

Bachelorarbeit

**Vergleich von regelmäßigen,
randomisierten und bedarfsorientierten
Abfahrtsplänen von Zügen im Kontext einer
Verkehrssimulation**

**Comparison of regular, randomized, and on-demand train departure
schedules in the context of a traffic simulation**

Christian J. Raue

Hasso-Plattner-Institut an der Universität Potsdam

17. Juni 2023

Bachelorarbeit

Vergleich von regelmäßigen, randomisierten und bedarfsorientierten Abfahrtsplänen von Zügen im Kontext einer Verkehrssimulation

**Comparison of regular, randomized, and on-demand train departure
schedules in the context of 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

17. Juni 2023

Zusammenfassung

TESTTESTTEST Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Inhaltsverzeichnis

1	Einleitung	1
1.1	Strukturwandel in der Lausitz	2
1.2	Das Schienennetz der LEAG	3
1.3	FlexiDug - Gemeinsam Nutzung durch Personen- und Güterzüge	4
1.4	Das Softwareprojekt und die Interaktion im Team	5
2	Grundlagen	7
2.1	Fahrpläne	8
2.2	Abfahrtspläne	9
2.3	Design Patterns	10
2.3.1	Strategy Pattern	10
2.3.2	Template Method	12
2.3.3	Observer Pattern	13
2.3.4	Mediator Pattern	16
2.3.5	Visitor Pattern und Double Dispatch	17
2.3.6	Factory Method	19
2.4	Object-Relational-Mapping	21
3	Hauptteil	23
3.1	Die Berechnung des Kohlebedarfs	24
3.2	Architekturdiskussion	25
3.2.1	Die Datenbank	25
3.2.2	Der Spawner	25
3.2.3	Die Konfiguration des Spawners	27
3.2.4	Der Eventbus	28
3.3	Implementierungsdetails	32
3.3.1	Warteschlange für die Zugerzeugung	32
3.3.2	Erzeugung bedarfsorientierter Züge	32
3.4	Simulationsergebnisse	34
4	Schlussbetrachtung	35
4.1	Architekturübersicht	36
4.2	Ergebnisdiskussion	37
4.3	Ausblick	38
A	Anhang	39
	Literaturverzeichnis	39

1 Einleitung

1.1 Strukturwandel in der Lausitz

Die Lausitz, eine Region in Südbrandenburg, steht vor in naher Zukunft vor tiefgreifenden Veränderungen. Einer der wichtigsten Wirtschaftsfaktoren in der Region ist der Abbau von Braunkohle durch die LEAG AG in mehreren Tagebauen. 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. Die Region möchte weiterhin den Tourismus stark ausbauen durch Nachnutzung der entstehenden Flächen als Erholungsgebiet. Um das erreichen zu können ist es notwendig, die Tagebaulandschaft zu renaturieren 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. Es erkundet Nachnutzungsmöglichkeiten der Schieneninfrastruktur, 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 Kraftwerke ist die regelmäßige und zuverlässige Belieferung mit Braunkohle. Zu diesem Zweck betreibt die LEAG ein Schienennetz von 391 Kilometern Länge. Es verbindet die Tagebaue Jänschwalde, Welzow-Süd, Nochten und Reichwalde mit den Braunkohlekraftwerken Jänschwalde, Boxberg und Schwarze Pumpe. Außerdem ist der Kohleveredlungsbetrieb in Schwarze Pumpe angeschlossen. Gleichzeitig fahren 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. Mit ca. 1600 Tonnen pro Zug erreichen sie eine Maximalgeschwindigkeit von 50 km/h und einen Bremsweg von 400 Metern. Zum Fuhrpark gehören 61 E-Loks und 14 Diesel-Loks.

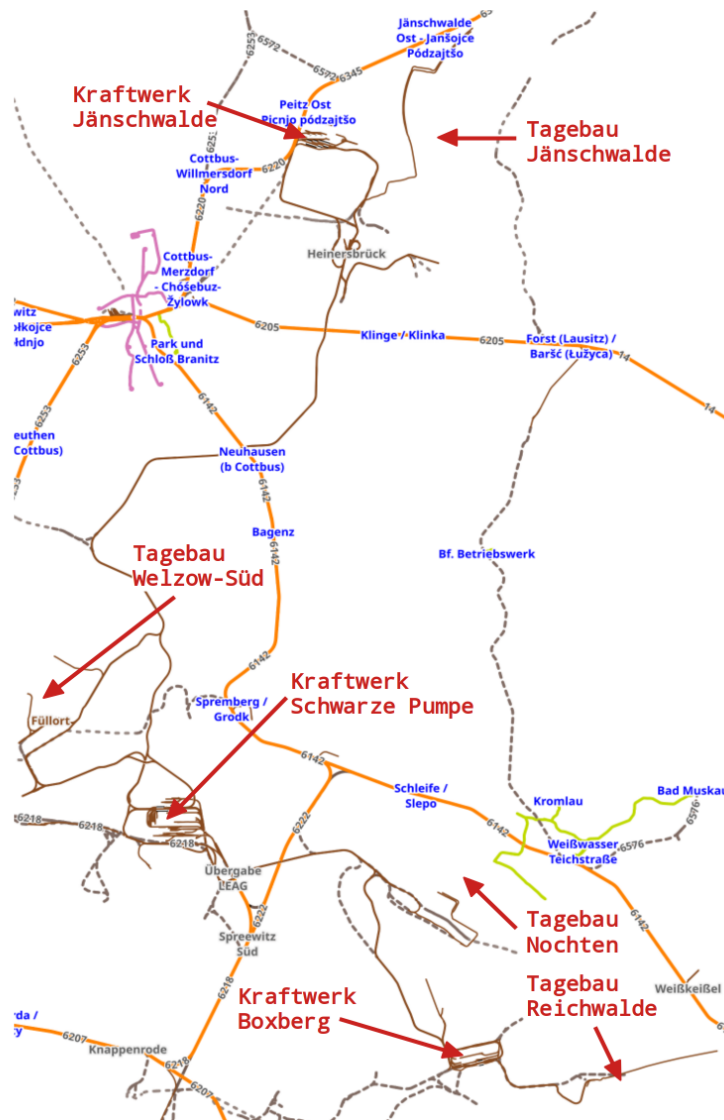


Abbildung 1.1: LEAG Netz

1.3 FlexiDug - Gemeinsam Nutzung durch Personen- und Güterzüge

Nach dem Kohleausstieg würde das Potential des Schienennetzen ohne ein Nachnutzungskonzept ungenutzt bleiben. Das Projekt FelxiDug (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.

