Abschlussarbeit

Entwurf und prototypische Realisierung einer
web- und graphdatenbankbasierten Visualisierungsplattform
für Softwarearchitekturen aus Systemwelten
von unterschiedlichen Programmiersprachen

zur Erlangung des akademischen Grades

Bachelor of Science

Im Studiengang Ingenieurinformatik
Fachbereich 2 - Ingenieurwissenschaften
Hochschule für Technik und Wirtschaft Berlin

Vorgelegt von	**Cynthia Rapp** 557972
Gutachter	Prof. Dr.-Ing. Jörg Schlingheider M.Sc. Lynn von Kurnatowski
Bearbeitungszeitraum	01.07.-20.09.2019
In Kooperation mit	Deutsches Zentrum für Luft- und Raumfahrt Simulations- und Softwaretechnik

Inhaltsverzeichnis

Zusammenfassung

Eine Visualisierung, mit der sich die Architektur von Software zu verschiedenen Zeitpunkten ihrer Entwicklung darstellen lässt, hilft das Verständnis für die Software zu verbessern und kann im Verlauf der Entwicklung als Kommunikationsgrundlage dienen. Die Softwarearchitektur ist abhängig von der Programmiersprache, in der sie implementiert wurde. Durch die Integration des Datensatzes, der die Architektur abbildet, in die Graphdatenbank *Neo4j* ergeben sich je nach Sprache unterschiedliche Datenbankschemata. Auf dieses Schema greift das System hinter der webbasierten Visualisierungsplattform zurück.

In dieser Arbeit werden Ansätze zur Generalisierung einer datensatzgebundenen Plattform zur Visualisierung von Software-Evolutionsgraphen untersucht, um die effektivste unter ihnen weiter zu untersuchen und umzusetzen. Ziel ist es, die Bindung dieser Plattform an das Datenbankschema und die Programmiersprache aufzuheben und durch eine universell anwendbare Lösung zu ersetzen. In der Konzeption der Generalisierung wurde ein Implementierungsansatz entwickelt, bei dem der Nutzer Angaben für die Konfiguration der Webanwendung vorgibt, damit diese im Anschluss korrekt generiert wird. Die anschließende Umsetzung der Generalisierung wurde erfolgreich implementiert, sodass es dem Nutzer möglich ist, neue Architekturen zu integrieren und zu visualisieren. Das entwickelte System beantwortet die zentrale Fragestellung der Arbeit, wie die Visualisierung flexibel auf verschiedene Schemata reagieren kann.

Abkürzungsverzeichnis

CSS Cascading Style Sheets

CSV Comma-Separated Values

HTML Hypertext Markup Language

OSGi Open Services Gateway initiative

RDBMS Relational Database Management System

SQL Structured Query Language

UML Unified Modeling Language

URL Uniform Resource Locator

XML Extensible Markup Language

1 Einleitung

In diesem ersten Kapitel werden die Motivation zum Verfassen sowie die zentrale Fragestellung und Ziele dieser Arbeit dargelegt. Im Anschluss folgt ein kurzer Überblick über den Aufbau und das weitere Vorgehen in der Arbeit.

1.1 Motivation

Das Ziel einer Visualisierungsplattform für die Darstellung der Architektur von Software besteht darin, dem Nutzer den Aufbau und die Struktur einer Software eindeutig und verständlich näher zu bringen und als Kommunikationsgrundlage zu dienen. Ein Verständnis von der Architektur der vorliegenden Software zu entwickeln steht im Fokus. Um dieses weiter zu verstärken und zu verbessern, werden zu den herkömmlichen UML (*Unified Modeling Language*)-Diagrammen andere Arten von Visualisierungen geschaffen. Dafür gibt es bereits einige sinnvolle und qualitative Tools auf dem Markt [1]. Dagegen werden hier, anders als bei vielen dieser Plattformen, die Daten aus einer Datenbank extrahiert, in welcher mehrere Versionen hinterlegt sind, um die Historie der Softwarearchitektur darstellen zu können.

Wenn man die Evolution der Architektur einer Software darstellen kann, trägt dies nicht nur zum Verständnis des gesamten Systems bei, es bietet zusätzlich mehr Möglichkeit zur Analyse des Entwicklungsprozesses und der Software selbst. In dieser Arbeit liegt der Fokus allerdings nicht auf der Evolution der Softwarearchitektur, sondern auf der Generalisierung eines Visualisierungstools zum Einbinden diverser Architekturen. Der Begriff „Generalisierung“ wird im Kontext dieser Arbeit als die Aufhebung der Datensatzbindung einer Visualisierungssoftware und Erweiterung zur Integration diverser Architekturmodelle verstanden.

Die Architektur einer Software ist abhängig von der Programmiersprache in der sie implementiert wurde. Programmiersprachen folgen einem bestimmten Aufbau, der bei jeder anders sein kann. *Java* beispielsweise verwendet *Packages* für die Strukturierung und Organisation seiner Module. Wird dann zusätzlich das *Open Services Gateway initiative* (OSGi) Framework integriert, kommen *Services* und *Bundles* hinzu und die Struktur des Programmcodes verändert sich drastisch. Hingegen verwendet beispielsweise C# sogenannte *namespaces*, um ihre Programmelemente zu strukturieren und abzugrenzen.

Durch die Ungleichheit der Struktur und des Aufbaus von verschiedenen Programmen ist es nicht möglich, ein einheitliches Schema zu definieren. Dazu kommt die Tatsache, wie unterschiedlich jeder Programmierer mit den jeweiligen Sprachen arbeitet und die Ergebnisse in die Datenbank integriert. Die Daten, die in einer Datenbank abgelegt werden, müssen strukturiert und benannt werden. Dabei können schon sprachliche Unterschiede beim gleichen Datensatz ausreichen, um verschiedene Ergebnisse zu erhalten. Deshalb ist es wichtig, das Auslesen und Verarbeiten der Daten aus einer Datenbank zur Darstellung der Softwarearchitektur universell und sprachunabhängig zu gestalten.

1.2 Ziele und Aufgaben

Es werden Ansätze zur Generalisierung einer datensatzgebundenen Visualisierungsplattform untersucht, um die effektivste unter ihnen in einem Prototyp umzusetzen. Ziel ist es, die Bindung der Plattform an das Datenbankschema und die Programmiersprache aufzuheben und eine universell anwendbare Lösung zu entwickeln. Das Ergebnis wird zur Abnahme evaluiert und ein Fazit über den Erfolg gezogen.

Die zentrale Fragestellung der Arbeit lautet: Wie kann die Visualisierung flexibel auf verschiedene Schemata reagieren. Um diese Frage beantworten zu können, werden im ersten Teil der Arbeit notwendige Grundlagen zusammengetragen und die Ausgangssituation erfasst. Es sollte geklärt werden, welche Möglichkeiten für Softwarevisualisierungen existieren, und wie diese aufgebaut sind.

Der zentrale Teil der Arbeit wird die Beantwortung der Fragestellung sowie die prototypische Umsetzung einer möglichen Generalisierung. Dafür wird ein Konzept entwickelt, das zuerst die Zielgruppe und deren Interessen definiert. Die in der Ausgangssituation gegebenen statischen Aspekte der Software werden erfasst und kategorisiert. Im Anschluss werden verschiedene Möglichkeiten eines Generalisierungsansatzes beschrieben und auf ihre Machbarkeit geprüft. Der gewählte Ansatz wird im Anschluss für die folgende Implementierung weiter ausgeführt. Die funktionalen Anforderungen an die Umsetzung lauten: die statischen Aspekte zu generalisieren und dabei nicht weniger Ergebnisse zu erzielen.

Der letzte Teil dieser Arbeit beinhaltet eine Evaluierung der prototypischen Umsetzung des Konzepts. Dazu wird ein zweiter Datensatz mit anderer Architektur als der bereits enthaltene, in die Graphdatenbank integriert und mithilfe der entwickelten Plattform visualisiert. Dabei werden alle Funktionen der Visualisierungsplattform auf ihre Funktionalität getestet. Damit erfolgt ein Fazit über den Erfolg der Generalisierung und die erneute Beantwortung der zentralen Fragestellung.

1.3 Gliederung der Arbeit

In Kapitel 2 werden die notwendigen Grundlagen für die weitere Arbeit dargelegt, wobei der Fokus auf der Softwarevisualisierung und auf Graphdatenbanken liegt. Im Anschluss wird die Ausgangssituation dieser Arbeit erläutert. Basierend auf dem Wissen des Grundlagenkapitels folgt in Kapitel 3 das Konzept, indem die Zielgruppe definiert und abgegrenzt und anschließend auf die zu beachtenden Aspekte und auf mögliche Herausforderungen bei der Implementierung eingegangen wird. Ziel dieses Kapitels ist es, Umsetzungsmöglichkeiten für die Generalisierung der bestehenden Software zu sammeln und zu vergleichen, um die geeignetste unter ihnen für die im nachfolgenden Kapitel beschriebene prototypische Umsetzung zu finden. Diese wird in Kapitel 4 weiter ausgeführt und in Kapitel 5 evaluiert. Die Evaluierung des Prototyps besteht aus einem Softwaretest und der möglichen Abnahme bei Erfolg. Das letzte Kapitel enthält eine Zusammenfassung der Arbeit und einen Ausblick in die Zukunft.

2 Grundlagen

Nachdem die Hintergründe und der Kontext dieser Arbeit deutlich gemacht und Ziele definiert wurden, wird im Folgenden auf die nötigen Grundlagen für die weitere Arbeit eingegangen. Im ersten Unterkapitel wird der Begriff der Architektur von Software definiert, um ein gemeinsames Verständnis im Kontext dieser Arbeit zu erhalten. Anschließend werden die Grundlagen zu Graphen und die Thematik von Datenbanken, sowohl allgemein als auch spezifisch am Beispiel von Neo4j, behandelt. Grundlagen zu Softwarevisualisierungen werden dargelegt und klassifiziert. Außerdem wird auf die Projektarbeit im Praktikum [2] zur Visualisierung von Software-Evolutionsgraphen eingegangen, die die Ausgangssituation dieser Arbeit darstellt. Dabei wird das Ergebnis ausführlich beschrieben, auf den verwendeten Datensatz und dessen Datenbankschema eingegangen und die für die Umsetzung verwendeten Bibliotheken werden ebenso näher erläutert wie die Vorgehensweise der Software.

2.1 Softwarearchitektur

„Software architecture is the conceptual glue that holds every phase of the project together for its many stakeholders.“ [3] - Software Engineering Institute der Carnegie Mellon University.

Es gibt über 50 verschiedene Definitionen für den Begriff der Softwarearchitektur aus diversen Quellen, die dabei unterschiedliche Aspekte hervorheben. Deshalb wird im Folgenden als erstes die Architektur von Software im Kontext dieser Arbeit definiert und im Anschluss auf den bereits zuvor erwähnten Aspekt der Evolution von Softwarearchitektur näher eingegangen und dem damit verbundenen Thema der Versionskontrollsysteme.

2.1.1 Definition und Abgrenzung

Der Begriff der Softwarearchitektur wird teilweise sehr verschieden definiert. Dabei unterscheidet sich dieser nach der Rolle, der Aufgabe oder dem Wissen der jeweiligen Person.

Der ISO/IEC/IEEE 42010 definiert die Architektur von Software folgendermaßen [4]: „The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.“

Eine weitere Definition mit ähnlichem Inhalt von H. Balzert lautet [5]: „Eine Softwarearchitektur beschreibt die Strukturen eines Softwaresystems durch Architekturbausteine und ihre Beziehungen und Interaktionen untereinander sowie ihre physikalische Verteilung.“

Die vorangegangenen Definitionen entsprechen dem Kontext dieser Arbeit; somit umfasst die Architektur alle Komponenten und Module, die sich auf Softwareebene befinden, und die Beziehungen zwischen diesen, um die Struktur und den Aufbau der Software zu beschreiben.

2.1.2 Evolutionsaspekt

Die Evolution einer Software beschreibt die Veränderung im Verlaufe ihrer Entwicklung. Nach den von M.M. Lehmann definierten „Laws of Program Evolution“ [6] verändert sich ein Softwaresystem im Laufe der Entwicklung stetig und damit nimmt zudem die Komplexität zu.

In der Evolution verändert sich auch die Architektur der Software. Zu diesen architekturrelevanten Veränderungen gehören die folgenden zwei:

- Komponenten werden hinzugefügt/entfernt

Damit verändert sich die Anzahl der Komponenten im System. Ein Beispiel dafür ist die Implementierung einer neuen Klasse.

- Beziehungen zwischen Komponenten werden hergestellt/aufgelöst

Diese Veränderung kann beispielsweise durch Hinzufügen der Abhängigkeit einer Klasse von einer anderen entstehen.

Durch diese Entwicklung entspricht die Ist-Architektur häufig nicht mehr der geplanten Soll-Architektur; besonders bei großen und komplexen Softwaresystemen kann der Überblick dadurch schnell verloren gehen. Die Darstellung dieser Veränderung und der Evolution selbst wird damit umso wichtiger, je komplexer die Software wird. Darauf wird im folgenden Kapitel zur Softwarevisualisierung näher eingegangen.

2.1.3 Versionskontrollsysteme

Ein Versionskontrollsystem verwaltet verschiedene Versionen einer Datei im Laufe ihrer Entwicklung und Veränderung. Außerdem unterstützt es die gemeinsame Arbeit an derselben Datei durch mehrere Personen. Jeder Zugriffsberechtigte kann die aktuelle Version lokal bei sich speichern, daran arbeiten und im Anschluss die neue Version einchecken. Dabei stellt ein „Commit“ eine neue, überarbeitete Version dar. Alle Commits werden im System hinterlegt und können jederzeit eingesehen werden. Ein Versionskontrollsystem stellt außerdem alle Veränderungen von der vorherigen Version zur aktuellen dar.

