abstrakte Klasse `ScheduleStrategy`. In ihrer `should_spawn`-Methode werden Bedingungen geprüft, die für jede Art von Abfahrtsplan erfüllt sein müssen. Speziell ist das die Prüfung, ob der aktuelle Zeitpunkt innerhalb des Intervalls liegt, welches von den Attributen `start_time` und `end_time` definiert wird. Diese Attribute sind optional, sodass auch ein offenes Intervall definiert werden kann.

Quelltext 3.2: Ausschnitt aus der abstrakten Klasse `ScheduleStrategy`. Gezeigt wird die Methode `should_spawn`, sowie die relevanten Attribute `start_time` und `end_time`.

```
class ScheduleStrategy(ABC):

    start_time: int
    end_time: int

    def should_spawn(self, seconds: int) -> bool:
        is_after_start_second = seconds >= (self.start_time if self.start_time else 0)
        is_before_end_second = seconds <= (self.end_time if self.end_time else seconds)
        return is_after_start_second and is_before_end_second
```

In den Subklassen wird zuerst stets die `should_spawn`-Methode der Superklasse aufgerufen, um die allgemeingültigen Bedingungen zu prüfen. Anschließend werden die spezifischen Bedingungen der jeweiligen Strategie geprüft.

Quelltext 3.3 zeigt die Implementierung der Klasse `RegularScheduleStrategy`. Diese Strategie erzeugt Züge in regelmäßigen Abständen. Dazu wird das Attribut `frequency` verwendet, welches die Anzahl der Sekunden zwischen zwei Zügen angibt. Die `should_spawn`-Methode prüft, ob die Anzahl der Sekunden, die seit dem Start des Abfahrtsplans (`start_time`) vergangen sind, durch die `frequency` teilbar ist.

Quelltext 3.3: Ausschnitt aus der Klasse `RegularScheduleStrategy` mit der Implementierung der Methode `should_spawn`.

```
class RegularScheduleStrategy(ScheduleStrategy):

    frequency: int

    def should_spawn(self, seconds: int) -> bool:
        return (
            super().should_spawn(seconds)
            and (seconds - self.start_time) % self.frequency == 0
        )
```

Quelltext 3.4 zeigt einen Ausschnitt der Implementierung der Klasse `RandomScheduleStrategy`. Diese Strategie erzeugt Züge mit einer bestimmten Wahrscheinlichkeit. Dazu wird das Attribut `trains_per_1000_seconds` verwendet, welches die mittlere Anzahl der Züge pro 1000 Sekunden angibt. Die `should_spawn`-Methode prüft, ob eine Zufallszahl zwischen 0 und 1000 kleiner als `trains_per_1000_seconds` ist. Außerdem

wird der Initialisierungsmethode (`__init__`) ein Startwert (`seed`) für den Zufallszahlengenerator (`random_number_generator`) übergeben, um die Reproduzierbarkeit auch bei pseudozufälligen Ergebnissen zu gewährleisten.

Quelltext 3.4: Ausschnitt aus der Klasse `RandomScheduleStrategy` mit der Implementierung der Methode `should_spawn`.

```
class RandomScheduleStrategy(ScheduleStrategy):

    trains_per_1000_seconds: float
    random_number_generator: Random

    def __init__(
        self,
        start_time: int,
        end_time: int,
        trains_per_1000_seconds: float,
        seed: int = None
    ):
        super().__init__(start_time, end_time)
        self.trains_per_1000_seconds = trains_per_1000_seconds
        self.random_number_generator = Random(seed)

    def should_spawn(self, seconds: int) -> bool:
        return (
            super().should_spawn(seconds)
            and self.random_number_generator.random() * 1000
            < self.trains_per_1000_seconds
        )
```

Quelltext 3.5 zeigt einen Ausschnitt der Implementierung der Klasse `DemandScheduleStrategy`. Die Zeitpunkte, zu denen Züge erzeugt werden sollen, werden nach dem im Hauptteil beschriebenen Verfahren, vorberechnet und in der Liste `spawn_seconds` gespeichert. In `should_spawn` wird dann geprüft, ob der aktuelle Zeitpunkt in der Liste enthalten ist.