Hier wird noch weiter geschrieben.

1.4 Das Softwareprojekt und die Interaktion im Team

2 Grundlagen

2.1 Fahrpläne

2.2 Abfahrtspläne

2.3 Design Patterns

Nach dem Architekten Christopher Alexander lässt sich ein Design Pattern 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."

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

Ein Design Pattern besteht dabei immer aus vier Elementen, seinem Namen, einer Problembeschreibung, einer Lösung und den sich ergebenden Konsequenzen. Die Problembeschreibung gibt dabei an, in welchen Fällen ein Pattern sich 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 Pattern bringt Vor- und Nachteile mit sich und geht Kompromisse zwischen verschiedenen Qualitätsaspekten ein. Die Konsequenzen beleuchten diese und helfen bei der Argumentation für oder gegen eine konkrete Designentscheidung.

Im Folgenden werden die abstrakten Design Patterns beschrieben, welche in der Architekturdiskussion eine Rolle spielen werden.

2.3.1 Strategy Pattern

Problembeschreibung

Es kann vorkommen, dass mehrere Varianten eines Algorithmus benötigt werden, welche ein einheitlichen Interface 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 ihrem Verhalten unterscheiden. Weiterhin werden Implementierungsdetails und Algorithmus-spezifische Daten vom Kontext abgekapselt.

Lösung

Der Context besitzt eine Referenz auf eine Strategie. Die konkrete Strategie wurde dem Context zuvor durch den Client zugewiesen. Die konkrete Strategie realisiert das Strategy-Interface, sodass Context und konkrete Strategie lediglich lose gekoppelt sind. Jede Strategie stellt die Methode 'execute' bereit, um den gekapselten Algorithmus auszuführen.

Der Context wird initialisiert, indem ihm vom Client eine konkrete Strategie zugewiesen wird (3), welche zuvor vom Client erstellt wurde (1). Der Client kann von nun an den Context dazu veranlassen, eine Aktion durchzuführen, welche die Verwendung des in der

hinterlegten Strategie hinterlegten Algorithmus nach sich zieht (5). Der Context kann den Algorithmus ausführen (6), ohne Wissen über die konkrete Strategie zu benötigen.

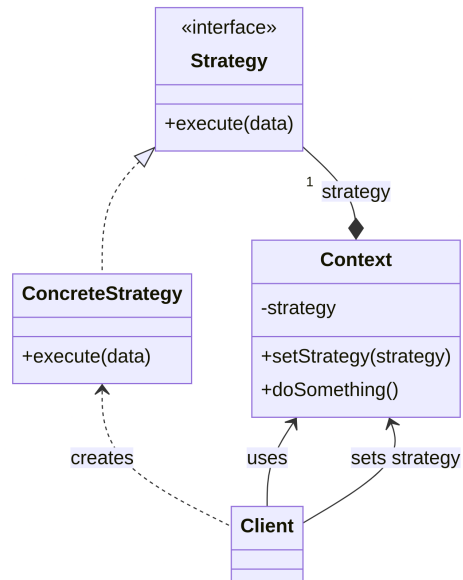


Abbildung 2.1: Klassendiagramm Strategy Pattern

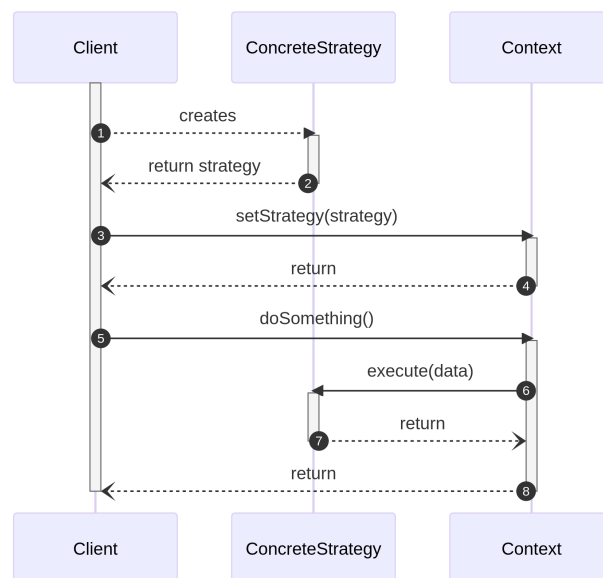


Abbildung 2.2: Sequenzdiagramm Strategy Pattern

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 Pattern 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 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 das gesamte Interface, um zu funktionieren. Da der Kontext allerdings nur das abstrakte Interface der Strategie kennt, muss er stets das gesamte Interface bedienen. Das bedeutet unnötige Aufrufe von Methoden. Weiterhin erhöht dieses Pattern die Gesamtzahl an Objekten und damit die Komplexität zur Laufzeit.

2.3.2 Template Method

Problembeschreibung

Es existieren ein Algorithmus, welcher aus einer Menge von Teilschritten besteht. Die einzelnen Teilschritte sollen austauschbar gehalten werden, um ein flexibles Anpassen des Algorithmus zu ermöglichen. Die Reihenfolge der Teilschritte ist hingegen festgelegt.

Lösung

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

Quelltext 2.1: Quelltextunterschrift

```
class AbstractClass:
    def templateMethod(self):
        if self.step1():
            self.step2()
        self.step3()
```

Konsequenzen

Template Methods sind ein Mechanismus, welcher die Wiederverwendung von Code ermöglicht und kann somit der Codeduplikation entgegen wirken. 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 sie 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-Methoden und die abstrakten Methoden. Während die "HookMethoden 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 "HookMethoden ist das optional.