Versionskontrollsysteme finden in vielen Anwendungsbereichen und aus verschiedensten Gründen anklang. So kann beispielsweise Datenverlust verhindert oder Veränderung an Daten rückgängig gemacht werden. Für die Softwareentwicklung sind diese Systeme unverzichtbar, nicht nur für die Arbeit im Team, sondern auch für den einzelnen Entwickler. Ein Beispiel für ein solches Versionskontrollsystem ist *Git*, welches im Falle dieser Arbeit verwendet wurde.

2.2 Softwarevisualisierung

Im Folgenden werden der Begriff der Softwarevisualisierung und deren Ziele näher erläutert. Außerdem folgt ein Überblick des Prozesses von den Rohdaten zur Visualisierung und Klassifizierungen in diesem Kontext.

2.2.1 Definition und Ziele

Softwarevisualisierung beschäftigt sich mit Herangehensweisen, Techniken und Methoden für die graphische Darstellung von Algorithmen, Programmen und Datenverarbeitung [7].

Ziele einer Visualisierung sind nach Schumann und Müller [8] die Analyse, das Verständnis und die Kommunikation von Modellen, Konzepten und Daten. Diese Ziele lassen sich einfach auf die Visualisierung von Software und ihrer Architektur im Speziellen übertragen. Eine geeignete visuelle Darstellung bereitzustellen, mit der sowohl bisher nicht sichtbare Zusammenhänge aufgedeckt werden können als auch eine Kommunikationsgrundlage zum Austausch von Arbeitsergebnissen erfolgen kann, sind die vordergründigen Ziele einer Softwarevisualisierung.

2.2.2 Visualisierungsprozess

Der Prozess, den eine Visualisierung von Rohdaten in der Datenbank bis zur endgültigen Darstellung durchläuft, besteht aus drei wesentlichen Schritten [8]: *Filtering, Mapping* und *Rendering.* Zu Beginn werden die Daten durch Vervollständigung, Reduzierung oder Glättung aufbereitet. Im zweiten Schritt werden die Daten auf geometrische Formen angepasst und Eigenschaften oder visuelle Variablen zugeordnet, damit sie im dritten und letzten Schritt visualisiert werden können.

Adaptiert man den Prozess auf diese Arbeit ergeben sich folgende Schritte:
- Extrahierung architekturrelevanter Daten aus der Graphdatenbank
- Verarbeitung dieser Daten für die korrekte Darstellung
- Visualisierung der Daten als Graphen zu bestimmten Zeitpunkten

Im ersten Schritt liegt der Fokus auf der Beschränkung der Daten, die aus der Graphdatenbank extrahiert werden sollen. Dafür muss klar sein, welche die architekturrelevanten Daten sind bzw. was für die Visualisierung dieser aus der Datenbank benötigt wird. Wurde diese Einschränkung der Daten definiert, kann alles Nötige extrahiert werden. Im Anschluss können die gefilterten Daten dann weiter verarbeitet werden. In diesem Schritt muss klar sein, welche Daten welche Aspekte darstellen sollen. Dabei könnte eine Orientierung an den Klassifizierungen in den folgenden beiden Kapiteln sinnvoll sein. So werden im Beispiel von Graphen die Art der Knoten den jeweiligen Komponenten und weitere Variablen, wie die Farbe oder die Größe der Knoten abhängig von den jeweiligen Eigenschaften übergeben. Nachdem die Zuordnung der extrahierten Daten in ein geometrisches System erfolgt ist, wird die Visualisierung entsprechend erstellt.

2.2.3 Klassifizierung nach Kriterien

Visualisierungen von Softwarearchitektur können auf verschiedene Weisen klassifiziert werden. Ghanam und Carpendale haben in ihrem Survey Paper zu diesem Thema nach drei Kriterien kategorisiert [9]: Ansichten, Dimension und Metapher.

Ansichten

Ob Multi- oder Single-View für die Visualisierung vorteilhafter ist, lässt sich pauschal nicht beantworten.

Bei einem Multi-View Ansatz werden die selben Daten auf verschiedene Arten betrachtet, sodass sich jeweils andere Interessengruppen ansprechen lassen. Dabei werden Aspekte der Architektur in unterschiedlichen Granularitäten in den Vordergrund gestellt. Ein Multi-View Ansatz würde dann beispielsweise in einer Ansicht die Hierarchie der Klassen visualisieren, während eine weitere die Kopplung der Module repräsentiert und eine dritte die evolutionären Veränderungen des Softwaresystems. Mit diesem Ansatz kann sich jede Interessengruppe auf einzelne Aspekte konzentrieren, ohne von Informationen überwältigt zu werden.

Hingegen werden bei dem Single-View Ansatz alle Informationen zur Architektur, die zur Verfügung stehen, in einer einzelnen Ansicht integriert. Das bedeutet, dass jeder Benutzer die gleiche Ansicht auf die Visualisierung erhält, was sich positiv auf die Kommunikation zwischen verschiedenen Interessengruppen auswirken kann, um ein gemeinsames Verständnis der Architektur zu entwickeln. Außerdem kann sich der Benutzer die verschiedenen Aspekte in einer Ansicht erklären lassen, ohne aufgrund von wechselnden Ansichten den Überblick zu verlieren.

Dimension

Darstellungen von Software können in 2D- und 3D-Visualisierungen unterteilt werden.

Das zweidimensionale Umfeld ist im Hinblick auf die Möglichkeiten der Visualisierung eingeschränkt, da die Tiefe im Raum fehlt. Dieser Beschränkung kann entgegengewirkt werden, indem graphische Variablen hinzugezogen werden, wie beispielsweise Größe, Form und Farbe bezogen auf die Attribute der einzelnen Architekturelemente. In diese Kategorie fallen beispielsweise auch die klassischen UML-Diagramme.

Die im 2D-Umfeld fehlende Tiefe, die der dreidimensionale Raum hergibt, erweitert zwar die Möglichkeiten der Darstellung, kann aber auch die Übersicht einschränken. Eine einfache und sinnvolle Navigation durch den Raum ist von Vorteil, um den Überblick über alle Elemente zu behalten. In 3D können klassische Visualisierungen aus dem zweidimensionalen Raum durch eine zusätzliche Höhe übertragen werden, wie beispielsweise Treemaps oder auch Graphen, oder reale Metaphern verwendet werden, welche hauptsächlich in dieser Dimension vorkommen.

Metapher

Bei der Wahl der Art der Darstellung wird zwischen der realen und der abstrakten Metapher unterschieden.

Die abstrakte Metapher verwendet einfache Körper oder Formen, wie beispielsweise Knoten und Kanten, um die jeweiligen Elemente zu visualisieren. Diese Art der Darstellung führt häufiger zu Unverständnis wegen der Komplexität oder des Grades der Abstraktion. Als abstrakte Metaphern gelten alle Visualisierungen ohne realen Bezug.

Die reale Metapher verwendet Szenarien oder Systeme aus dem realen Leben, wie eine Stadt [10], Inseln [11] oder ein Sonnensystem [12]. Durch den Bezug zur Realität soll das Verständnis auch für nicht-technische Rollen sichergestellt werden. Im Beispiel der Inselmetapher [11] stellt eine Insel ein Bundle dar und farbige Bereiche darauf jeweils ein Package, während ein Haus eine Klasse visualisiert. Des Weiteren werden Exporte und Importe der Bundles durch Häfen symbolisiert.

In dieser Arbeit wurde eine abstrakte Metapher im zweidimensionalen Raum umgesetzt. In Bezug auf die Ansichten, werden zwar mehrere mit verschiedenem Fokus aufgezeigt, wie bei einem Multi-View-Ansatz, aber diese Ansichten entsprechen alle der gleichen Art der Visualisierung (Graphen), wie bei einem Single-View-Ansatz.

2.2.4 Klassifizierung nach Aspekt

Visualisierungsplattformen können auch nach den Aspekten Struktur, Evolution und Verhalten kategorisiert werden [1]. Dabei wird bei den meisten Systemen der Fokus der Visualisierung von Software auf einen dieser Aspekte gelegt.

Wird der Fokus auf die Struktur der Software gelegt, können damit beispielsweise die Modularität oder der Aufbau eines Softwaresystems dargestellt werden. Beim Fokus auf der Evolution können dann die Veränderung auf Code-Ebene oder Interaktionen der Entwickler im Laufe des Entwicklungsprozesses dargestellt werden. Liegt der Fokus auf dem Verhalten, soll ein Verständnis für den Vorgang geschaffen werden. So kann das Verhalten einer Software während der Laufzeit visualisiert werden, was das testen erleichtern kann.

Es gibt aber auch einige Tools, die diese Aspekte vereinen, wie beispielsweise *Vizz3D* [13]. Es vereint die Struktur als Entitäten und Relationen, das Verhalten als Zustand im Laufe der Entwicklung und die Evolution durch die Veränderungen der zuvor genannten beiden Aspekte.

In dieser Arbeit wird der Fokus der Software auf die zwei Aspekte Evolution und Struktur gelegt, indem die Historie der Architektur visualisiert wird.

2.3 Graphdankenbanken

In einer Graphdatenbank lässt sich stark vernetzter und komplexer Informationsgehalt abspeichern. Das Besondere an dieser Art der Datenhaltung ist, dass die Beziehungen zwischen den Entitäten nicht erst bei einer Abfrage erstellt werden, sondern bereits abgespeichert sind. Somit sind Abfragen weniger aufwändig und schneller ausführbar, was auch in der folgenden Tabelle 1 zum Performance-Experiment [14] von Neo4j sichtbar wird. Sie eignen sich deshalb auch für besonders große Mengen von Daten.

2.3.1 Graphen

Graphen folgen einer bestimmten Struktur, welche in Abbildung 1 veranschaulicht wird.

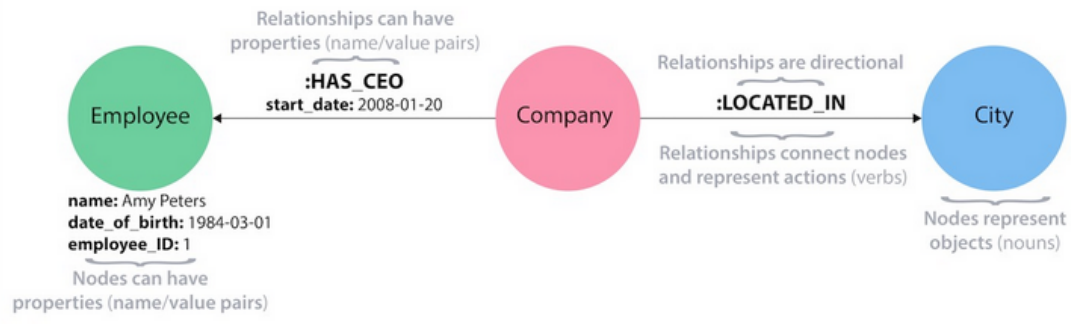


Abbildung 1: Struktur eines Graphen [15]

Graphen bestehen aus Knoten (*Nodes*) und Kanten (*Edges*). Diese wiederum können verschiedene Eigenschaften (*Properties*) besitzen. Ein Knoten repräsentiert ein Objekt und eine Kante eine Beziehung zwischen zwei Knoten. Diese wird auch *Relationship* genannt und hat eine Richtung und somit einen Start- und einen End-Knoten. Außerdem gehört eine Kante immer einem *Relationship Type* an.

2.3.2 Neo4j

Neo4j [16] ist eine NoSQL Graphdatenbank, die als Desktop-Anwendung mit einem Browser zur Verfügung steht oder sich in eigene Projekte diverser Sprachen integrieren lässt. Neo4j verwendet die eigene Abfragesprache *Cypher* [17], welche SQL (*Structured Query Language*) ähnelt, aber spezifisch für Graphen entwickelt wurde. Außerdem ist die Datenbank besonders effizient im Speichern und Präsentieren der Daten und geht sehr flexibel mit Veränderungen um. Die folgende Tabelle 1 zeigt die Ergebnisse eines Performance-Experiments, bei dem die Ausführungszeit von Befehlen bei Neo4j und einem Relationalen Datenbank Management System (RDBMS) gemessen wurden. Bei diesem Experiment wird die Effizienz von Neo4j deutlich.

Tabelle 1: Performance von Neo4j gegenüber relationalen Datenbanken [14]

Depth	RDBMS execution time(s)	Neo4j execution time(s)	Records returned
2	0.016	0.01	~2,500
3	30.267	0.168	~110,000
4	1543.505	1.359	~600,000
5	Unfinished	2.132	~800,000

2.4 Praktikumsarbeit - Ausgangssituation

Im Praktikum [2] ist eine Webanwendung zur Visualisierung von Software-Evolutionsgraphen entstanden. Diese wurde statisch, an einen Datensatz gebunden, der auf OSGi basiert, entwickelt. Im Folgenden werden das Ergebnis und die verwendeten Bibliotheken für die Umsetzung sowie die Vorgehensweise der Software näher erläutert. Das Ergebnis der Praktikumsarbeit stellt die Ausgangssituation für die Entwicklung des Konzepts und den anschließenden Prototyp im Rahmen dieser Arbeit dar.

2.4.1 Ergebnis

Die entstandene Webanwendung zur Visualisierung der Evolution von Softwarearchitektur extrahiert ihre Daten aus der Datenbank Neo4j und stellt diese in Form von Graphen dar. Abbildung 2 zeigt einen Screenshot der Visualisierungsplattform, die den Ausgangspunkt für die weitere Arbeit darstellt.

