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 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 Kohleveredelungsbetrieb 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.

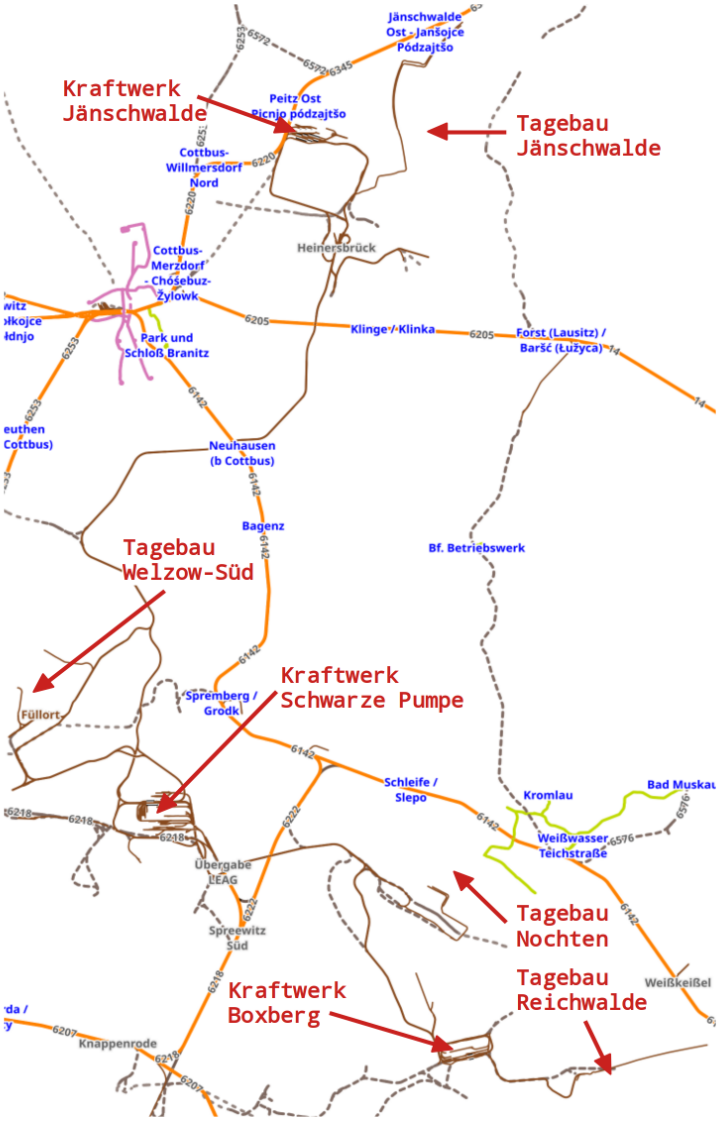

Abbildung 1.1: LEAG Netz

1.3 FlexiDug - Geminsam 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:

Ëach 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 Iinterface 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.

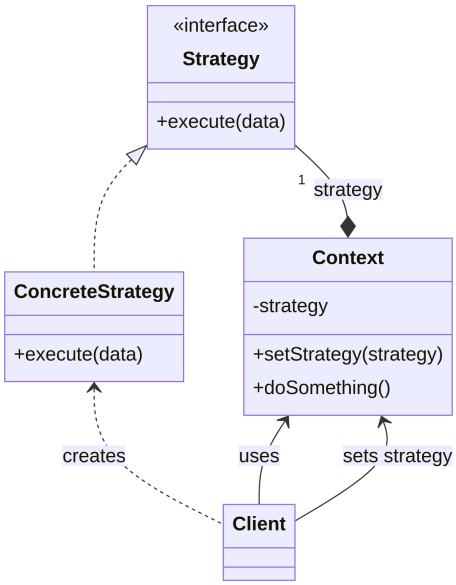

Abbildung 2.1: Klassendiagramm Strategy Pattern

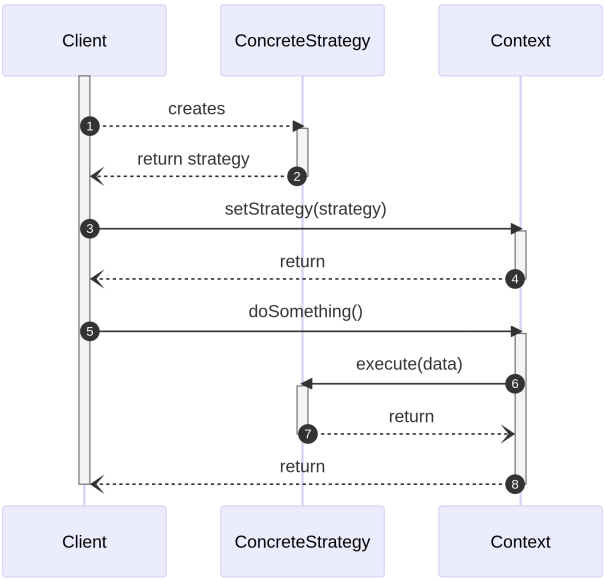

Abbildung 2.2: Sequenzdiagramm Strategy Pattern

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 Aggreation 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 synergiert 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-