Die Template-Method synergisiert mit dem Strategy-Pattern. Einzelne Schritte eine Algorithmus können in einer Strategie-Klasse implementiert sein.

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. Wenn ein 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 könn-

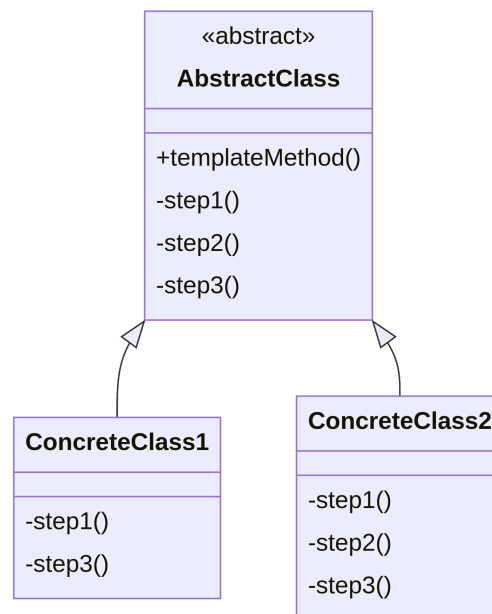


Abbildung 2.3: Klassendiagramm Template Method

ten dann in regelmäßigen Abständen prüfen, ob eine Zustandsänderung stattgefunden hat. 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. Der Aufwand für diese Prüfung steigt dabei linear mit der Anzahl der beteiligten Objekte.

Das Observer-Pattern kann Anwendung finden, wenn es zwei voneinander getrennte Konzepte gibt und eines von dem anderen abhängig ist. Die Abhängigkeit kann modelliert werden, ohne die Objekte zu koppeln. Weiterhin ist es möglich, die Anzahl der abhängigen Objekte variablen zu halten.

Lösung

Das Observer-Pattern besteht aus einem Publisher und mehreren Subscribern. Die Subscriber implementieren das Subscriber-Interface, welches eine Methode 'update' zur Aktualisierung des Zustandes bereitstellt. Der Publisher hält eine Liste von Referenzen auf Subscriber und verfügt über die Methoden 'subscribe' und 'unsubscribe', welche es ermöglichen, der Liste Subscriber hinzuzufügen, oder sie zu entfernen.

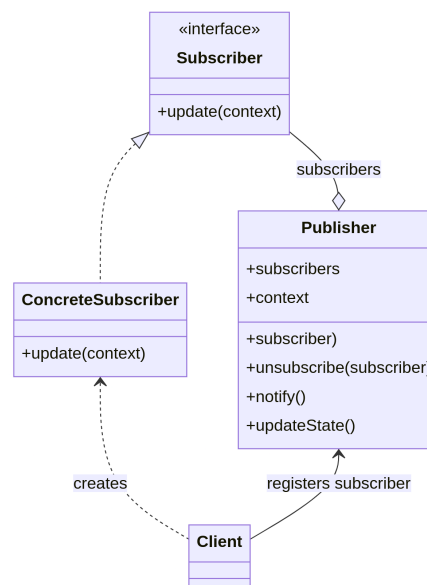


Abbildung 2.4: Klassendiagramm Observer

Der Client kann den Zustand des Publishers durch Senden von 'updateState' verändern (1). Der Publisher sendet sich draufhin selbst 'notify' (2) und beginnt über seine Liste von Subscribern zu iterieren. Jedem Subscriber sendet er dann 'update' (3, 5) und übergibt den notwendigen Kontext, sodass der Subscriber seinen Zustand entsprechend aktualisieren kann.

Der folgende Code veranschaulicht die Benachrichtigung aller Subscriber. In 'notify' wird an jeden im Publisher referenzierten Subscriber 'update' gesendet. Dabei wird der Kontext des Publishers an jeden Subscriber übergeben.

Quelltext 2.2: Quelltextunterschrift

```

class Publisher
    def notify(self):
        for subscriber in self.subscribers:
            subscriber.update(self.context)

```

Konsequenzen

Template Methods sind ein Mechanismus, welcher die Wiederverwendung von Code ermöglicht und kann somit der Codeduplikation entgegen wirken. Anstatt Das Observer-Pattern bietet eine Reihe von Vorteilen. Durch Separation von Publisher und Subscriber und durch die Abstraktion des Subscriber-Interfaces wird eine lose Kopplung der beiden erreicht. Diese lose Kopplung ermöglicht es, sowohl den Publisher, als auch die Subscriber beliebig auszutauschen. Weiterhin können sich der Publisher und die Subscriber auf unterschiedlichen Abstraktionsniveaus befinden. Ein Publisher auf einem niedrigen Level kann einen Subscriber auf einem hohen Level benachrichtigen. Wären Subscriber und Publisher nicht getrennt, so wäre dafür ein Abstraktionshierarchie-übergreifendes Objekt notwendig, welches die Trennung der Abstraktionsschichten beeinträchtigen würde. Ein weiterer Vorteil ist die dynamische Anzahl der Subscriber, mit denen ein Publisher interagieren kann. So kann der Client über den Publisher eine beliebige Zahl an Subscribern erreichen.

Ein Nachteil des Observer-Patterns ist, dass der Publisher stets alle seine Subscriber benachrichtigt. Es kann vorkommen, dass nur eine Teilmenge der Subscriber die Benachrichtigung benötigt, was in unnötigen Methodenaufrufen resultiert.