Auf `Bundle`-, `Package`-, `Class`- und `Service`-Ebene lässt sich die Architektur zu mehreren Zeitpunkten ihrer Evolution darstellen. Bei den Zeitpunkten handelt es sich um die Commits aus dem Git Repository des dargestellten Projekts. Eine zusätzliche benutzerdefinierte Ansicht zeigt nur bestimmte Knoten, die der Nutzer vorher selektiert hat. Durch Klicken der `Buttons` lässt sich zwischen den Ansichten navigieren und über den Zeitstrahl lässt sich jeder beliebige Commit auswählen. Eine Vor- und Zurück-Funktion erleichtert das Wechseln zwischen den zeitlich aufeinander folgenden Commits. Ein zusätzlicher Zeitraffer läuft die Commits automatisch in zeitlicher Reihenfolge durch; er lässt sich jederzeit starten und pausieren. Eine Suchleiste mit automatischer Vervollständigung wurde integriert, um direkt nach Knoten suchen und diese selektieren zu können. Beim Selektieren von Knoten oder Kanten werden zusätzliche Informationen an der rechten Seite des Bildes ausgegeben. Beim Berühren eines Elements in der Visualisierung oder dem Zeitstrahl wird ein `Tooltip` mit dem jeweiligen Namen der Node bzw. Relationship Type oder das Datum des Commits angezeigt.

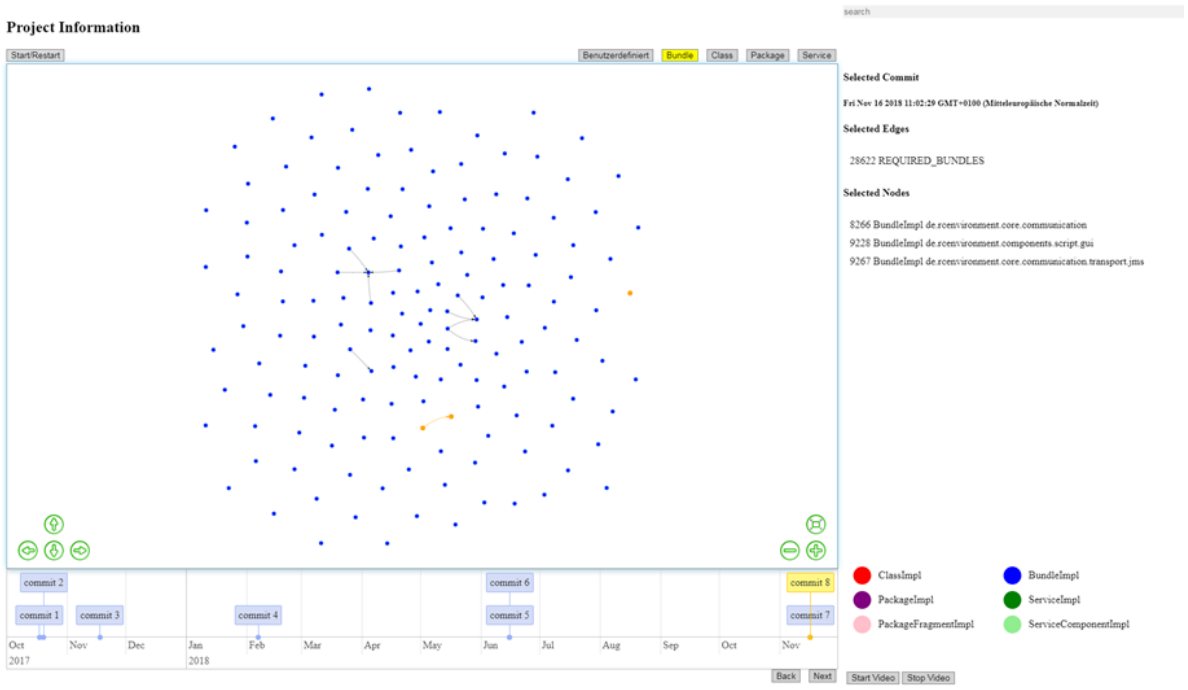


Abbildung 2: Screenshot der im Praktikum entstandenen Webseite

Für die Erstellung der Webseite wurden HTML (*Hypertext Markup Language*) und CSS (*Cascading Style Sheets*) verwendet und *Javascript* für die funktionellen Aspekte und Logik dahinter. Für die Extrahierung der Daten wurde der *Neo4j Javascript Driver* verwendet. Der Zeitstrahl und die Graphvisualisierung wurden mit der Javascript-Bibliothek *Vis.js* umgesetzt. Die genannten Sprachen und Bibliotheken werden weitgehend übernommen für die Umsetzung dieser Arbeit.

2.4.2 Datensatz

Der verwendete Datensatz, um dieses Projekt umsetzen und testen zu können, wurde in Java basierend auf OSGi entwickelt und stammt aus den Commits eines Git-Projekts. Die Daten waren bereits aus *GitLab* exportiert und in Neo4j integriert; bestehend aus acht Commits mit insgesamt 60.945 Knoten und 115.512 Kanten. Es existieren 13 Arten von Knoten (*Node Label*), welche aber nicht alle für die Architektur relevant sind, und 19 Kantentypen.

Java gehört zur Familie der objektorientierten und klassenbasierten Programmiersprachen. OSGi dient zur Erweiterung der Modularisierung von Code, die bei Java für besonders große Projekte zu schwach ist. Es handelt sich um eine Spezifikation von Java, um eine komponentenorientierte Entwicklung zu unterstützen und Services abrufen und bereitstellen zu können. Das Framework unterteilt die Java-Applikationen in Module und verwaltet diese und deren Abhängigkeiten über den gesamten Lebenszyklus.

2.4.3 Neo4j Javascript Treiber

Für die Extrahierung der Daten aus Neo4j wurde der speziell dafür entwickelte Neo4j JavaScript Driver [18] verwendet. Mithilfe des Treibers kann Neo4j über JavaScript integriert und angesprochen werden. Das von Neo4j unterstütze binäre Protokoll nennt sich *Bolt-protocol* und unterstützt das *Cypher type system*.

Nachdem der Treiber mit Übergabe der URL (*Uniform Resource Locater*), einem Benutzernamen und dem entsprechenden Passwort instanziert wurde, wird eine Sitzung erstellt, um darin Cypher-Befehle ausführen zu können. Die Abfragen werden direkt an Neo4j übermittelt, wo sie ausgeführt werden, um passende Daten an das Programm zurück zu liefern. Im Anschluss können die aus der Cypher-Abfrage resultierenden Ergebnisse ausgewertet und weiter verarbeitet werden. Wird die Sitzung nicht mehr benötigt wird diese, genau wie der Treiber nach erfolgreicher Nutzung geschlossen.

2.4.4 Visualisierungsbibliothek Vis.js

Die dynamische, browserbasierte Visualisierungsbibliothek Vis [19] bietet verschiedene Komponenten zur Darstellung von Daten an, wie `Network` und `Timeline`, welche für die Umsetzung genutzt wurden.

Durch beinahe lückenlose Dokumentation und Beispiele lassen sich schnell und einfach Visualisierungen erstellen. Dabei wird auf Übersichtlichkeit und Eindeutigkeit Wert gelegt. Viele Funktionen beim Erstellen einer Visualisierung sind bereits integriert, was die Verwendung und Bedienung besonders für Anfänger deutlich erleichtert. Die Bibliothek wurde so entwickelt, dass sie einfach zu benutzen ist, große Mengen an dynamischen Daten verarbeiten kann, und um die Daten manipulieren und mit ihnen interagieren zu können.

Der festgelegte Layout-Algorithmus beim Graphen ist für einfache Visualisierungen vorteilhaft, da dieser die Knoten und Kanten solange verschiebt, bis das optimale Layout gefunden wurde für beste Übersichtlichkeit. Allerdings kann sich diese Eigenschaft auch negativ auswirken, wie im Falle des Evolutionsaspekts. Je mehr Nodes und Edges der Graph hat, desto länger rendert er, was für den Nutzer sichtbar ist und somit bei einem Wechsel zwischen Commits die Übersichtlichkeit der direkten Veränderung zunichte macht. Der Zeitstrahl benötigt lediglich das richtige Format an Daten, sowie einen Start- und End-Zeitpunkt für die korrekte Darstellung.

2.4.5 Vorgehensweise

Beim Start der Anwendung werden die Commits automatisch aus der Datenbank extrahiert, wie sich dem Programmablaufplan aus Abbildung 3 entnehmen lässt. Aus der Abbildung lassen sich auch sofort statische Funktionen erfassen.

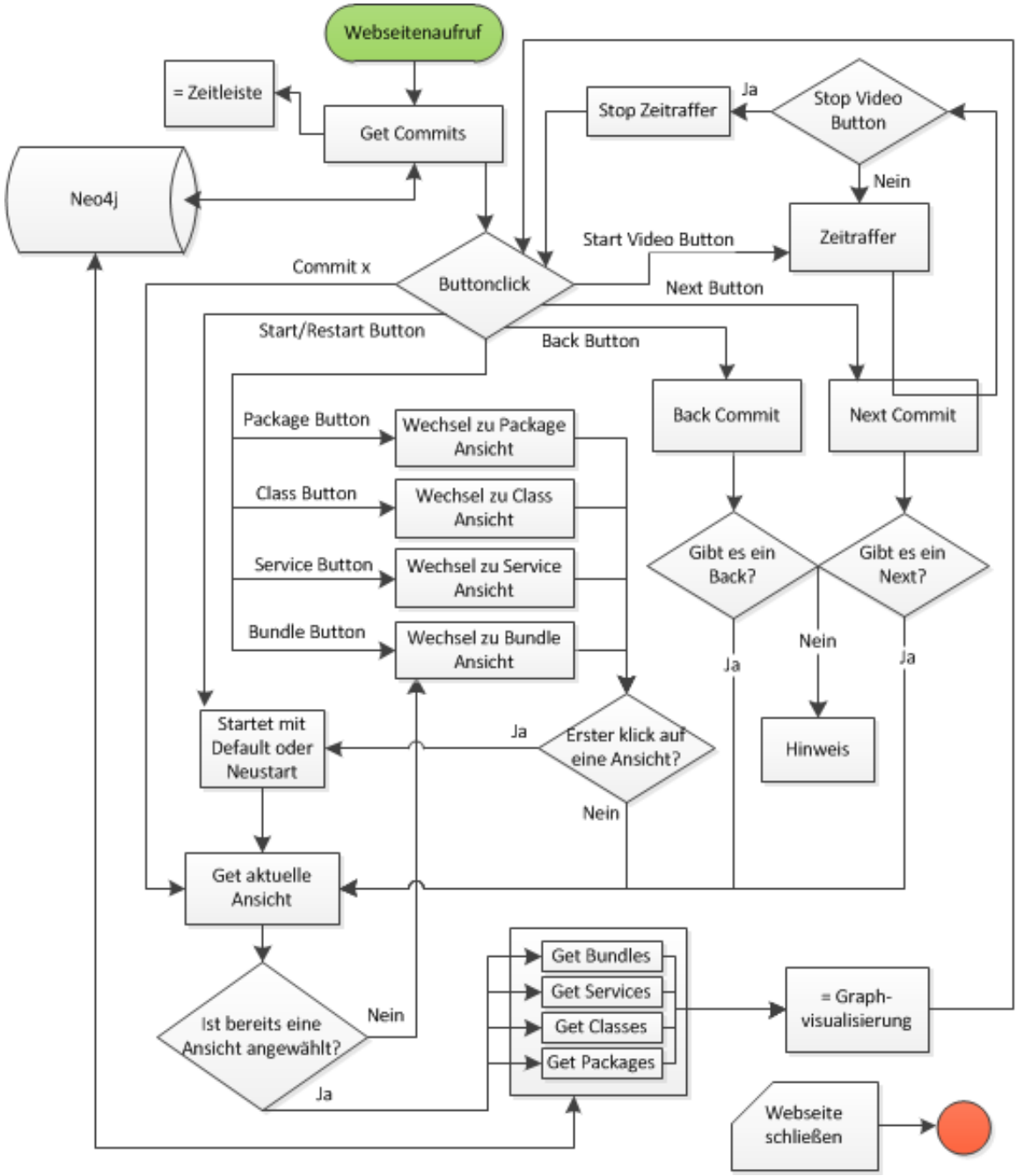


Abbildung 3: Programmablaufplan des statischen Systems

Das Extrahieren der Daten für die Commits erfolgt statisch mit dem Befehl:
`MATCH (n:CommitImpl) RETURN n ORDER BY n.time`.
Bei der Abfrage wird explizit nach dem Node Label `CommitImpl` gefragt, aufgrund des Wissens, dass die Commits in diesen Nodes enthalten sind.

Der in den Commits hinterlegte *Unix*-Zeitstempel mal 1000 wird als `time` angegeben, um das Resultat in zeitlicher Reihenfolge verarbeiten zu können. Diese Zeit entspricht den Sekunden, die seit dem `01.01.1970 00:00:00 Uhr` vergangen sind. Eine Konvertierung der Unix-Zeit erfolgt für die Zeitangabe der Commits auf der Zeitleiste, um ein von Vis interpretierbares Format zu übergeben.

Nach der Extrahierung der Commits kann der Nutzer einen beliebigen der Buttons anwählen, da jede mögliche Reihenfolge der Ereignisse abgedeckt wurde. Sollte der Nutzer einen der Commits auswählen, wird geprüft ob eine Ansicht zuvor gewählt wurde, ist dies nicht der Fall, wird die der Bundles ausgewählt. Wurde allerdings eine Ansicht zuvor gewählt, wird der entsprechende Graph dieser aus Neo4j extrahiert und visualisiert.