Quelltext 3.5: Ausschnitt aus der Klasse `DemandScheduleStrategy` mit der Implementierung der Methode `should_spawn`.

```
class DemandScheduleStrategy(ScheduleStrategy):

    spawn_seconds: list[int]

    def should_spawn(self, seconds: int) -> bool:
        return super().should_spawn(seconds) and seconds in self.spawn_seconds
```

3.4 Ergebnisse

In diesem Abschnitt werden die Ergebnisse diskutiert, welche sich aus der Berechnung des Kohlebedarfs und der Bestimmung der Zugerzeugungszeitpunkte ergeben haben. Abbildung 3.15 a zeigt den berechneten Kohlebedarf für das Kraftwerk *Jänschwalde* in

Tonnen zwischen dem 25.01.2023 und dem 30.01.2023. Jeder Datenpunkt steht dabei für einen Zeitraum von 15 Minuten. Grau hinterlegt sind die Zeiträume A und B, welche in den Teilabbildungen 3.15 b und 3.15 c genauer betrachtet werden. Abbildung 3.15 b zeigt den Zeitraum A vom 26.01. 12:00 bis 27.01. 12:00. Der Kohlebedarf ist in dieser Zeit praktisch konstant bei rund 750 t pro 15 Minuten. Die vertikalen roten Linien zeigen an, zu welchen Zeitpunkten ein Kohlezug erzeugt werden würde. Es ist zu beobachten, dass die Abstände dieser Linien zueinander relativ gleichmäßig sind. Abbildung 3.15 c zeigt den Zeitraum B vom 28.01. 12:00 bis 29.01. 12:00. In diesem Zeitraum unterliegt der Kohlebedarf einer deutlichen Schwankung. Der Zeitraum B ist zur weiteren Betrachtung in die Teilzeiträume B1 und B2 aufgeteilt, deren Grenze bei 00:00 Uhr (markiert durch die schwarze Strich-Punkt-Linie) liegt. Innerhalb von B1 liegt der Kohlebedarf bei etwa 700 t pro 15 Minuten. In B2 sinkt er auf ca. 400 bis 500 t pro 15 Minuten ab. Es ist zu erkennen, dass die Anzahl der in einer Zeitspanne erzeugten Kohlezüge direkt mit dem Kohlebedarf korreliert. Je höher also der Wert des Kohlebedarfs ist, desto mehr Züge werden erzeugt.

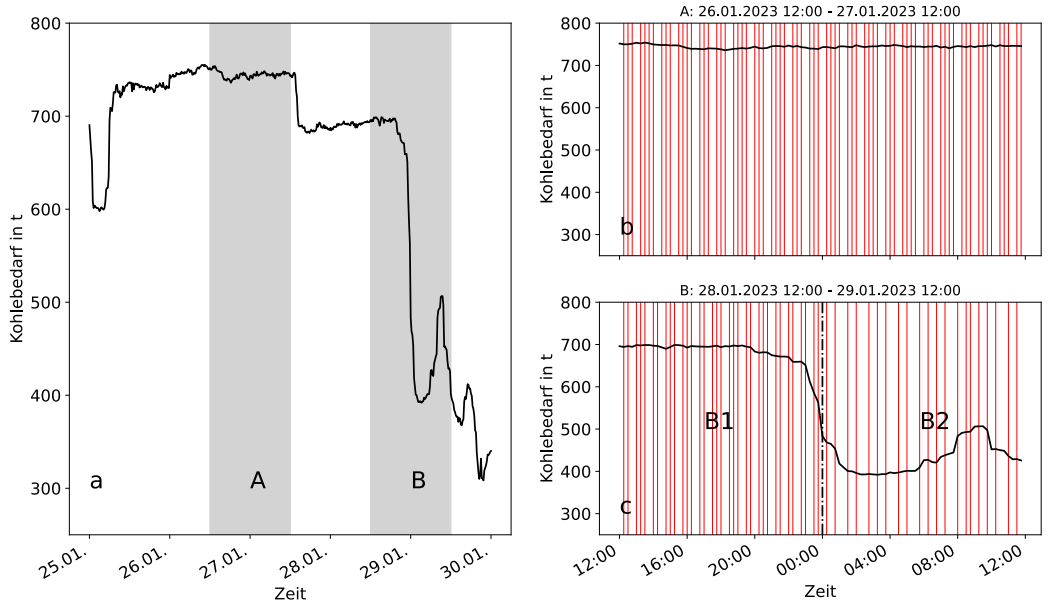

Abbildung 3.15: Kohlebedarf in Tonnen für das Kraftwerk *Jänschwalde* zwischen dem 25.01.2023 und dem 30.01.2023 (a). Die Teilabbildungen b und c zeigen die Zeiträume A und B genauer. Die vertikalen, roten Punkt-Linien stellen die Abfahrtzeitpunkte von Kohlezügen dar. Die Strich-Punkt-Linie in Teilabbildung c teilt den Zeitraum in die Teilzeiträume B1 und B2 auf.