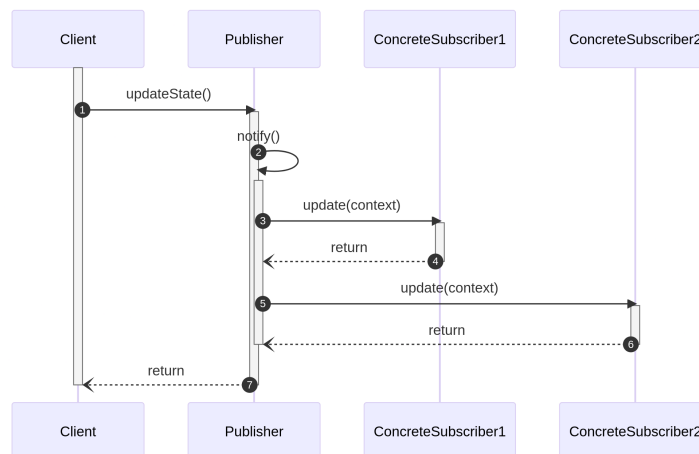


Abbildung 2.5: Sequenzdiagramm Observer

2.3.4 Mediator Pattern

Problembeschreibung

Große Softwareprojekte bestehen meist aus einer enormen Anzahl an Klassen bzw. Objekten, welche miteinander interagieren. Ziel ist es stets, die Kopplung zwischen diesen Komponenten so gering wie möglich zu halten. Komplexe Interaktionsmuster zwischen diesen Objekten lassen sich nicht immer verhindern, das 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.

Lösung

Der konkrete Mediator implementiert das Mediator-Interface, welches eine Methode 'notify' bereitstellt, um Benachrichtigungen der einzelnen Komponenten 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 des übergebenen Senders 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.

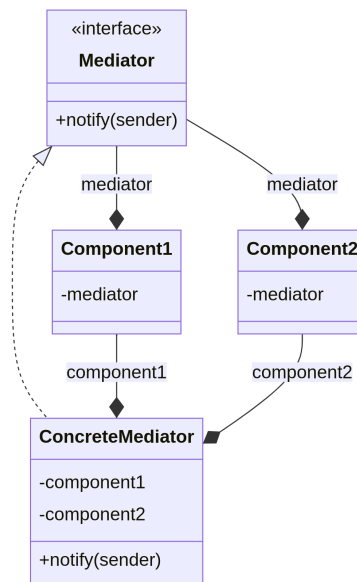


Abbildung 2.6: Klassendiagramm Mediator Pattern

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 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.

2.3.5 Visitor Pattern und Double Dispatch

Problembeschreibung

Gelegentlich muss eine Operation auf einer Menge von Objekten durchgeführt, die zwar all Teil einer Aggregationshierarchie aber dennoch unterschiedlich sind. Diese Objekte besitzen unter Umständen voneinander abweichende Interfaces. Das Visitor-Pattern erlaubt es, solche Operationen außerhalb der Objekte und für alle betroffenen Objekte innerhalb einer Klasse zu definieren.

Lösung

Der konkrete Visitor realisiert das Visitor-Interface, welches die Methode 'visit' bereitstellt. Diese erlaubt es dem Visitor, ein Element zu "besuchen" und auf ihm eine Operation durchzuführen. Die Elemente "akzeptieren" den "Besuch" des Visitors mit Hilfe der Methode 'accept', welche als Argument den besuchenden Visitor erhält.

Der Client 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. Zu beachten ist hierbei, dass es für jede Element-Klasse eine eigene Methode im Visitor gibt. Dies kann realisiert werden durch das Bereitstellen von Methoden mit unterschiedlichen, zu den aufrufenden Klassen korrespondierenden Namen oder durch Multimethoden. 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

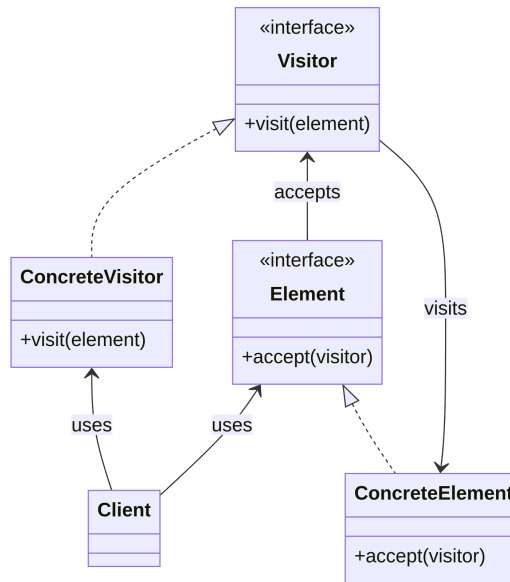


Abbildung 2.7: Klassendiagramm Visitor

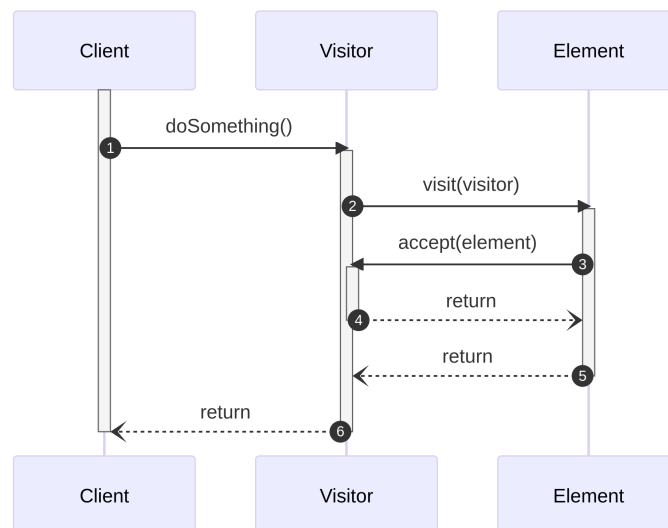


Abbildung 2.8: Sequenzdiagramm Visitor

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-Pattern sollte daher nur verwendet werden, wenn entweder die Menge an Elementklassen abgeschlossen oder die Menge an Visitor-Klassen übersichtlich ist. Weiterhin müssen die Elemente dem Visitor ein Interface bereitstellen, welches 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 Patterns gekapselt geblieben wäre.