Wählt der Nutzer einen der Ansicht-Buttons wird die entsprechende Ansicht gewählt oder auf diese gewechselt. Es folgt eine Abfrage, ob es das erste anwählen einer Ansicht war. Ist das der Fall, wird der letzte Commit als `Default` übergeben und dann die Ansicht erneut abgefragt, um die korrekte Funktion für die Extrahierung und Visualisierung des Graphen aufzurufen. War das nicht das erste Anwählen einer der Ansichten, wird nur die aktuelle Ansicht nochmals abgefragt, um die korrekte Funktion zu erfassen, da dann bereits eine Zeitangabe gewählt wurde.

Ähnlich den Commits zuvor werden die Knoten und Kanten der einzelnen Ansichten abgefragt und verarbeitet. Dafür werden beispielsweise für die Bundle-Ansicht die Knoten mit dem Node Label `BundleImpl` und in einem separaten Befehl die Kanten mit dem Relationship Type `REQUIRED_BUNDLES` abgefragt. Dieses statische Vorgehen wurde entsprechend des Vorwissens über den Datensatz in allen Ansichten angewendet. Der Grund für eine Separation von Abfragen der Knoten und Kanten, ist das Verhindern des Entstehens eines Kreuzprodukts der Elemente.

In die Abfrage wird statisch der Zeitstempel als Parameter übergeben, um den korrekten Commit darstellen zu können. Bei der Verarbeitung der Daten werden auch die Property Keys direkt mit ihrer Bezeichnung abgefragt, wie beispielsweise in der Commit-Abfrage der Zeitstempel mit `time`.

Mit den `Back`- und `Next`-Button kann solange zwischen den aufeinander folgenden Commits navigiert werden, solange es Commits gibt. Vor dem ersten bzw. nach dem letzten wird ein entsprechender Hinweis ausgegeben, dass das Ende erreicht wurde. Der Zeitraffer läuft ab dem Anwählen des entsprechenden `Start`-Buttons die Commits durch. Dabei ruft er die Funktion hinter `Next Commit` auf, bis zur Unterbrechung durch den `Stop`-Button oder das Erreichen des letzten Commits.

Durch das Wissen über Modell und Struktur des Datensatzes wurde die Software entsprechend aufgebaut. Node Label, Relationship Types und Property Keys sind statisch an den Datensatz gebunden integriert worden. Sowohl bei der Extrahierung jeglicher Daten aus Neo4j und der Verarbeitung dieser, als auch bei der anschließenden Visualisierung wurde der Aspekt der Erweiterung und Integration von weiteren Datensätzen außer Acht gelassen.

3 Konzept

In der folgenden Konzeptionierung einer Generalisierung werden zuerst die Zielgruppe und deren Interessen definiert. Im Anschluss wird auf die statischen Aspekte und mögliche Herausforderungen bei der Implementierung eingegangen. Ziel dieses Kapitels ist es, Umsetzungsmöglichkeiten für eine Generalisierung der Software zu vergleichen, um die geeignetste unter ihnen für die im nachfolgenden Kapitel beschriebene prototypische Umsetzung zu finden.

3.1 Zielgruppe

Da Art und Aufbau der Softwarevisualisierung bereits feststehen, kann nicht von der Zielgruppe ausgegangen werden, sie kann aber bei der Weiterentwicklung mitberücksichtigt werden. Um die Frage zu beantworten, für wen diese Art der Visualisierung von Softwarearchitektur geeignet ist oder interessant sein kann, werden Rollen und Interessengruppen gesammelt und anschließend anhand verschiedener Faktoren weiter eingeschränkt.

Laut einer überschaubaren Befragung [20] im Rahmen der Test-Tage 2009, die hauptsächlich von Programmierern, Testern und Architekten, aber auch von ein paar Beratern, Managern und Projektleitern beantwortet wurde, kommt es bei der Wahl der Visualisierungsart auf die Nähe zum Code und den Kommunikationspartner in der Situation an. Es wurde nach den am Arbeitsplatz eingesetzten Softwarearchitektur-Visualisierungen gefragt und in welchen Fällen diese eingesetzt werden. Etwa zwei Drittel der Befragten verwendeten in Projekten Visualisierungen für die Kommunikation und Diskussion von Softwarearchitektur. Je weiter die Tätigkeit der Befragten von der Entwicklung entfernt stattfand, desto eher wurden freie Architektur-Visualisierungen verwendet. Teilnehmer aus technischen Rollen arbeiteten hauptsächlich mit den klassischen UML-Diagrammen, griffen aber zur Kommunikation mit nicht-technischen Rollen häufiger zusätzlich auf freie Visualisierungen zurück. Somit wären sowohl technische als auch nicht-technische Rollen Teil der Zielgruppe.

Um die Zielgruppe genauer definieren zu können sollten die einzelnen Interessen der Personengruppen, mit einbezogen werden. In einer Zusammenfassung von Paper und Arbeiten zu Softwarearchitektur-Visualisierungen haben Ghanam und Carpendale [9] Personengruppen und deren Interesse in die Detailliertheit von Visualisierungen der Softwarearchitektur zusammengebracht. Folgende Graphik in Abbildung 4 hat sich daraus ergeben, an der die deutlichen Interessenunterschiede erkennbar werden.

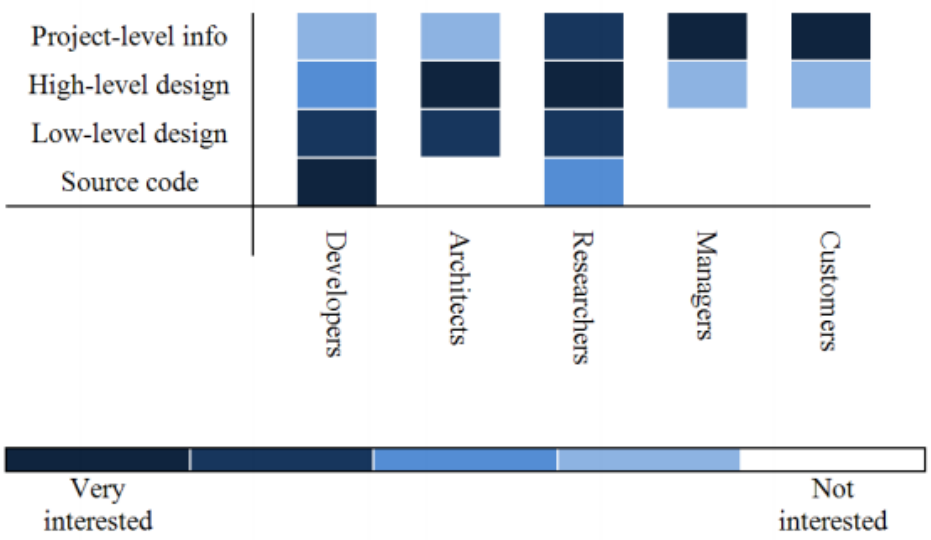


Abbildung 4: Interessengruppen im Bereich Softwarevisualisierung [9]

Für den Kunden, der lediglich einen generellen Überblick über die Softwarearchitektur gewinnen möchte, ist die Visualisierung in Form von Graphen schnell zu abstrakt. Auch Managern könnte diese Art der Visualisierung der Architektur zu abstrakt sein; allerdings könnten diese dadurch einen Überblick über den Prozess und den Erfüllungsgrad der gesetzten Ziele und des Projekts erhalten. Da diese Arbeit allerdings keine Analysefunktionen oder Fortschrittsuntersuchungen unterstützt, in der man die Ist- mit der Ziel-Architektur visuell vergleichen könnte, fällt die Rolle des Managers nicht in die Zielgruppe dieser Arbeit.

Der Architekt interessiert sich vor allem für die Charakteristiken der Softwarearchitektur, wie die Komplexität, Kohäsion und Kopplung der einzelnen Elemente im System, welche in dieser Arbeit nicht weiter spezifiziert und behandelt werden. Da Architekten aber die Ansprechpartner und Beauftragten rund um die Architektur einer Software sind, fallen sie in diese Zielgruppe.

Es bleiben die Interessengruppen Entwickler (inkl. Maintainer) und Forscher, welche aufgrund ihrer Arbeit, ihres Wissens und den Interessen, der Zielgruppe dieser Arbeit entsprechen. Forscher sind vor allem an den Trends und der Evolution der Architektur interessiert. Die Hauptzielgruppe stellen die Entwickler und Maintainer einer Software dar, denn diese sind vorrangig am Aufbau der Architektur und dem aktuellen Status des Systems interessiert. Aber auch für nicht-technische Rollen, die etwas Wissen im Bereich Softwarearchitektur besitzen, kann diese Art der Visualisierung von Interesse sein.

3.2 Statische Aspekte

Der Fokus der Generalisierung der Visualisierungsplattform liegt auf den statischen Aspekten, die sich durch die drei Schritte des Visualisierungsprozesses ziehen. Dabei geht es vor allem um das Extrahieren der Daten ohne zu wissen, wie diese aufgebaut sind, welchem Schema sie folgen oder was für untergeordnete Daten enthalten sind.

Tabelle 2: Sammlung der statischen Aspekte

#	Statischer Aspekt	Kategorie
A01	Node Label	Bestimmt durch Abfrage
A02	Relationship Type	
A03	Property Key	
A04	Format der Daten	
A05	Beschriftung der Ansicht-Buttons	Architekturabhängig
A06	Anzahl der Ansichten	
A07	Anzahl der Abfragen	
A08	Navigation zwischen den Ansichten	Codegebunden
A09	Verbindungsaufbau mit Neo4j	
A10	Node Legende	
A11	Farbgebung der Nodes	
A12	Funktionsnamen	
A13	Anzahl der Funktionen	

Die auf Anhieb optisch sichtbaren Aspekte, die generalisiert oder automatisiert werden sollen, sind die auf der Webseite angezeigten (vgl. Abbildung 2). Die Beschriftung der Buttons für die Ansichten sowie die Anzahl dieser sind statisch im Programmcode verankert. Diese müsste man durch variable Zahlen mit variablen Namen automatisch entsprechend der vorhandenen Ansichten generieren. Die Legende ist ebenfalls statisch in HTML implementiert worden, entsprechend der statischen Farbzuordnung der Node Labels. Somit ist auch in den Ansichten die Farbauswahl der Labels nicht automatisiert erfolgt.

Alle Property Keys müssen durch Variablen ersetzt werden, da diese genauso unbekannt sind wie die Node Label oder Relationship Types. Des Weiteren ist das jeweilige Format der Eigenschaft bekannt und wird entsprechend statisch weiter verarbeitet, beispielsweise bei der Konvertierung eines Zeitformats in ein anderes.

Für jede der vier statischen Ansichten wurde eine separate Funktion zum Abfragen, Verarbeiten und Visualisieren der Knoten und Kanten des Graphen implementiert (vgl. Abbildung 3), die entsprechend benannt wurden. Zudem ist die Anzahl der Cypher-Abfragen gebunden an die Anzahl an Node Labeln und Relationship Types, die in der entsprechenden Ansicht enthalten sind, da jede einzeln abgefragt wird. Auch der Wechsel zwischen diesen Ansichten erfolgt statisch, durch eine Abfrage der Farbe jedes Buttons, denn die aktuell angewählte Ansicht wird farblich hervorgehoben.

Die Verbindung zur Datenbank erfolgt ebenfalls statisch durch die Initialisierung des Treibers im Programmcode, mithilfe von definierten Daten. Daraus sollte ein Datenbank-Login entstehen, indem der Nutzer seine individuellen Zugangsdaten eingeben würde.

Zusammenfassend lassen sich die statischen Aspekte in drei unterschiedliche Kategorien untergliedern, welche in Tabelle 2 gesammelt und durchnummeriert wurden. Die Aspekte sind somit entweder abhängig von der betrachteten Softwarearchitektur, bestimmt durch die Cypher-Abfrage oder gebunden an den implementierten Programmcode. An diesen Stellen muss nun die Generalisierung angesetzt und im Anschluss eventuell in die anderen Bereiche erweitert werden.

3.3 Herausforderungen

Wie bereits häufiger erwähnt ist das Datenbankschema nicht nur von der Programmiersprache der Software abhängig, sondern auch von der Integration der Architekturdaten in die Graphdatenbank. Im Falle von Neo4j ist die einfachste Möglichkeit die Daten in einer *comma-separated values* (CSV)-Datei, die beispielsweise aus dem Git eines Projekts stammen kann, abzuspeichern und diese in die Graphdatenbank einzubinden. Darin sind dann zwar alle relevanten Daten des Projekts enthalten, diese müssen aber beim Integrieren noch benannt und strukturiert werden.

Abbildung 5 zeigt im unteren Abschnitt die CSV-Datei, die aus den Commits eines eigenen Projekts aus GitLab gewonnen wurde. Aus dem oberen Bildabschnitt lässt sich der Befehl zum Importieren und Verarbeiten der Datei entnehmen. Dabei wird jedes Element einer Property zugeordnet, wie Name, Autor und Zeitpunkt des jeweiligen Commits, der durch eine Zeile repräsentiert wird. So werden in diesem Fall Nodes mit dem Label „Commit“ erstellt. Im Anschluss daran müsste man die Relationships definieren, indem man diese durch Node Label bezogenene Befehle individuell generiert. Das Integrieren des Datensatzes ist somit eindeutig an die Person gebunden, die diesen Vorgang unternimmt.

Zu den Herausforderungen bei der Implementierung gehören damit das korrekte Auslesen der Daten bei der Extrahierung aus der Datenbank und das Extrahieren selbst. Herauszufinden, welche Programmiersprache vorliegt, um die Architektur dieser in entsprechende Ansichten zu untergliedern und die zugehörigen Daten anhand ihrer Node Labels oder Relationship Types einzubinden, stellt die größte Herausforderung dar. Dicht gefolgt vom Auslesen der Labels, Types und Keys, um diese korrekt interpretieren zu können, damit eine vollständige Visualisierung und die davon abhängigen Funktionen der Plattform gewährleistet werden.