Abbildung 3.16 stellt die Daten aus Tabelle 3.1 grafisch dar. In Abbildung 3.16 a sind die mittleren Abfahrtsperioden von Kohlezügen in Richtung Kraftwerk *Jänschwalde* in den Zeiträumen A, B1 und B2 dargestellt. Es fällt auf, dass die Zeiträume A und B1 eine ähnliche Periodenlänge zwischen um 20 Minuten besitzen und die Periodenlänge in B2 mit etwa 3 Minuten deutlich höher ist.

In Abbildung 3.16 b sind die mittleren Abfahrtsperioden in Zeitraum A für die Kraftwerke *Jänschwalde*, *Boxberg* und *Schwarze Pumpe* zu erkennen. Das Kraftwerk *Jänschwalde* besitzt

hierbei den niedrigsten Wert mit etwa 19 Minuten. Das Kraftwerk *Boxberg* besitzt eine mittlere Abfahrtsperiode von ca. 28 Minuten und den höchsten Wert weist das Kraftwerk *Schwarze Pumpe* mit rund 43 Minuten auf.

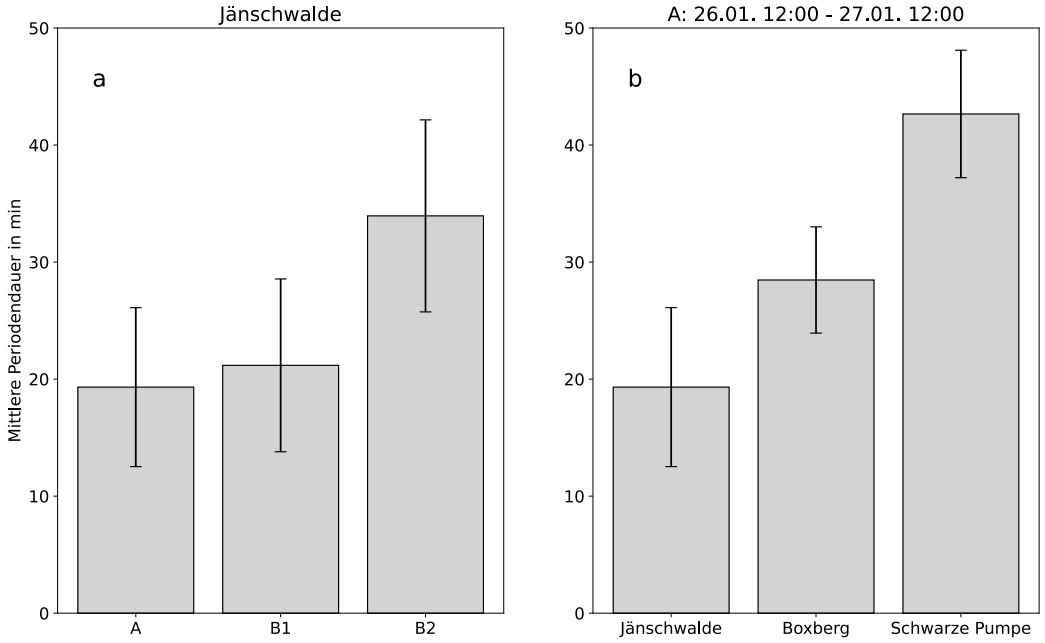

Abbildung 3.16: Mittlere Abfahrtsperioden in min für die drei Zeiträume A, B1 und B2 zum Kraftwerk *Jänschwalde* (a) und für die drei Kraftwerke *Jänschwalde*, *Boxberg* und *Schwarze Pumpe* im Zeitraum A (b). Zeitraum A: 26.01. 12:00 - 27.01. 12:00, Zeitraum B1: 28.01. 12:00 - 29.01. 00:00, Zeitraum B2: 29.01. 00:00 - 29.01. 12:00