2.3.6 Factory Method

Problembeschreibung

Es wird ein Interface benötigt, um eine Reihe von konkreten Produkten erzeugen zu können. Jedes konkrete Produkt hat jedoch andere Anforderungen an seine eigene Erzeugung. Eine Factory-Method kann eingesetzt werden, wenn eine Klasse kein Wissen darüber besitzt oder besitzen soll, welches konkrete Produkt sie zu erzeugen hat oder eine Klasse die Verantwortlichkeit über diese Entscheidung ihren Subklassen überlassen möchte.

Lösung

Es existiert eine abstrakte 'Creator'-Klasse, welche ein Interface bereitstellt, um Produkte zu erzeugen. Die Details der Erzeugung dieser Produkte sind in den konkreten Subklassen von 'Creator' implementiert. Jeder konkrete 'Creator' kann somit einen Typ von konkretem Produkt erschaffen.

Konsequenzen

Ein Objekt über eine Factory-Method zu erzeugen ist flexibler, als das Objekt direkt über den Konstruktor der Klasse zu instanziiieren. Die erzeugende Klasse braucht nur das Interface des abstrakten 'Creator' zu kennen und ist somit in der Lage beliebige konkrete Produkte über deren korrespondierende konkrete 'Creator' zu erzeugen. Hierbei fällt auf, dass die 'Creator'-Klassenhierarchie die Produkt-Klassenhierarchie spiegelt. Für jeden Produkttyp existiert also auch eine 'Creator'-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.

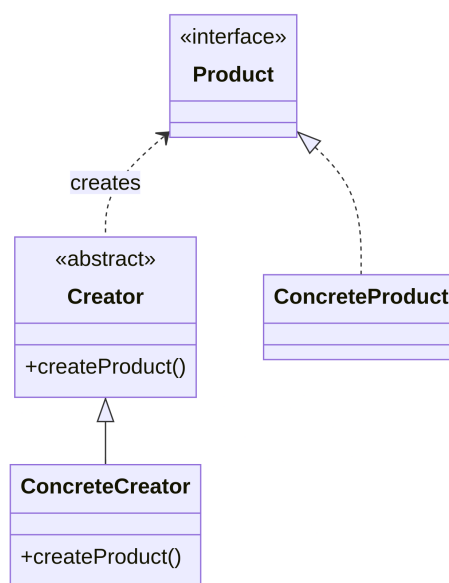


Abbildung 2.9: Klassendiagramm Factory Method

2.4 Object-Relational-Mapping

3 Hauptteil

3.1 Die Berechnung des Kohlebedarfs

3.2 Architekturdiskussion

3.2.1 Die Datenbank

3.2.2 Der Spawner

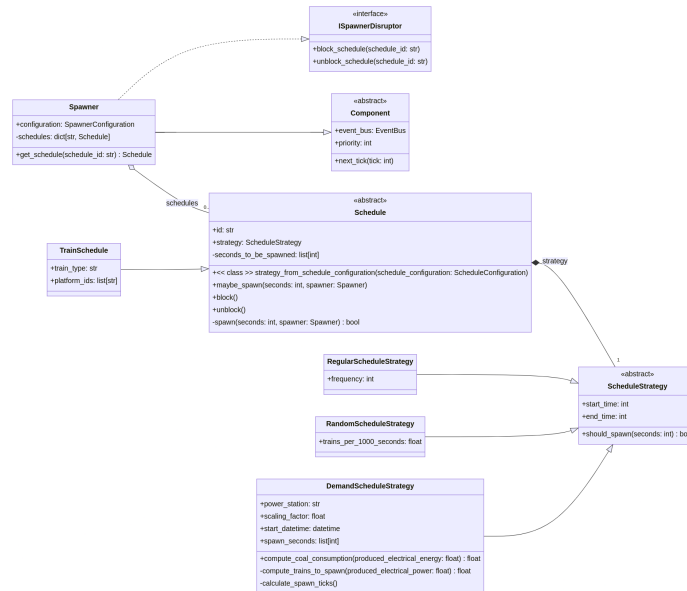


Abbildung 3.1: Klassendiagramm Spawner

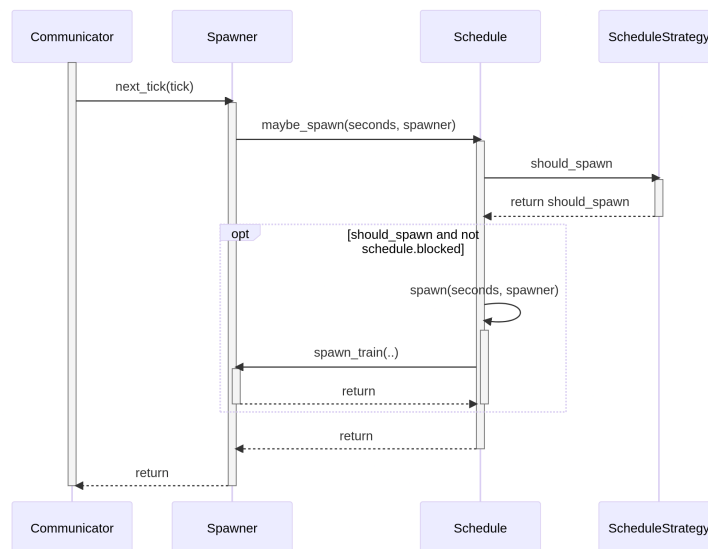


Abbildung 3.2: Sequenzdiagramm Spawner

Unser System simuliert die Interaktion von Zügen und diese Züge müssen zu bestimmten Zeiten instanziiert werden. Jedoch könne wir keine SUMO-Flows für das Spawning verwenden, weil diese nicht in der Lage sind, das von uns gewünschte Fahrstraßenverhalten abzubilden. Deshalb haben wir die Spawner-Komponente entwickelt, welche mit Hilfe einer Liste von Stationen, einen Zug in die Simulation setzt.