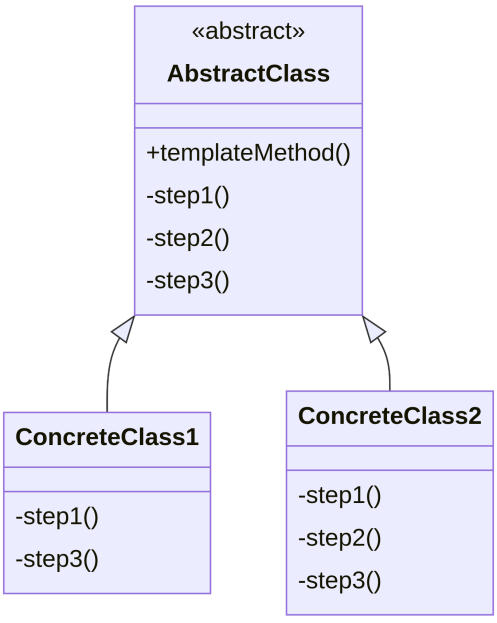

Abbildung 2.3: Klassendiagramm Template Method

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.

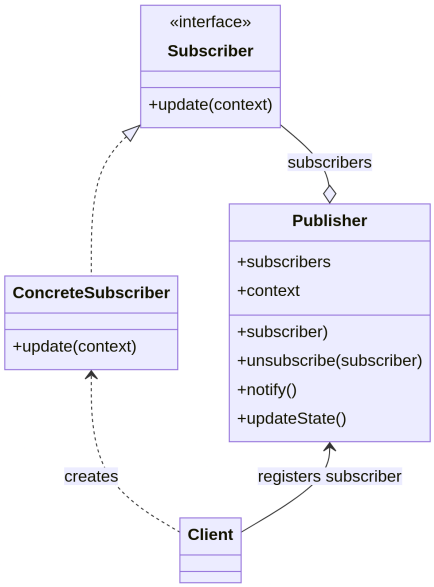

Abbildung 2.4: Klassendiagramm Observer

Der Client kann den Zustand des Publishers durch Senden von ‘updateSate‘ 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.

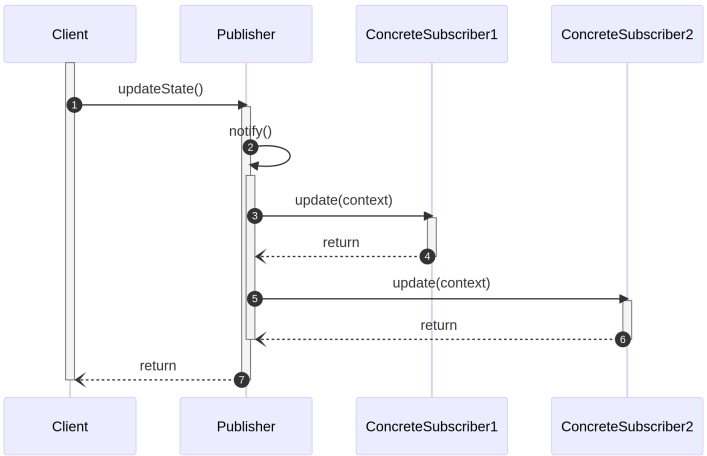

Abbildung 2.5: Sequenzdiagramm Observer

2.3.4 Mediator Pattern

Problembeschreibung

Große Softwareprojekte bestehen meist aus einer 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 wiederspiegeln. In diesem Fall ist eine Lösung notwending, 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.