Die Herausforderung ist, mit dem selben Datensatz nicht weniger Funktionen und Ergebnisse in der Generalisierung zu erzielen, als bei der bestehenden Plattform.

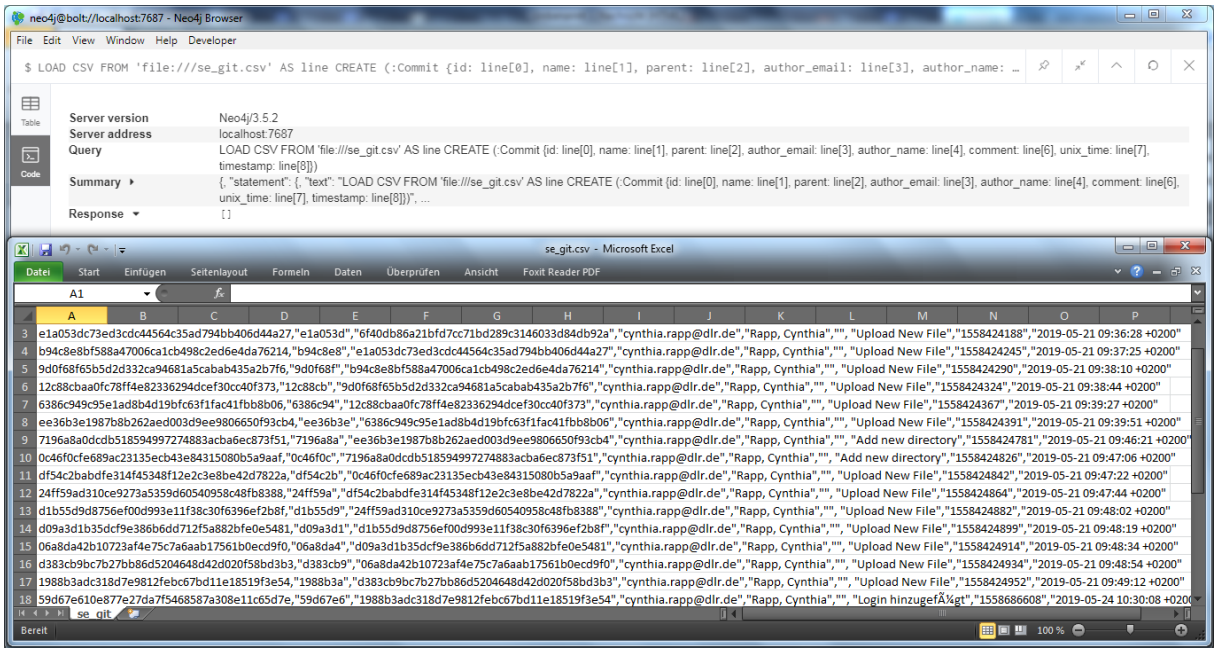


Abbildung 5: Import-Befehl und CSV-Datei

3.4 Generalisierungsansätze

Es gibt viele verschiedene Ansätze, um Software zu generalisieren. Dabei wird in dieser Arbeit der Vergleich auf folgende drei mögliche Methoden beschränkt.

Ansatz A Hierbei wird eine Art Konfigurationsdatei bereitgestellt, für die der Nutzer zu Beginn einige Angaben zur Zuordnung der Datenbank mit der Visualisierung machen muss. Für die Ansichten werden Informationen über das Datenbankschema und die verwendeten Variablen benötigt. Dabei muss klar sein, welche Komponenten es gibt, die in den einzelnen Ansichten zu sehen sein sollen, welches Node Label für die Commits steht und welche Property Keys welchen Eigenschaften entsprechen.

Es könnte eine Art Formular vorgeben werden, mit dem der Benutzer aufgefordert wird, diese Angaben zum Datensatz zu machen. Die Benutzereingaben würden ausgelesen und verarbeitet und im Anschluss entsprechend die Ansichten generiert und Graphen erzeugt werden.

Die Eindeutigkeit der Zuordnung ist hierbei der größte Vorteil, da alles vordefiniert wird und nur noch ausgelesen und erzeugt werden muss. Man könnte mit dieser Methode einige Programmiersprachen mit diversen Datenbankstrukturen einlesen und visualisieren.

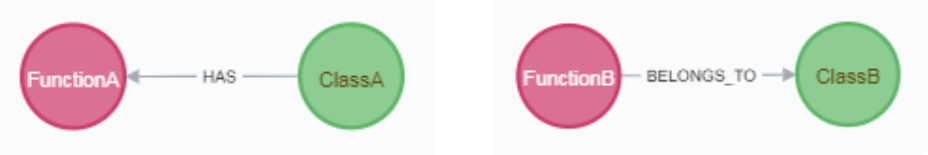
Allerdings ist das für den Nutzer weniger sinnvoll, da er die gesamte Struktur des Graphen und die Eigenschaften kennen muss, um die Eingaben korrekt machen zu können. Die Plattform erst sollte dem Nutzer die Struktur näher bringen und vor allem benutzerfreundlich und unterstützend sein, was durch die spezifischen Eingaben eher aufwändig und kompliziert erscheinen würde.

Ansatz B Gäbe man dem Nutzer eine Auswahl an Programmiersprachen, könnte dieser die entsprechende unter ihnen auswählen und es würde automatisch nach vordefinierten Regeln visualisiert werden. Dieser Ansatz entspricht Ansatz A, allerdings mit vorgegebener Architektur anhand der ausgewählten Sprache in der Konfigurationsdatei. Somit ist der Aufwand für den Benutzer deutlich geringer als im vorherigen Ansatz.

Das größte Problem dieser Methode ist die Beschränkung auf ein paar Programmiersprachen, da es im Rahmen dieser Arbeit unmöglich sein wird, eine Konfigurationsdatei für jede existierende Sprache zu erstellen. Man könnte sich hier beispielsweise auf objektorientierte Sprachen beschränken oder auf die zehn Häufigsten in der Praxis. Allerdings müsste man dann zusätzlich definieren, ob Frameworks wie das OSGi, welches die Struktur eines Java-Programms verändert, hinzugezogen werden oder nicht. Hinzu kommt die Tatsache, dass die Graph-Variablen weiterhin nicht eindeutig sind und von der Person abhängen, die programmiert oder das Einlesen des Graphen in die Datenbank übernommen hat. Wenn von der englischen Sprache ausgegangen wird, welche beim Programmieren meist Standard ist, könnte man die Begriffe der Komponenten, die jeweils die Architektur der ausgewählten Programmiersprache ausmachen, in den Node Labeln suchen. Das garantiert aber noch kein funktionierendes System, da die Daten bei der Integration individuell benannt werden können und beispielsweise Abkürzungen verwendet worden sein können.

Ansatz C Folgt das Datenbankschema einer Hierarchie, was meist der Fall ist, könnte ein automatisches Auslesen der Nodes implementiert werden. Dabei wird von der obersten Komponente ausgegangen und dann die Beziehungen Ebene für Ebene durchschritten, um darauf basierend die Komponenten zu realisieren.

Dieser Ansatz wäre der dynamischste unter den genannten. Allerdings müssten die Programmiersprachen auch hier beschränkt werden. Es größeres Problem dieser Methode ist aber, dass in den Nodes die Hierarchieebene vermerkt sein müsste, da ein Anhaltspunkt für den Beginn und das Ende benötigt werden würde. Das ist von der Integration der Daten in die Datenbank abhängig. Während eine Person die Beziehung zwischen einer Klasse und einer Funktion der Hierarchie entsprechend definiert (Abbildung 6a), kann die andere Person einem anderen Gedankengang gefolgt sein und somit nach Zugehörigkeit definiert haben (Abbildung 6b).



(a) Klasse A enthält Funktion A (b) Funktion B gehört zu Klasse B

Abbildung 6: Gegenüberstellung der Richtung einer Beziehung zwischen zwei Knoten

Somit ist die Aussage der beiden Beziehungen die gleiche, aber die Software kann das nicht gleich verarbeiten. Wenn das Auslesen also gelingen könnte, würde es trotzdem unterschiedliche Ergebnisse zur Folge haben und die Richtigkeit der Darstellung der Architektur kann nicht länger gewährleistet werden.

3.5 Implementierungsansatz

Vergleicht man die gegebenen Ansätze auf ihre Machbarkeit, wird deutlich, dass keine davon allein eine geeignete Lösung darstellt. Es ist zum heutigen Stand der Technik kaum möglich, eine vollständige Automatisierung des Visualisierungsprozess umzusetzen, da nach wie vor kein Standard in Bezug auf die Strukturierung der Knoten und Kanten in der Datenbank gegeben ist.

Nachdem die Optionen und die Vor- und Nachteile der einzelnen Ansätze betrachtet wurden, scheint die vielversprechendste Schlussfolgerung eine Kombination dieser umzusetzen. Dabei werden die Möglichkeiten zur Auswahl und Eingaben des Benutzers und die anschließend darauf basierende automatische Generierung der Ansichten umgesetzt.

Die Idee dahinter ist es, den Benutzer erst durch die Auswahl einer Programmiersprache die weitere Konfigurationen einzugrenzen und anschließend darauf basierend zuordnen zu lassen, welche Arten von Knoten in welcher Ansicht enthalten sind. Dabei werden durch die vordefinierte Struktur der jeweiligen Sprache die Architekturkomponenten vorgegeben und der Benutzer kann die entsprechenden Node Label dieser zuordnen. Das entsprechende Label für die Nodes, in denen die Commits hinterlegt sind, muss ebenfalls von Nutzer zugeordnet werden. Im Anschluss kann die Software die gesammelten Eingaben verarbeiten. Entsprechend der Angaben werden die Ansichten automatisch generiert und weitere Daten aus der Datenbank für die Visualisierung der einzelnen Graphen der Commits erhoben.

Zu den weiteren Daten, die für die korrekte Übergabe und Darstellung der Graphen benötigt werden, gehören die Property Keys. An dieser Stelle wird die Annahme getroffen, dass der Datenbankintegrateur jegliche Bezeichnungen in englischer Sprache durchgeführt hat. Die weiteren Angaben zu den Eigenschaften werden dann mithilfe von sinnvollen Übereinstimmungen von Begrifflichkeiten aus diesem Kontext getroffen. Des Weiteren wird davon ausgegangen, dass jeder relevante Knoten die Zeitangabe des jeweiligen Commits als Property Key enthält, sowie einen eindeutigen Namen.

Weitere Funktionen wie ein Datenbank-Login werden hinzukommen, ebenso ein Farbhandler, der das korrekte Zuordnen von Farben zu Node Labeln unabhängig von der Anzahl dieser übernimmt. Dieser wird auch für die passende Zuordnung der Farben in der Legende sorgen. Des Weiteren soll das System so gestaltet werden, dass man es um weitere Sprachen und deren Architekturkomponenten erweitern kann. Dafür sollen keine weiteren Veränderungen nötig sein und trotzdem alle Funktionen weiter gewährleistet werden.

4 Umsetzung

Das im Kapitel zuvor entwickelte Konzept wird im Folgenden prototypisch umgesetzt. Dabei wurde die Ausgangssituation für die weitere Entwicklung in Kapitel 2.4 und die zu generalisierenden statischen Aspekte, die den Fokus dieser Umsetzung darstellen, in Kapitel 3.2 beschrieben. Die Umsetzung lässt sich an dieser Stelle in zwei Teile untergliedern: Die Konfiguration und den anschließenden Visualisierungsprozess. Dabei wird in den Unterpunkten jeweils auf die Nutzung des Systems und auf die technischen Hintergründe eingegangen. Anschließend folgt ein Überblick über die Vorgehensweise des Systems als Beantwortung der zentralen Fragestellung. Um das Folgende abzurunden werden zum Ende zusätzlich einige Erweiterungs- und Optimierungsmöglichkeiten genannt.

4.1 Konfiguration

Zur Konfiguration gehören alle Schritte, die vom Benutzer durchgeführt werden müssen; die Eingabe der Login-Daten für den Zugriff auf die Datenbank, die Auswahl der Programmiersprache der zu betrachtenden Software und die Zuordnung der Node Labels zu den Architekturkomponenten in Form der Ansichten. Diese Konfigurationen bestehen teilweise aus vordefinierten Einstellungen, die durch die Auswahl des Nutzers übernommen werden, und aus solchen, die rein vom Nutzer eingestellt werden. Um die Konfigurationen möglichst benutzerfreundlich und effizient zu gestalten, wurden separate Formulare für die einzelnen, aufeinander aufbauenden Funktionen entwickelt, die im Folgenden näher beschrieben werden. Im Hintergrund der Abbildung 7 ist die noch unvollständige Webseite zu erkennen, die mit der Bestätigung der Eingaben in den Formularen weiter ergänzt wird.

4.1.1 Datenbank-Login

Wie bereits in Kapitel 2.4.3 zum Neo4j Javascript Treiber erwähnt, fand der Verbindungsaufbau zu Neo4j über die Treiberinitialisierung durch statisch im Code hinterlegte Daten statt. Damit dieser Schritt dynamisch mit unterschiedlichen anderen Zugangsdaten umgesetzt werden konnte, wurde ein Formular mit Eingabefeldern implementiert. Es werden die Datenbank-URL, der Benutzername und das Passwort zum Zugriff auf die aktive Datenbank gefordert. Das Formular überlagert die Webseite beim Starten, wie in Abbildung 7 erkennbar ist. Hier wurden zur Veranschaulichung bereits Zugangsdaten im verlangten Format eingetragen.