Tabelle 3.1: Mittlere Abfahrtsperioden in min für die drei Zeiträume A, B1 und B2 zum Kraftwerk *Jänschwalde* (oberer Tabellenteil) und für die drei Kraftwerke *Jänschwalde*, *Boxberg* und *Schwarze Pumpe* im Zeitraum A (unterer Tabellenteil). Zeitraum A: 26.01. 12:00 - 27.01. 12:00, Zeitraum B1: 28.01. 12:00 - 29.01. 00:00, Zeitraum B2: 29.01. 00:00 - 29.01. 12:00

Zeitraum	**Mittlere Abfahrtsperiode in min**
A	$19,32 \pm 6,79$
B1	$21,18 \pm 7,38$
B2	$33,95 \pm 8,20$
Kraftwerk	
Jänschwalde	$19,32 \pm 6,79$
Boxberg	$28,47 \pm 4,54$
Schwarze Pumpe	$42,66 \pm 5,45$

4 Schlussbetrachtung

Zuletzt werden die Ergebnisse der Arbeit diskutiert. Weiterhin wird ein Ausblick auf mögliche Erweiterungen gegeben.

4.1 Ergebnisdiskussion

Die Ergebnisse in Abbildung 3.15 zeigen, dass die Längen der Abfahrtsperioden der Kohlezüge direkt mit dem berechneten Kohlebedarf korrelieren. Das bedeutet, dass umso mehr Kohlezüge abfahren, je mehr Kohle benötigt wird. Dieses Ergebnis ist sinnvoll und deckt sich mit der Realität und dem was man erwarten würde. Laut Sascha Lesche (siehe Anhang) können an einem Tag etwa 80 Züge in ein großes Kraftwerk einfahren. Diese Zahl kann jedoch je nach tatsächlichem Bedarf auch höher oder niedriger sein. Umgerechnet ergibt sich aus diesem Wert eine mittlere Abfahrtsperiodenlänge von 18 Minuten. Das Ergebnis für das größte der drei Kraftwerke, *Jänschwalde*, liegt mit einer mittleren Abfahrtsperiodenlänge von etwa 19 Minuten sehr nah an diesem Wert.

Auch die Verhältnisse der Abfahrtsperiodenlänge der drei Kraftwerke *Jänschwalde*, *Boxberg* und *Schwarze Pumpe* decken sich mit der Realität. Das Kraftwerk *Jänschwalde* hat mit einer Gesamtleistung von rund 3,3 GW die höchste Gesamtleistung der drei Kraftwerke. Entsprechend muss es den höchsten Kohlebedarf besitzen, was die niedrigste Abfahrtsperiodenlänge von ca. 19 Minuten plausibel macht. Auch die Gesamtleistungen des zweit-leistungsfähigsten Kraftwerkes *Boxberg* mit 2,7 GW und des Kraftwerkes *Schwarze Pumpe* mit 1,7 GW decken sich entsprechend mit den Ergebnissen der Simulation. [4]

Zusammenfassen lässt sich sagen, dass die Ergebnisse der Simulation plausibel sind und sich mit der Realität decken.

4.2 Ausblick

Die von uns entwickelte Software stellt eine Erweiterung des Verkehrssimulators *SUMO* dar, welche es ermöglicht, Zugverkehr auf einem realistischeren Niveau durchzuführen, als dies vorher möglich war. Wir stellen eine Plattform bereit, welche Züge mithilfe einer Stellwerkslogik durch ein Schienennetz bewegen kann. Für die Abfahrt der Züge lassen sich sogenannte Abfahrtspläne konfigurieren. Damit können bisher regelmäßige und zufällige Abfahrten simuliert werden, als auch solche, die sich aus dem Kohlebedarf eines Kraftwerkes ergeben. Insbesondere die letzte Möglichkeit ist von besonderer Bedeutung

für die Fragestellungen, die es im Rahmen des Projektes *FlexiDug* zu beantworten gilt. Natürlich gibt es nach wie vor Aspekte des Bahnverkehrs, welche nicht durch unser System abgebildet werden können.