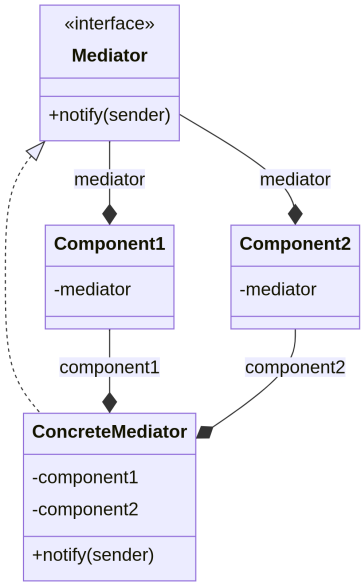

Abbildung 2.6: Klassendiagramm Mediator Pattern

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 Aggregationshiearchie 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 "besuchenünd auf ihm eine Operation durchzuführen. Die Elemente äkzeptieren"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

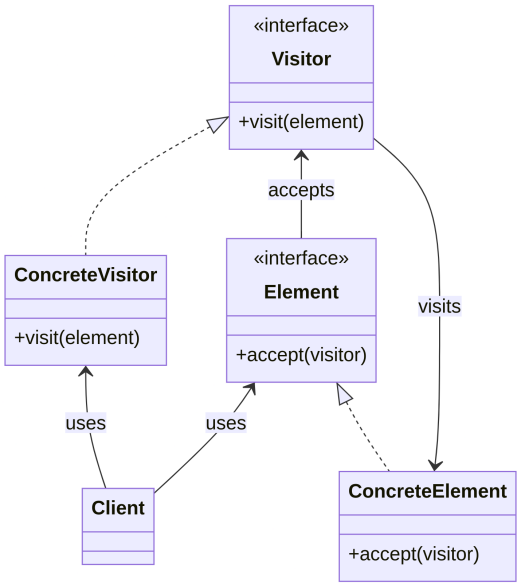

Abbildung 2.7: Klassendiagramm Visitor

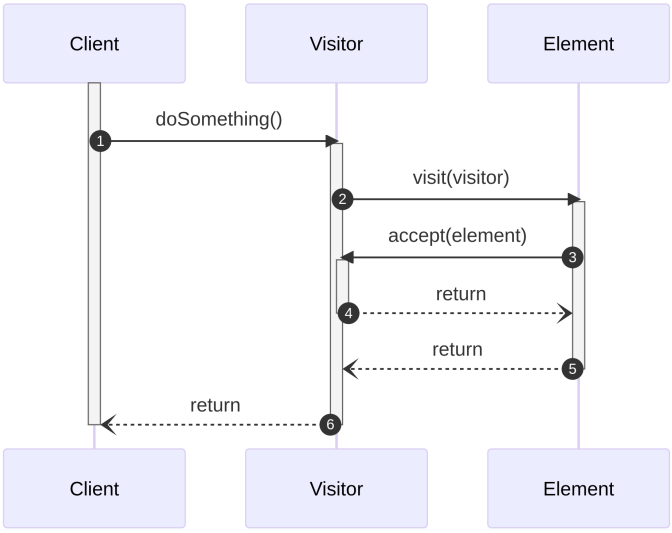

Abbildung 2.8: Sequenzdiagramm Visitor

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 ein 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 instanziieren. 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.

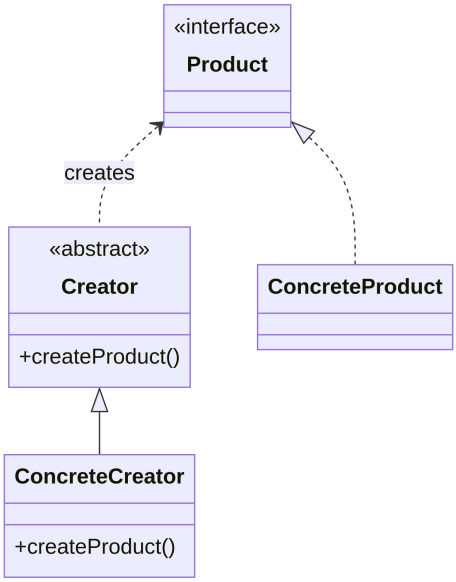

Abbildung 2.9: Klassendiagramm Factory Method

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

3.2.3 Die Konfiguration des Spawners

3.2.4 Der Eventbus

3.3 Implementierungsdetails

3.4 Simulationsergebnisse

4 Schlussbetrachtung

4.1 Architekturübersicht

4.2 Ergebnisdiskussion

4.3 Ausblick

A Anhang

Eins (ohne extra Eintrag im Inhaltsverzeichnis)

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.

Zwei (ohne extra Eintrag im Inhaltsverzeichnis)

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.

Drei (ohne extra Eintrag im Inhaltsverzeichnis)

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.

Vier (ohne extra Eintrag im Inhaltsverzeichnis)

Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

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)