Mit dem Bestätigen der Eingaben über den `Connect`-Button wird das Formular geschlossen und ein neues zur Auswahl der Programmiersprache öffnet sich an der Stelle, was im folgenden Kapitel 4.1.2 näher beschrieben wird.

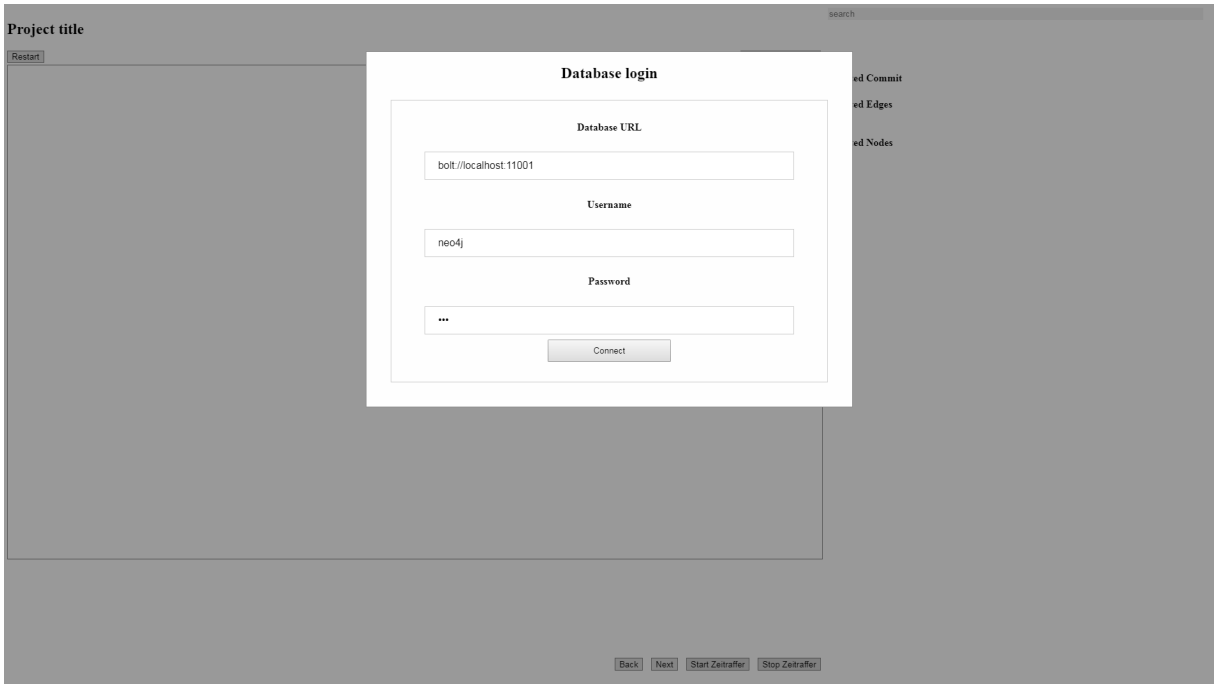


Abbildung 7: Login-Formular mit beispielhaften Zugangsdaten

Sofern die Eingaben des Benutzers korrekt sind und dieser zugriffsberechtigt ist, wird der Treiber anhand der Daten initialisiert, sodass jederzeit eine Sitzung gestartet werden kann. Des Weiteren wird das gesamte Schema des Datensatzes aus der Datenbank extrahiert und bereits im Hintergrund visualisiert, wie sich in Abbildung 8 erkennen lässt.
Mit diesem Login-Formular ist der statische Aspekt A09 zur Verbindung mit Neo4j aus Tabelle 2 dynamisch umgesetzt und die Bindung an den Code aufgehoben worden.

4.1.2 Auswahl der Programmiersprache

Um die korrekten Ansichten, die auf den Architekturkomponenten basieren, anzeigen zu können, wird die Programmiersprache der Software benötigt, denn diese bestimmt die Architektur. Der Benutzer soll in einem Formular aus einer Liste von im System hinterlegten Sprachen eine auswählen können. Abbildung 8 zeigt dieses Formular, in dem die bisher eingebundene Programmiersprache und ein beispielhafter zweiter Eintrag mit der Beschriftung „Other“ enthalten sind. An dieser Stelle werden die Programmiersprachen aufgelistet, die in einer erweiterbaren XML-Datei (*Extensible Markup Language*) hinterlegt sind. Die Erweiterbarkeit dieser Datei um weitere Programmiersprachen erlaubt es theoretisch jede Sprache integrieren zu können. In Abbildung 9 lässt sich der Aufbau der Datei `languages.xml` deutlich erkennen. XML arbeitet wie HTML mit *Tags*, die beliebig angeordnet werden können. Jede Sprache erhält eine ID, einen Titel und eine unbegrenzte Anzahl von Architekturkomponenten.

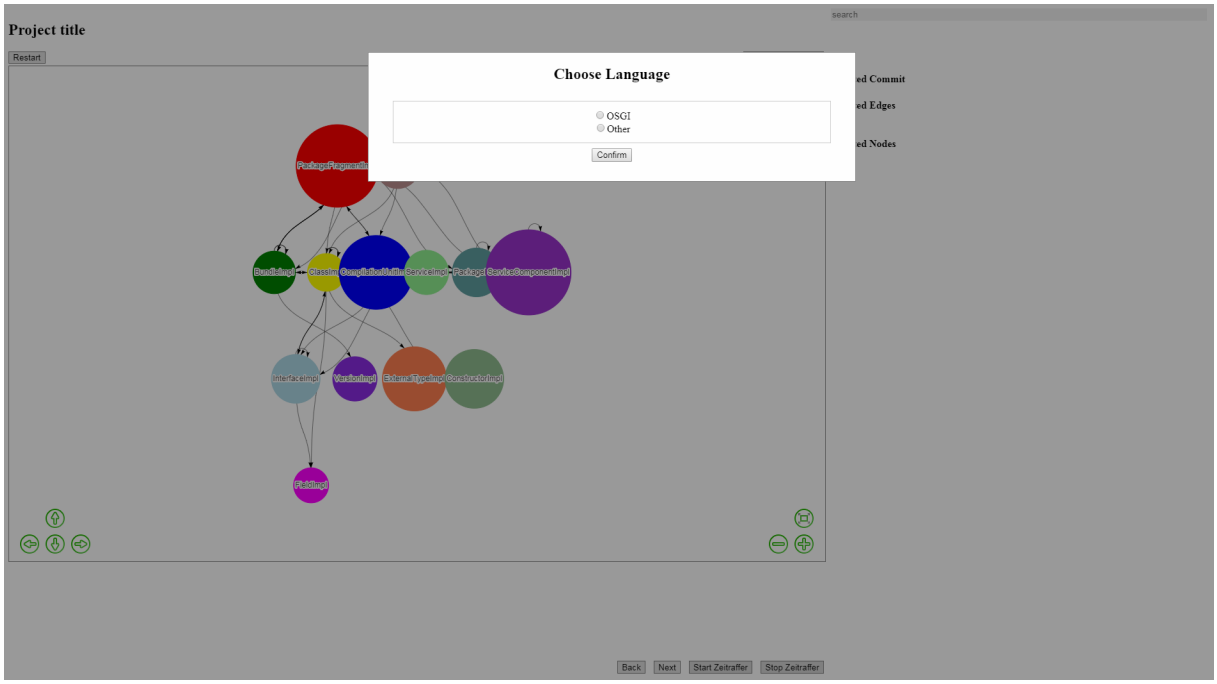


Abbildung 8: Formular zum Auswählen der Programmiersprache

Die Programmiersprachen werden nach dem Öffnen des Formulars mithilfe eines XML-Request und ihrem Tag `language` aus der Datei abgefragt und mit jeweils einem `Radiobutton` im Formular aufgelistet. Nun kann einer der Radiobuttons und somit die Sprache ausgewählt und dann mit dem `Confirm`-Button bestätigt werden. Bevor allerdings das dritte Formular, zur Zuordnung der Node Labels des Datenbankschemas zu den sprachabhängigen Ansichten erzeugt wird, wird ein erneuter `XML-Request` durchgeführt. Hierbei werden die Ansichten abgefragt, die durch die Komponenten bestimmt und deshalb mit dem Tag `Component` in der jeweiligen Sprache hinterlegt sind. Diese Komponenten werden abgespeichert und dann im folgenden Formular als Ansichten dargestellt. Des Weiteren werden an dieser Stelle im Hintergrund die Buttons für die Ansichten entsprechend generiert.

```
<?xml version="1.0" encoding="UTF-8"?>
<languages>
    <language id="OSGI">
        <title>OSGI</title>
        <component>Bundle</component>
        <component>Package</component>
        <component>Class</component>
        <component>Service</component>
    </language>
</languages>
```

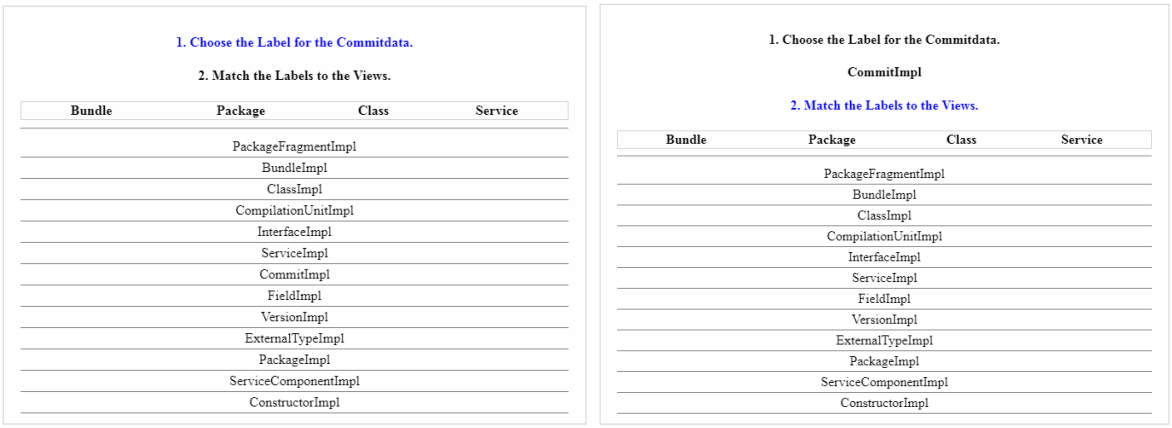
Abbildung 9: XML-Datei der Programmiersprachen

Durch die Angabe der Programmiersprache durch den Benutzer werden sowohl A05 zur Beschriftung der Ansicht-Buttons als auch A06 zur Anzahl der Ansichten, der statischen Aspekte aus Tabelle 2 aufgehoben, da diese nun abhängig von der ausgewählten Sprache generiert werden. Voraussetzung dafür ist, dass die jeweilige Programmiersprache und ihre architekturrelevanten Komponenten korrekt in der entsprechenden XML-Datei hinterlegt sind.

4.1.3 Zuordnung der Ansichten

Im dritten Formular für die Konfiguration des Systems durch den Benutzer werden die Node Label, die dem Datenbankschema zu Beginn entnommen wurden, den Ansichten zugeordnet, die durch die Sprache in der XML-Datei definiert wurden. Abbildung 10a zeigt das Formular unmittelbar nach dem Öffnen. Die vier Ansichten der Sprache werden mittig angezeigt und die Node Label in einer Liste darunter.

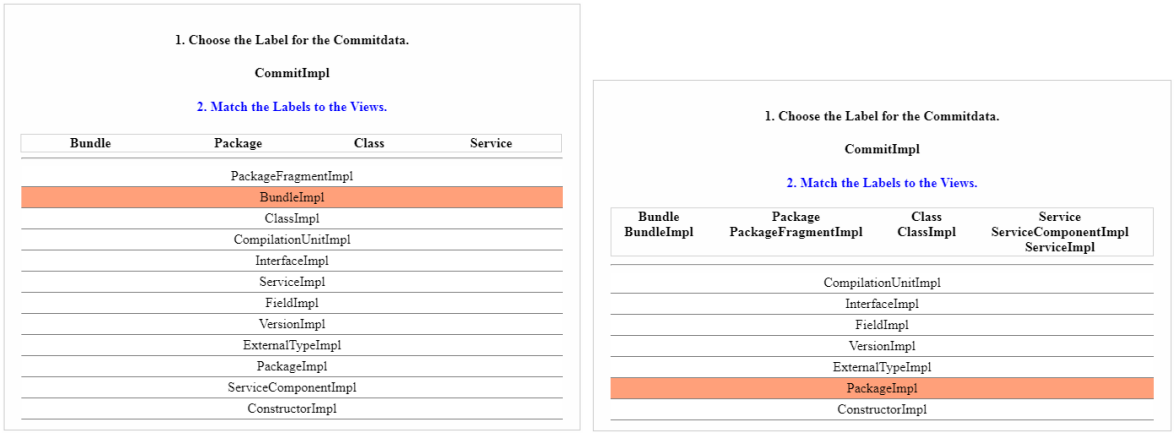
Im ersten Schritt soll der Benutzer das Label auswählen, indem die Knoten der Commits hinterlegt sind. Das wird dem Benutzer durch die abhebende Färbung des Satzes „`1. Choose the Label for the Commitdata.`“ mitgeteilt. Durch Anklicken des jeweiligen Labels aus der Liste, wird dieses, wie in Abbildung 10b zu sehen, unter den Satz geschrieben und aus der Liste entfernt. Nun hebt sich die zweite Anweisung „`2. Match the Labels to the Views.`“ vom restlichen Text farblich ab, um dem Benutzer den nächsten Schritt aufzuzeigen.