Eine Möglichkeit zur Erweiterung der Software, um den Realismus der Simulation zu erhöhen, sind *Reaktive Abfahrtspläne*. Diese würden es ermöglichen, dass Züge auf Ereignisse in der Simulation reagieren. Zentrales Element der Implementierung dieser Abfahrtspläne ist der Eventbus, welcher in der Software bereits implementiert ist. Die Routenberechnung könnte beispielsweise ein Ereignis auslösen, wenn ein Zug seine Endhaltestelle erreicht hat. Ein reaktiver Abfahrtsplan könnte dann auf dieses Ereignis reagieren und einen neuen Zug abfahren lassen. Damit ließe sich die Rückfahrt eines Zuges simulieren. Auch können damit Abhängigkeiten zwischen Abfahrten modelliert werden. So könnte ein Zug erst abfahren, wenn ein anderer Zug eine bestimmte Haltestelle erreicht hat, beispielsweise weil das Zugpersonal selbst erst mit der Bahn anreisen muss. Eine weitere zukünftige Verwendungsmöglichkeit des Eventbus wird von Persitzky in seiner Arbeit [17] diskutiert.

Es liegt nun in der Hand der Projektpartner von *FlexiDug*, ihre Fragestellungen mithilfe unseres Systems zu untersuchen. Sie verfügen damit über ein Werkzeug, welches es ihnen ermöglicht, die Auswirkungen verschiedener Szenarien für den Bahnverkehr im Schienennetz der LEAG zu untersuchen. Aufgrund der Flexibilität und Erweiterbarkeit der Software, kann diese in Zukunft auch für andere Schienennetze und ähnliche Fragestellungen eingesetzt werden.

Literaturverzeichnis

[1] *Braunkohle*. Ministerium für Wirtschaft, Arbeit und Energie (MWE). URL: http://brandenburg.de/de/bb1.c.478774.de (besucht am 3. Juli 2023).

[2] BTU. *FlexiDug*. 22. Okt. 2022. URL: https://www.b-tu.de/fg-betriebssysteme/forschung/drittmittelgefoerderte-projekte/flexidug (besucht am 18. Juni 2023).

[3] *bundesAPI/smard-api*. GitHub. URL: https://github.com/bundesAPI/smard-api (besucht am 22. Juni 2023).

[4] *Bundesnetzagentur - Kraftwerksliste*. URL: https://www.bundesnetzagentur.de/DE/Sachgebiete/ElektrizitaetundGas/Unternehmen_Institutionen/Versorgungssicherheit/Erzeugungskapazitaeten/Kraftwerksliste/kraftwerksliste-node.html (besucht am 22. Juni 2023).

[5] Bundesverband Braunkohle. *Braunkohle ist der einzige heimische Energieträger, der in großen Mengen subventionsfrei zu wettbewerbsfähigen Konditionen bereitgestellt werden kann.* URL: https://braunkohle.de/wp-content/uploads/2019/08/Bodenschatz-Braunkohle-%E2%80%93-Langfassung.pdf (besucht am 4. Juli 2023).

[6] *Eclipse SUMO - Simulation of Urban MObility*. Eclipse SUMO - Simulation of Urban MObility. URL: https://www.eclipse.org/sumo/ (besucht am 21. Juni 2023).

[7] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. ISBN: 978-0-201-63361-0.

[8] Thomas Andreas Gröger und Wulf Schwanhäußer. „Simulation der Fahrplanerstellung auf der Basis eines hierarchischen Trassenmanagements und Nachweis der Stabilität der Betriebsabwicklung“. Medium: online, print. Dissertation. Aachen: Publikationsserver der RWTH Aachen University, 2002. 181 Seiten.

[9] Hasso Plattner Institut. *FlexiDug: Neue Perspektiven für das Lausitzer Werkbahnnetz*. 13. Juli 2022. URL: https://hpi.de/news/jahrgaenge/2022/flexidug-neue-perspektiven-fuer-das-lausitzer-werkbahnnetz.html (besucht am 18. Juni 2023).

[10] Daniela Hertzer. *Eisenbahner im Lausitzer Revier | LEAG Blog*. 20. Nov. 2018. URL: https://www.leag.de/de/seitenblickblog/artikel/eisenbahnerin-im-lausitzer-revier/ (besucht am 18. Juni 2023).

[11] *interlocking*. original-date: 2023-01-24T16:25:25Z. 24. Jan. 2023. URL: https://github.com/simulate-digital-rail/interlocking (besucht am 21. Juni 2023).