Der **Spawner** erbt wie die meisten anderen Teile der Software von **Component**, um über *'next_tick'* auf die nächste Iteration der Simulation reagieren zu können. Außerdem erhält der Spawner somit Zugriff auf den **EventBus**, welcher es ermöglicht, Events zu emittieren und auf emittierte Events zu reagieren.

Der **Spawner** hält eine Liste von **Schedules**. Ein **Schedule** repräsentiert eine Menge von SUMO-Entitäten, den Weg, den diese in der Simulation zurücklegen sowie die Zeitpunkte, zu denen die Entitäten instanziiert werden. Weiterhin kann ein **Schedule** vom **FaultInjector** geblockt werden, um die Erzeugung von Entitäten für einen bestimmtes Zeitintervall zu unterbinden.

Für einen **Schedule** können Spezialisierungen angelegt werden, um auf die Besonderheiten verschiedener Arten von Entitäten Rücksicht zu nehmen. So müssen z.B. bei der Erzeugung von Fußgängern zum Teil andere Faktoren berücksichtigt werden, als bei der Erzeugung von Zügen. Daher haben wir uns entschieden, die Spezialisierung **TrainSpawner**, zur Erzeugung der Züge zu verwenden. Ein **PedestrianSpawner** war geplant, konnte jedoch im Rahmen des Projektes nicht mehr umgesetzt werden. Der **TrainSchedule** enthält zusätzlich Informationen über den Zugtyp (z.B. Personenzug oder Güterzug), die Route, die der Zug abfährt und dessen Haltepunkte. Diese Route ist als Liste von IDs hinterlegt, welche die anzufahrenden Bahnsteige referenzieren.

Im Rahmen unseres Projektes untersuchten wir drei verschiedene Möglichkeiten, die Erzeugungszeitpunkte für Entitäten festzulegen. Dies waren eine regelmäßige, eine zufällige und eine bedarfsorientierte Erzeugung. Ein naiver Ansatz, um diese Möglichkeiten in der Architektur widerzuspiegeln, ist es, Vererbung zu verwenden. Dieser Ansatz ist allerdings nicht skalierbar, da n Möglichkeiten der Festlegung der Zeitpunkte und m Arten von Entitäten zu $m \cdot n$ verschiedenen Spezialisierungen führen würden. Das würde zu einem quadratischen Anwachsen von Spezialisierungsklassen führen, wenn weitere Optionen hinzugefügt werden würden.

Stattdessen haben wir uns dazu entschieden, das Strategie-Entwurfsmuster zu verwenden. Wie bereits erwähnt, existieren Spezialisierungen für jede Art von SUMO-Entität. Die Erzeugungszeitpunkte werden über Strategieklassen realisiert, welche eine Spezialisierung von **ScheduleStrategy** sind. Jeder **Schedule** hat nun eine Referenz auf eine **ScheduleStrategy**. Wird *'next_tick'* auf dem **Spawner** aufgerufen, so iteriert er über seine Liste von **Schedules** und veranlasst jeden davon, einen Zug zu erzeugen, falls entsprechende Bedingungen erfüllt sind. Dazu wird das Besucher-Entwurfsmuster verwendet, bei welchen der **Spawner** mit jedem **Schedule** über einen Double-Dispatch interagiert, indem der **Spawner** sich selbst übergibt. Dies wird durch Aufrufen von *'maybe_spawn'* initiiert. Die Zuständigkeit für die Entscheidung, ob im aktuellen *'tick'* ein Zug erzeugt werden soll, liegt bei der **ScheduleStrategy**. Die entsprechende konkrete **ScheduleStrategy** prüft die vorliegenden Bedingungen und gibt im Anschluss ihre Entscheidung als **Boolean** zurück.

Fällt diese Entscheidung positiv aus und ist der **Schedule** nicht blockiert, so ruft er die private *'spawn'*-Methode seiner Superklasse auf, auf deren Funktionalität unter

****Implementierung**** im Detail eingegangen wird. Letztendlich kehrt die Ausführung zum ***Spawner*** zurück, welcher die Erzeugung des Zuges veranlasst.

3.2.3 Die Konfiguration des Spawners

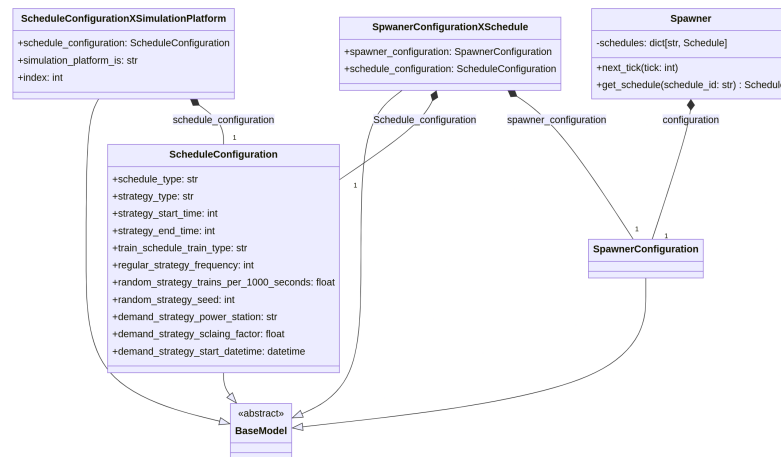


Abbildung 3.3: Klassendiagramm Spawnerkonfiguration

Der ***Spawner*** und insbesondere dessen ***Schedules*** und ***ScheduleStrategies*** müssen konfiguriert werden. Zum Zweck der Nachvollziehbarkeit und der Reproduzierbarkeit, müssen diese Konfigurationen persistent gespeichert werden. Dazu werden sie als Objekte in der Datenbank abgelegt.