(a) Unberührtes Formular (b) Zuordnung des Commit-Labels

Abbildung 10: Auswahl eines Labels für die Commits

Um ein Label zu einer Ansicht zuzuordnen, wird dieses aus der Liste ausgewählt, wodurch es sich farblich hervorhebt bis es zugeordnet wurde. Abbildung 11a zeigt die farbliche Hervorhebung des ausgewählten Labels. Die Zuordnung folgt durch einen weiteren Klick auf eine der Ansichten. Das zuvor gewählte Label wird unter die jeweilige Ansicht geschrieben und aus der Liste entfernt. Dieses Verfahren kann solange durchgeführt werden, bis kein Label mehr in der Liste vorhanden ist. Die Richtigkeit dieser Zuordnung obliegt dem Nutzer.



(a) Auswahl eines Labels (b) Zuordnungen

Abbildung 11: Zuordnung der relevanten Labels zu den Ansichten

Abbildung 11b zeigt eine beinahe vollständige Zuordnung im Beispiel des vorhandenen Datensatzes. Die übrigen Label in der Liste werden nicht weiter benötigt und außer im Datenbankschema nicht weiterverwendet.

Nachdem die Zuordnung durch einen weiteren `Confirm`-Button im Formular bestätigt wurde, vervollständigt sich die Webseite. Dabei wird zum einen die Legende erstellt und zum anderen die Funktion für die Extrahierung und Visualisierung der Commits im Zeitstrahl aufgerufen, welche im folgenden Kapitel 4.2 näher beschrieben wird. Die Legende benötigt sowohl die Label als auch eine Farbe, die bei jedem vorkommen der Knoten damit übereinstimmt. Dafür ist der `colorhandler` zuständig, der ein `Array` aus Farben und ein Array der gesamten Node Labels indexweise zuordnet. Im Farb-Array sind aktuell 15 Farben hinterlegt; sollten mehr als 15 Node Label in einem Datenbankschema vorkommen, müsste dieses um weitere Farben erweitert werden.

Mithilfe einer Abfrage aller Node Labels aus der Datenbank konnte Aspekt A01 zu den statisch festgelegten Labeln aus der Tabelle aufgehoben werden, sodass diese nun dynamisch an das Formular übergeben werden. In diesem werden durch die Zuordnung die Node Label entsprechend an die Ansichten übergeben; somit wurden diese abhängig vom Nutzer definiert, anstatt statisch im Programmcode. Und durch das dynamische Erstellen der Node Legende anhand der extrahierten Node Labels und anhand der Farbgebung mithilfe des Colorhandlers kann der statische Aspekt A10 aus Tabelle 2 aufgehoben werden.

4.2 Visualisierungsprozess

Der Visualisierungsprozess bestand in dieser Arbeit aus den Schritten: Extrahierung, Verarbeitung und Visualisierung. Im Folgenden werden diese drei Schritte jeweils für die Zeitleiste (Timeline) und die Graphen (Network) ausführlich beschrieben. Dabei werden die verwendeten Cypher-Abfragen für die Extrahierung aufgelistet und näher erläutert, die Verarbeitung und dafür verwendeten Funktionen der Daten beschrieben und zum Abschluss die Visualisierungen mit entsprechenden Abbildungen dargelegt.

4.2.1 Extrahierung

Bei der Extrahierung der Daten werden diese bereits gefiltert, damit kein unnötiger Datenüberschuss entsteht und die Ausführungszeiten minimiert werden können. Um diese Filterung bei der Abfrage vornehmen zu können, werden Parameter wie Node Label oder Relationship Type benötigt, aber auch die Property Keys für weitere Eingrenzungen und für die anschließende Weiterverarbeitung. Diese Variablen wurden in den folgenden Tabellen farblich hervorgehoben. Der Property Key der Zeitangabe und diese selbst stehen im Fokus der Extrahierung der Daten.

Timeline

Aus Tabelle 3 ist zu entnehmen, dass es drei Abfragen braucht, um die benötigten Daten für die Zeitleiste zu erhalten. Vor der Generalisierung wurde lediglich eine Abfrage übermittelt, da der Property Key der Unix-Zeit bereits bekannt war. Dieser muss in der Generalisierung erst abgefragt werden, um ihn als Parameter zur Sortierung und für die weitere Verarbeitung nutzen zu können.

Tabelle 3: Cypher-Abfragen: Extrahierung der Daten für die Zeitleiste

#	Cypher-Abfrage
T1	MATCH (n:Label) UNWIND keys(n) AS key RETURN DISTINCT key
T2	MATCH (n:Label) RETURN n LIMIT 1
T3	MATCH (n:Label) RETURN n ORDER BY n.UnixTimeKey

Als Label jeder dieser Abfragen aus Tabelle 3 wird das vom Nutzer in der Zuordnung angegebene Label für die Commits übergeben. Abfrage T1 fragt alle Property Keys des Node Labels ab und mithilfe des Stichworts `DISTINCT` wird eine mehrfache Auflistung des selben Keys verhindert. Im Falle des vorhandenen Datensatzes (vgl. Kapitel 2.4.5) würde diese Abfrage zweierlei Zeitkomponenten zurückgeben: `time` im Unix-Zeitformat und `timestamp` im lesbaren Zeitformat. Auch bei der Integration einer CSV-Datei, die aus den Commits des Gits entstanden ist, würden diese zwei Zeitformate übergeben werden (vgl. Abbildung 5), weshalb an dieser Stelle nun davon ausgegangen wird, dass dies der Standard ist.

Da das System das Format des jeweiligen Property Keys nicht kennt, und somit nicht zuordnen kann, welche Zeitkomponente jeweils dargestellt wird, wird Abfrage T2 ausgeführt. Dabei wird ein Knoten des Labels aus der Datenbank extrahiert, um an diesem die jeweiligen Zeitkomponenten zu erfassen, was im folgenden Kapitel zur Verarbeitung näher beschrieben wird.

Nachdem die Property Keys definiert und den Zeitformaten zugeordnet wurden, werden mithilfe von Abfrage T3 die Commits aus der Datenbank extrahiert. Mit dem Property Key, der die Unix-Zeitkomponente darstellt, erfolgt die Sortierung, sodass die Commits in zeitlicher Reihenfolge verarbeitet werden können. Die Weiterverarbeitung wird im anschließenden Kapitel 4.2.2 behandelt.

Network

Im Falle des Datenbankschemas wird lediglich der Cypher-Befehl `call db.schema()` zum Extrahieren der Knoten und Kanten benötigt. Für die Daten der Graphen in den Ansichten werden je Node Label mindestens vier Abfragen an Neo4j übermittelt. Wurde mehr als ein Node Label einer Ansicht zugeordnet, werden entsprechend mehr Abfragen gestellt. Die folgende Tabelle 4 enthält die möglichen fünf Abfragen, die benötigt werden für die Weiterverarbeitung und die anschließende Visualisierung der jeweiligen Ansicht.

Tabelle 4: Cypher-Abfragen: Extrahierung der Daten für die Graphen der Ansichten

#	Cypher-Abfrage
N1	MATCH (n:Label) UNWIND keys(n) AS key RETURN DISTINCT key
N2	MATCH (n:Label) RETURN n LIMIT 1
N3	MATCH (n:Label)-[r]->(m) WHERE n.UnixTimeKey=UnixTimestamp AND m:Label RETURN DISTINCT r
N4	MATCH (n:Label)-[r]->(m) WHERE n.UnixTimeKey=UnixTimestamp AND m:OtherLabel RETURN DISTINCT r
N5	MATCH (n:Label) WHERE n.UnixTimeKey=UnixTimestamp RETURN n

Der Unterschied der Cypher-Abfragen aus Tabelle 3 zu denen in Tabelle 4 liegt im Label, welches hier entsprechend der Labels der aktuellen Ansicht übergeben wird. Für jedes Node Label in einer Ansicht, das der Benutzer in der Konfiguration zugeordnet hat, werden die Abfragen der Tabelle ausgeführt.

Die Cypher-Abfragen N1 und N2 aus der Tabelle 4 sind die gleichen wie T1 und T2 aus der vorherigen Tabelle 3 zur Timeline. Wiederum geht es bei N2 nicht um die Verarbeitung von zwei verschiedenen Zeitkomponenten aus N1, sondern um die Namensgebung des Knoten.

Im vorhandenen Datensatz sind in vier verschiedenen Property Keys in teilweise unterschiedlichen Formaten Namen hinterlegt. Dabei enthält ein Name beispielsweise auch die Namen der übergeordneten Knoten wie ein Pfad und ein anderer nur die Bezeichnung des Knotens selbst. Das bedeutet, ein Knoten kann mehrere Namen in unterschiedlichen Keys haben, was bei weiteren Datensätzen wieder der Fall sein könnte.

Zusätzlich zu den Property Keys der Namen von Knoten wird der Key für die Zeitangabe benötigt, um diese in den weiteren Abfragen als Zeitpunkt übergeben zu können. Deshalb wird mit der Abfrage N2 wieder ein einzelner Knoten aus der Datenbank extrahiert, mit dem die Zuordnung der Eigenschaften zu den vorhandenen Property Keys erfolgt. Die weitere Verarbeitung dieser Daten wird dann ausführlich in Kapitel 4.2.2 beschrieben.

Die Abfrage N3 extrahiert alle Kanten zwischen zwei Knoten eines Labels der aktuellen Ansicht. Falls in einer Ansicht beispielsweise ein zweites Node Label hinterlegt ist, wird dieses mit OtherLabel in N4 übergeben, um so die Beziehungen zwischen Knoten unterschiedlicher Labels zu erhalten. In beiden Abfragen werden der Property Key der Unix-Zeitkomponente und die Zeit des ausgewählten Commits übergeben. In der letzten Abfrage der Tabelle werden alle Knoten des Labels zum aktuell ausgewählten Zeitpunkt abgefragt.
Durch das Abfragen der Property Keys der zu visualisierenden Nodes der Ansicht können diese dynamisch zur Weiterverarbeitung übergeben werden. Dadurch wird der Aspekt A03 zu den statisch übergebenen Property Keys aus Tabelle 2 aufgehoben. Des Weiteren werden die Relationship Types durch diese Methode der Extrahierung der Kanten nicht für die Abfragen benötigt, weshalb Aspekt A02 zu den statisch übergebenen Relationship Types wegfällt. Durch die dynamische Erweiterung der Anzahl von Abfragen für die Knoten und Kanten, die abhängig von der Anzahl der Node Labels einer Ansicht sind, wurde ein weiterer Aspekt (A07) aus der Tabelle angepasst.

4.2.2 Verarbeitung

Die Verarbeitung, im Visualisierungsprozess auch „Mapping“ genannt, beinhaltet die Zuordnung und Adaption der extrahierten Daten zu den Elementen der Visualisierungsart. In diesem Fall wurden die Daten bereits als Knoten und Kanten extrahiert, um sie dann auch als solche zu visualisieren. Mapping muss hier allerdings in Form von korrekter Zuordnung von Property Keys vor der anschließenden Visualisierung vorgenommen werden.

Timeline

Die Verarbeitung der extrahierten Daten für die Zeitleiste besteht, wie die Cypher-Abfragen dafür aus Tabelle 3, aus mehreren Schritten. Als erstes werden nach der Abfrage T1 aus Tabelle 3 alle Property Keys, in denen das Stichwort „time“ vorkommt, abgespeichert. An dieser Stelle wird für die gesamte Arbeit die Annahme getroffen, dass die Property Keys in englischer Sprache benannt wurden.

Die Liste mit den Keys, in denen time vorkommt, wird im nächsten Schritt, nach Abfrage T2, Eigenschaften zugeordnet. Die Länge des Inhalts dieser Keys eines Knotens ist der entscheidende Faktor für die Zuordnung.

Die Unix-Zeitkomponente hat 10 Stellen und wird bei der Umrechnung in eine lesbare Zeitkomponente zuerst mal 1000 genommen, bevor diese konvertierbar ist. Im Falle des vorhandenen Datensatzes wurde bereits Unix-Zeit mal 1000 integriert, weshalb diese hier 13 Stellen enthält. Eine lesbare Zeitkomponente, ungeachtet des Formats, kann nicht nur mit 13 Stellen dargestellt werden, da diese mindestens aus Datum und Uhrzeit besteht. Somit kann eine Abfrage der Länge des Inhalts ausschlaggebend dafür sein, ob es sich bei dem Property Key um die Unix- oder eine andere Zeitkomponente handelt.

Wurden die Property Keys den Eigenschaften zugeordnet, wird die Abfrage nach den Commits übermittelt und anschließend das Resultat verarbeitet. Die Unix-Zeitkomponente muss in ein von Neo4j verwertbares Format konvertiert werden: `YYYY-MM-DDThh:mm:ss`. Das `T` stellt dabei lediglich einen Trennbuchstaben zwischen Datum und Uhrzeit dar. Dafür wird zunächst geprüft, ob es sich um eine einfache Unix-Zeit handelt, die beim Umrechnen zuerst mal 1000 gerechnet werden muss, oder ob diese bereits multipliziert hinterlegt ist. Im Anschluss wird dann mit Javascripts `Date()` konvertiert und mit weiteren vordefinierten Funktionen wie `getFullYear()` oder `getHours()` zum genannten Format zusammengesetzt. Der Datensatz zur Übergabe an die Visualisierung der Zeitleiste besteht aus den Commits als Zeitpunkte.

Network

Ähnlich wie zuvor bei der Zeitleiste wird nach der ersten Cypher-Abfrage N1 aus Tabelle 4 eine Übereinstimmung der Worte „time“ und „name“ in den resultierenden Property Keys überprüft. Der Grund ist die Zuordnung der Keys zu den Eigenschaften, die Zeitkomponente und Namen der Nodes darstellen. Wie bereits erläutert kann ein Node Label mehrere Property Keys mit Namen enthalten, weshalb hier ein Array mit Keys für die Namen angelegt wird.