[12] Christopher Ireland, David Bowers, Michael Newton und Kevin Waugh. „Understanding Object-Relational Mapping: A Framework Based Approach“. In: 2 (2009), Seiten 202–216. URL: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=35eefea817172677f03b12baa2916060957cfb23#page=38.

[13] Antony Kamp. „Architektur und Entwicklung einer REST-API zur Simulation von Zugverkehr am Beispiel des LEAG-Schienennetzes in der Lausitz“. Bachelorarbeit. Potsdam: Hasso-Plattner-Institut, Universität Potsdam, 2023.

[14] Charles Leifer. *coleifer/peewee: a small, expressive orm – supports postgresql, mysql and sqlite*. 2023. URL: https://github.com/coleifer/peewee (besucht am 18. Juni 2023).

[15] Max Lietze. „Evaluierung und Anpassung von Routingalgorithmen für das Finden von nebenläufigen Routen zwischen Betriebsstellen am Beispiel einer Simulation des LEAG-Schienennetzes in der Lausitz“. Bachelorarbeit. Potsdam: Hasso-Plattner-Institut, Universität Potsdam, 2023.

[16] Luke Ortlam. „Implementierung und Evaluierung einer API-Abstraktion um eine simulatorunabhängige Zugsimulation zu ermöglichen“. Bachelorarbeit. Potsdam: Hasso-Plattner-Institut, Universität Potsdam, 2023.

[17] Anton Persitzky. „Fehlerinjektion als Methode zur Bewertung der Resilienz von simulierten Eisenbahnverkehrssystemen“. Bachelorarbeit. Potsdam: Hasso-Plattner-Institut, Universität Potsdam, 2023.

[18] *Quickstart — peewee 3.16.2 documentation*. 2023. URL: https://docs.peewee-orm.com/en/latest/peewee/quickstart.html#model-definition (besucht am 18. Juni 2023).

[19] rbb. *HPI prüft Kohlestrecken in der Lausitz für Personenzugverkehr*. 15. Juli 2022. URL: https://www.rbb24.de/studiocottbus/panorama/2022/07/lausitz-schienen-kohle-streckennetz-zugverkehr-leag-hpi-konzept.html (besucht am 18. Juni 2023).

[20] Lucas Reisener. „Entwurf einer Software zur Protokollierung und Auswertung von Daten aus Simulationen von Zugverkehr am Beispiel des LEAG-Schienennetzes“. Bachelorarbeit. Potsdam: Hasso-Plattner-Institut, Universität Potsdam, 2023.

[21] Olga Skobeleva und Oleksandr Shvets. *Factory Method*. 2023. URL: https://refactoring.guru/design-patterns/factory-method (besucht am 18. Juni 2023).

[22] Olga Skobeleva und Oleksandr Shvets. *Mediator*. 2023. URL: https://refactoring.guru/design-patterns/mediator (besucht am 18. Juni 2023).

[23] Olga Skobeleva und Oleksandr Shvets. *Observer*. 2023. URL: https://refactoring.guru/design-patterns/observer (besucht am 18. Juni 2023).

[24] Olga Skobeleva und Oleksandr Shvets. *Strategy*. 2023. URL: https://refactoring.guru/design-patterns/strategy (besucht am 18. Juni 2023).

[25] Olga Skobeleva und Oleksandr Shvets. *Template Method*. 2023. URL: https://refactoring.guru/design-patterns/template-method (besucht am 18. Juni 2023).

[26] Olga Skobeleva und Oleksandr Shvets. *Visitor*. 2023. URL: https://refactoring.guru/design-patterns/visitor (besucht am 18. Juni 2023).

[27] *SMARD | Regelzone*. URL: https://www.smard.de/page/home/wiki-article/446/205156 (besucht am 22. Juni 2023).

[28] *SQLAlchemy*. 2023. URL: https://www.sqlalchemy.org (besucht am 19. Juni 2023).

[29] *Tagebau Lausitz | Geschäftsfeld Bergbau | LEAG*. 2023. URL: https://www.leag.de/de/geschaeftsfelder/bergbau/ (besucht am 18. Juni 2023).