Die initiale Idee zur Umsetzung dessen, war die konkreten ***Schedules*** und ***ScheduleStrategies*** als Spezialisierung von ***BaseModel*** anzulegen und somit eine Persistenz dieser Objekte in der Datenbank zu ermöglichen. Die hätte für den ***Schedule*** eine Vererbungshierarchy der Form ***BaseModel* <- *Schedule* <- *Konkreter Schedule*** zur Folge gehabt (analog für die ***ScheduleStrategies***). Das von uns verwendete ORM setzt abstrakte Klassen in Form konkret existierender Datenbankrelationen um. Diese Relationen wären entsprechend stets leer. Da jedoch zur Laufzeit Variablen abstrakten Typs auf eben diese Relationen abgebildet werden, war diese Entwurfsidee keine Option für uns.

Stattdessen haben wir uns dazu entschieden, die Konfiguration von der Logik zu trennen und für die Konfiguration auf Vererbung zu verzichten. Dies löst das Problem der abstrakten Klassen mit dem ORM und ermöglicht weiterhin die Verwendung von Spezialisierung für die Funktionalität.

Die Konfiguration der ***Schedules*** und ***ScheduleStrategies*** ist nun in einer einzigen Klasse, der ***ScheduleConfiguration***, zusammengefasst. Dies erschien sinnvoll, da eine 1:1-Beziehung zwischen beiden Klassen besteht.

Zwischen dem ***Spawner*** und dem ***Schedule*** besteht eine m:n-Beziehung, da ein Spawner mehrere Schedules besitzen kann. Ebenso kann ein Schedule in mehreren ***Spaw-**

ners* Verwendung finden. Die Beziehung zwischen der **ScheduleConfiguration** und der **SpawnerConfiguration** wird daher über die Hilfsklasse **SpawnerConfigurationXSchedule** realisiert. Diese Klasse ist leer und lediglich eine technische Notwendigkeit zur Spezifikation der m:n-Beziehung für den ORM.

Ein Schedule benötigt eine Liste von Stationen, welche der Zug nacheinander ansteuern soll. Er wird an der ersten dieser Stationen erzeugt und wird an der letzten Station aus der Simulation entfernt. Diese Stationen existieren nur im Laufzeitkontext, sind also nicht in der Datenbank abgelegt. Die Klasse **ScheduleConfigurationXSimulationPlatform** repräsentiert die m:n-Beziehung zwischen einer Station und einen **Schedule**. Da m:n-Beziehungen in relationalen Datenbankmodellen ungeordnet sind, wird jedem Element dieser Beziehung ein Index zugeordnet, um eine Totalordnung über alle Elemente aufzubauen. Diese legt die Reihenfolge der Stationen fest.

Die Konstruktion der **Schedules** und **ScheduleStrategies** erfolgt durch die Factory-Methoden **strategy_from_schedule_configuration** im abstrakten **Schedule** und **from_schedule_configuration** in den entsprechenden konkreten Strategieklassen.