[30] *What is Object/Relational Mapping? - Hibernate ORM*. Hibernate. 2023. URL: https://hibernate.org/orm/what-is-an-orm/ (besucht am 18. Juni 2023).

A Anhang

Interview mit Sascha Lesche von der LEAG

Christian Raue: Lassen Sie uns zunächst zum Thema des Kohletransportes kommen. Wie viele Tonnen Kohle passen in einen Zug?

Sascha Lesche: Ein Zug besteht immer aus einer Lok und 16 Wagen. Jeder Wagen hat eine Aufnahmekapazität von 84 Kubikmeter, was 60 Tonnen Kohle entspricht. Damit kommen Sie auf eine Gesamtmenge von 60 Tonnen.

Christian Raue: Welchen Energiegehalt hat die transportierte Kohle im Schnitt?

Sascha Lesche: Der Energiegehalt der Kohle schwankt. Da sie nicht gleichmäßig gewachsen ist, weist sie teilweise sehr unterschiedliche chemische Zusammensetzungen auf, welche den Energiegehalt beeinflusst. Für eine Faustformel empfehle ich, sich auf allgemeine Quellen zu berufen.

Max Lietze: Als Nächstes möchten wir gern mit Ihnen darüber reden, wie wir die Züge in unserer Simulation möglichst realistisch repräsentieren können. Eine Frage ist dabei, wie lang so ein Zug ist.

Sascha Lesche: Die Länge eines Wagens beträgt 12,5 Meter. Zusammen mit der Lok kommen Sie dann auf eine Länge von ca. 220 Metern.

Max Lietze: Wie schnell kann ein Kohlezug fahren?

Sascha Lesche: Die Loks schaffen eine Geschwindigkeit von 60 km/h, dürfen aber gemäß Bau- und Betriebsanweisung maximal 50 km/h fahren.

Max Lietze: Wie schnell könne die Züge beschleunigen und abbremsen?

Sascha Lesche: Es gibt zunächst sogenannte Zuglastdiagramme, bei welchen die Zugkraft in Zusammenhang mit der Masse gebracht wird. Wahrscheinlich wird es aber schwierig sein, die Werte daraus abzuleiten. Ich kann Ihnen jedoch theoretische Annahmen geben. Für die Beschleunigung wären das 0,15 Meter pro Quadratsekunde und für das Abbremsen 0,4 Meter pro Quadratsekunde.

Max Lietze: Wie lange dauert das Be- und Entladen?

Sascha Lesche: Das Be- und Entladen dauert zwischen 20 und 30 Minuten. Wenn Sie ihr System stabil laufen lassen wollen, sollten Sie vielleicht besser von 30 Minuten ausgehen.

Christian Raue: Wir möchten unsere Simulationsergebnisse gern mithilfe von Echtweltdaten verifizieren. Können Sie uns sagen, in welchem Takt die Züge im Mittel in den Kraftwerken ankommen?

Sascha Lesche: Vor vielen Jahren, fuhren die Kraftwerke noch eine reine Grundlast. Da hätte man ein Tagesmittel angeben können. Im Moment ist das nicht möglich. Es

gibt je nach Bedarf große Schwankungen. Am Tag können 80 Züge in ein großes Kraftwerk einfahren. Manchmal sind es mehr, bei entsprechend geringeren Leistungen auch weniger.

Christian Raue: Wie viele Züge befinden sich im Schnitt gleichzeitig auf dem Schienennetz?

Sascha Lesche: Das ist leider genauso schwer zu beantworten. Es können zehn bis zwanzig Züge sein. Bei 25 Zügen ist dann aber langsam ein Limit erreicht, von Reststoff- und Transportfahrten abgesehen.

Eidesstattliche Erklärung

Hiermit versichere ich, dass meine Bachelorarbeit „Entwurf und Implementierung von regelmäßigen, randomisierten und bedarfsorientierten Abfahrtsplänen für Züge in einer Verkehrssimulation“ („Design and Implementation of Regular, Randomized and Demand-Driven Departure Schedules for Trains in a Traffic Simulation“) selbständig verfasst wurde und dass keine anderen Quellen und Hilfsmittel, als die angegebenen benutzt wurden. Diese Aussage trifft auch für alle Implementierungen und Dokumentationen im Rahmen dieses Projektes zu.

Potsdam, den 7. Oktober 2023,

_____

(Christian J. Raue)