3.2.4 Der Eventbus

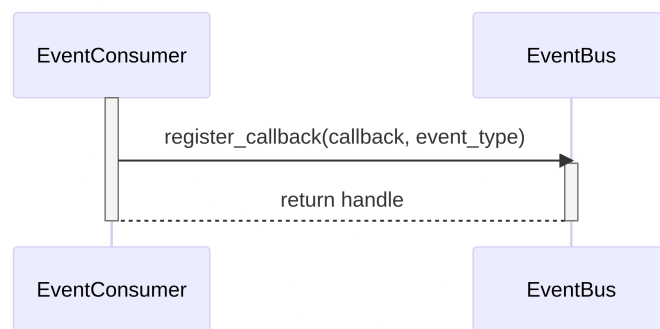


Abbildung 3.4: Sequenzdiagramm Registrierung Eventbus

Variante 1

Variante 2

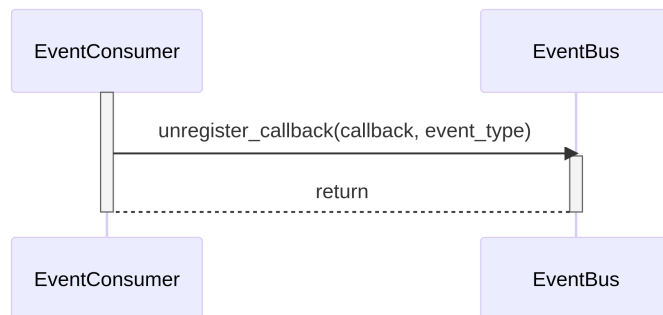


Abbildung 3.5: Sequenzdiagramm Deregistrierung Eventbus

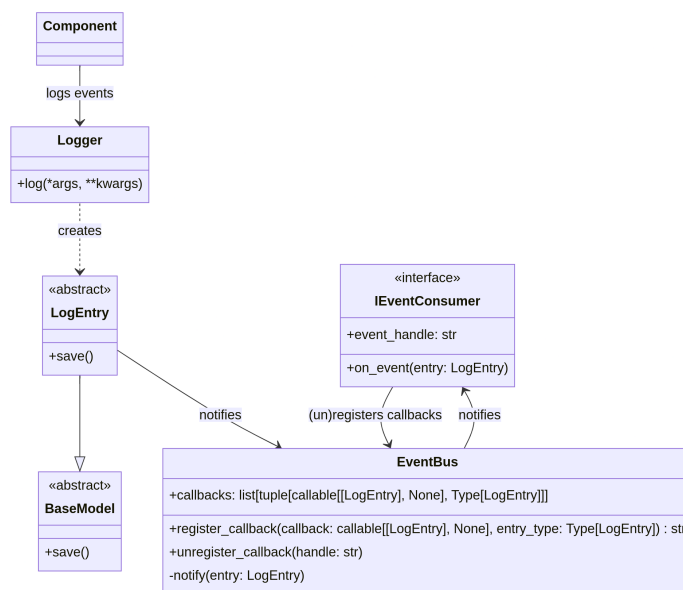


Abbildung 3.6: Klassendiagramm Variante 1 Eventbus

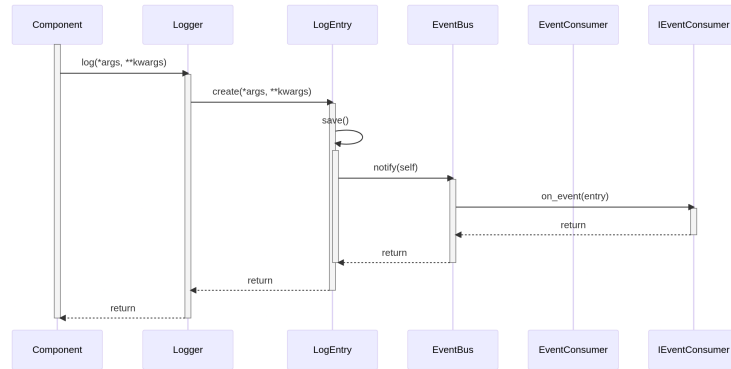


Abbildung 3.7: Sequenzdiagramm Variante 1 Eventbus

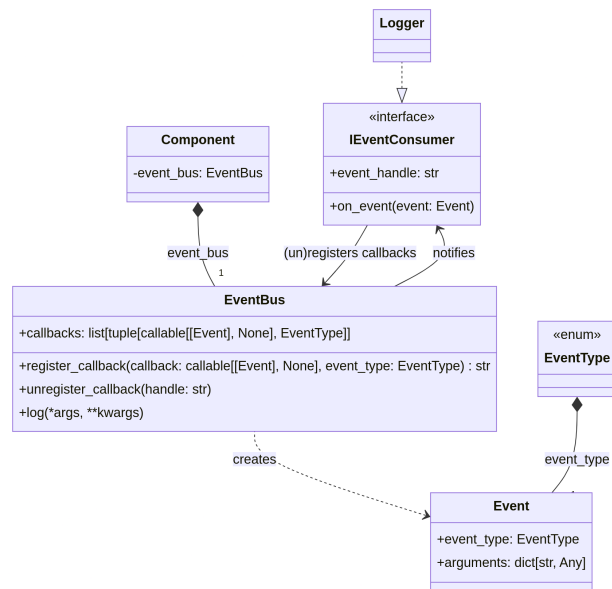


Abbildung 3.8: Klassendiagramm Variante 2 Eventbus

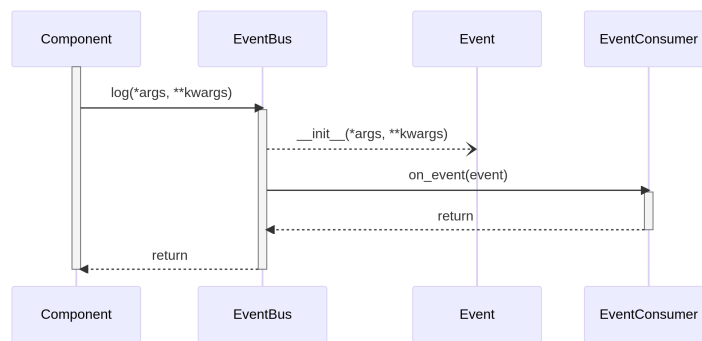


Abbildung 3.9: Sequenzdiagramm Variante 2 Eventbus

3.3 Implementierungsdetails

3.3.1 Warteschlange für die Zugerzeugung

Quelltext 3.1: Quelltextunterschrift

```
class Schedule(ABC):
    _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 Erzeugung bedarfsorientierter Züge

Quelltext 3.2: Quelltextunterschrift

```
class DemandScheduleStrategy(ScheduleStrategy):
    SECONDS_PER_QUARTER_HOUR: int
    COAL_PER_TRAIN: float

    start_datetime: datetime
    spawn_seconds: list[int]

    def __init__(self, ...):
        ...
        self._calculate_spawn_seconds()
        ...

    def _compute_trains_to_spawn(self, produced_electrical_power: float) -> float:
        coal_consumption = self.compute_coal_consumption(produced_electrical_power)
        return coal_consumption / self.COAL_PER_TRAIN

    def _calculate_spawn_seconds(self):
        end_datetime = self.start_datetime + timedelta(
            seconds=self.end_time - self.start_time
        )
        data = self._api.get_data(self.start_datetime, end_datetime)
        train_accumulator = 0.0
        for quarter_hour, entry in enumerate(data):
            train_accumulator += self._compute_trains_to_spawn(entry.value)
```



```
seconds = int(quarter_hour * self.SECONDS_PER_QUARTER_HOUR + self.start_time)
while train_accumulator >= 1.0:
    self.spawn_ticks.append(seconds)
    train_accumulator -= 1.0
    tick += 1

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

3.4 Simulationsergebnisse

4 Schlussbetrachtung

4.1 Architekturübersicht

4.2 Ergebnisdiskussion

4.3 Ausblick

A Anhang

Eins (Interview mit Sascha Lesche von der LEAG)

Christian Raue: Lassen Sie uns zunächst zum Thema des Kohletransportes kommen. Wie viele Tonnen Kohl 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 960 Tonnen.

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

Sascha Lesche: Der Energiegehalt der Kohle schwankt sehr stark. 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. 230 Metern.

Max Lietze: Wie schnell kann ein Kohlezug fahren?

Sascha Lesche: Die Loks schaffen eine Geschwindigkeit von 60 km/h, dürfen aber aufgrund der Leit- und Sicherungstechnik maximal 50 km/h fahren. Dabei werden wie im klassischen Zugverkehr auch Gefahrpunktabstände berücksichtigt. Da wir jedoch über keine klassischen Durchrutschwege und auch keine Zugbeeinflussung verfügen, ist die Geschwindigkeit auf 50 km/h begrenzt.

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 Mter pro Quadratsekunde und für das Abbremsen 0,4 bis 0,5 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: Wieviele 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.

Eidesstattliche Erklärung

Hiermit versichere ich, dass meine Bachelorarbeit „Vergleich von regelmäßigen, randomisierten und bedarfsorientierten Abfahrtsplänen von Zügen im Kontext einer Verkehrssimulation“ („Comparison of regular, randomized, and on-demand train departure schedules in the context of 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 17. Juni 2023,

(Christian J. Raue)