Im nächsten Schritt, nachdem ein Knoten mit einem Node Label der aktuellen Ansicht extrahiert wurde, wird überprüft, ob mehrere Property Keys mit Namen im Array vorhanden sind. Sollte dies der Fall sein, wird der mit dem längeren Inhalt als Key mit dem Namen des Knotens übergeben, da davon ausgegangen werden kann, dass ein längerer Name mehr Aussagekraft hat. Bei gleicher Länge des Inhalts wird der erste der beiden übergeben, da eine Auswahl getroffen werden muss.

Im Anschluss an die Zuordnung der Property Keys zu den entsprechenden Eigenschaften eines Knotens werden diese und die passenden Kanten dazwischen extrahiert. Bei der Übergabe der Daten der Kanten hat sich im Gegensatz zu den Knoten nichts verändert. Es werden weiterhin Start- und End-Knoten, Kanten-ID und Relationship Type an die Visualisierung übergeben. Im Falle der Knoten werden die Namen zusätzlich in einer weiteren Liste abgespeichert, die im Anschluss an die Suchleiste zur automatischen Vervollständigung übergeben wird.

Des Weiteren wird an dieser Stelle erneut der Colorhandler aufgerufen um die entsprechende Farbe je nach Node Label zurück zu geben. Der Visualisierung werden die Knoten mit ID, Name, Label und Farbgruppe übergeben.

In der Verarbeitung wurden die Eigenschaften und deren Format den Property Keys zugeordnet, damit sie korrekt interpretiert und eventuell konvertiert werden können. Aspekt A04 zum statischen Format der Daten aus Tabelle 2 wurde hier behandelt, sodass eine mögliche Konvertierung unabhängig vom Format der Daten weiterhin funktioniert.

4.2.3 Visualisierung

Die Bibliothek Vis benötigt zur Visualisierung einen `Container`, Daten und Optionen. Der sogenannte Container legt den Bereich der Visualisierung fest, deren Begrenzung und somit die Stelle an der diese erstellt wird. Die Optionen sind vordefinierte Einstellungsmöglichkeiten zu verschiedenen Aspekten der Visualisierung, wie beispielsweise zu den Knoten, Kanten oder dem Layout im Network. Jede Option lässt sich individuell anpassen oder es werden die Default-Werte verwendet. Die Möglichkeiten sind bei den beiden verwendeten Visualisierungsarten Timeline und Network sehr unterschiedlich.

Timeline

Die wichtigsten Optionen der Timeline sind die booleschen Aussagen über die Selektierbarkeit und die Veränderbarkeit, aber auch das Anzeigen eines Tooltips und das Verwenden nach dem Anklicken. Diese sind bis auf die Veränderbarkeit auf `true` gesetzt worden, da diese Funktionen ein wichtiger Teil des Systems sind. Die Veränderbarkeit wird nicht unterstützt, da das bedeuten würde, dass der Nutzer die Zeitleiste manipulieren könnte. Des Weiteren wurde in den Einstellungen die Größe der Boxen, die die Zeitpunkte darstellen, sowie die Form dieser festgelegt. Folgende Abbildung 12 zeigt die visualisierte Zeitleiste des vorhandenen Datensatzes.

Abbildung 12: Visualisierung der Zeitleiste/Timeline

Zu den Optionen, die an die Visualisierung übergeben werden, gehören auch ein Start- und ein Endpunkt auf der Zeitleiste, damit diese überhaupt erstellt wird. Damit diese Daten sinnvoll und dynamisch an die Commits angepasst sind, werden sie berechnet. Dafür wird für den Start der Tag auf den ersten des Monats vom ersten Commit gesetzt und für das Ende auf den ersten Tag vom folgenden Monat des letzten Commits.

Das bedeutet, wenn der erste Commit am 20.03.2016 getätigt wurde, würde das Startdatum der Zeitleiste auf den 01.03.2016 und das Enddatum wegen des letzten Commits am 05.05.2018 auf den 01.06.2018 fallen. Die Start- und Endpunkte der Zeitleiste werden im Format `YYYY-MM-DD` übergeben, nachdem alle Commits verarbeitet wurden und bevor diese visualisiert werden.

Das Event, das durch Selektion eines Zeitpunktes ausgelöst wird, ist erhalten geblieben, sodass der Nutzer weiterhin zwischen den Commits navigieren kann. Lediglich der Funktionsaufruf hat sich in der Generalisierung verändert, da nur noch eine Funktion für alle Ansichten existiert anstatt für jede eine. Auch bei den Vor- und Zurück-Buttons hat sich bis auf den Funktionsaufruf nicht viel im Vergleich zur statischen Version verändert.

Network

Zur Visualisierung des Schemas werden lediglich die Knoten und Kanten aus der Abfrage übergeben und der Colorhandler für die Farbgebung aufgerufen. Durch den Aufruf des Colorhandlers an dieser Stelle wurde ein weiterer statischer Aspekt aus Tabelle 2 behandelt, A11 zur Farbgebung der Nodes.

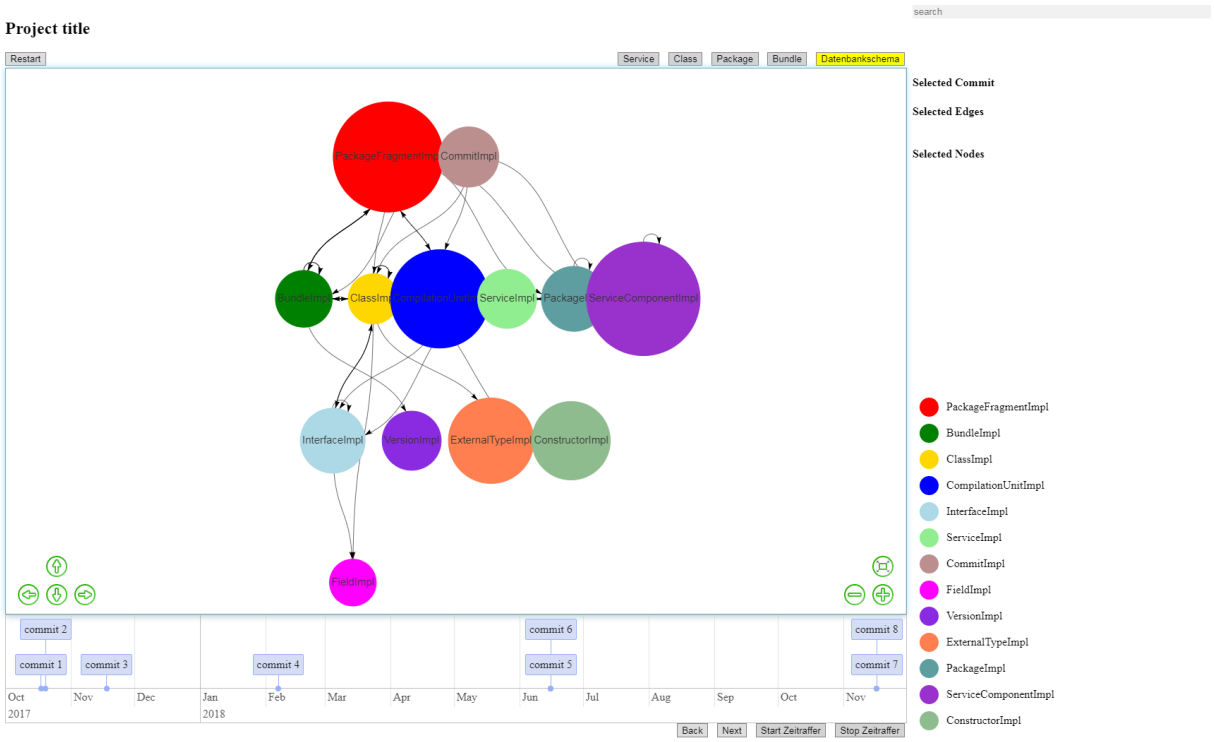


Abbildung 13: Visualisierung des Datenbankschemas auf der Webseite

Der Unterschied zu den Graphen der Ansichten liegt in den Einstellungen, die für das Datenbankschema im Bezug auf das Layout und an wenigen anderen Stellen, angepasst werden mussten.

Die Einstellungen zu den Kanten wurden angepasst, sodass die Pfeile geschwungener, länger und somit eindeutiger bei Überschneidungen erkennbar sind, da weniger Knoten auf einmal angezeigt werden. In den Nodes wird das Label direkt angezeigt und deren Größe daran angepasst. Das Layout wird hierarchisch nach der Richtung der Beziehungen aufgebaut, wie in Abbildung 13 der Webseite erkennbar ist.

In den Einstellungen für die Graphen der Ansichten werden die Knoten alle gleich groß und ohne Beschriftung festgelegt. Die Kanten sind geradlinig mit einem kleinen Pfeil ausgestattet, um die Richtung zu definieren. Das Layout ist nicht hierarchisch angeordnet; stattdessen wurde die Möglichkeit des `improvedLayout` aktiviert, sodass sich die Knoten und Kanten mit möglichst wenig Überschneidungen anordnen. Des Weiteren wurden, wie auch bei den Optionen des Datenbankschemas, die `navigationButtons` aktiviert. Diese sind in den unteren Ecken der Visualisierung in Abbildung 13 und 14 zu erkennen. Die Buttons dienen zur Navigation innerhalb der Visualisierung, wozu das Zoomen mit den Plus- und Minus-Symbolen, Verschieben mit den Pfeiltasten in die entsprechenden Richtungen und das Herauszoomen auf den gesamten Graphen mit dem `ExtendedZoom`-Button gehören. Sie sind bereits von der Bibliothek definiert und funktionstüchtig, solange sie in der vorgesehenen Ordnerstruktur hinterlegt wurden.

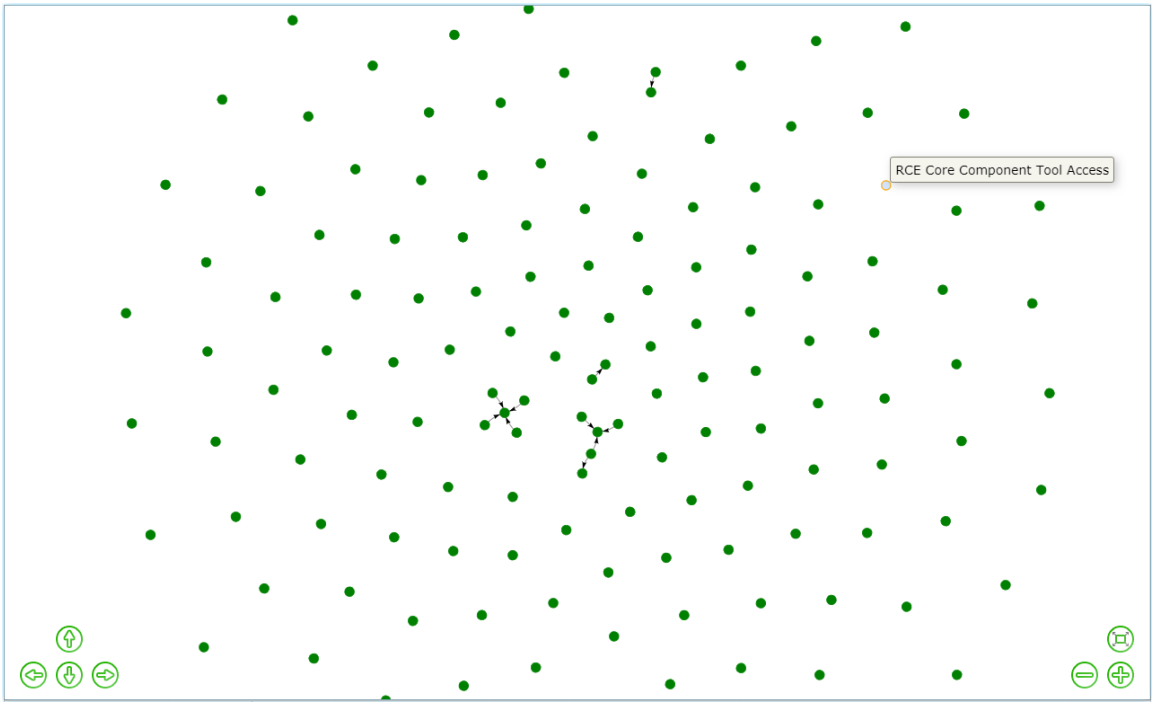


Abbildung 14: Visualisierung der Bundles im letzten Commit

Die Events, die beim Datenbankschema durch Hovern oder Selektieren ausgelöst werden, sind dieselben wie bei den Graphen der Ansichten. Das Hovern über ein Element lässt einen Tooltip erscheinen mit dem Namen des Knotens oder dem Typ der Kante. Die Tooltip-Funktion ist in Abbildung 14, die die Bundles des 8. Commits visualisiert, zu erkennen. Durch Selektieren wird nach wie vor Zusatzinformation zum Element im Kontextinformationsbereich auf der rechten Seite des Bildschirms angezeigt.

4.3 Vorgehensweise

Im Folgenden sollen noch einmal der Ablauf und die Vorgehensweise der implementierten Software betrachtet und mit der Ausgangssituation abgeglichen werden. Zur besseren Übersicht dient der Programmablaufplan aus Abbildung 15. Dabei wurde das Diagramm auf die nötigsten Aspekte für ein funktionierendes System reduziert. Es wird davon ausgegangen, dass zum Start der Software, wie sie im Ablaufplan dargestellt wird, eine aktive Datenbank mit Daten existiert. Andernfalls gibt es nichts zu visualisieren.

Die Software startet mit dem Aufruf der Webseite, mit der sich das erste Formular zum Login der Datenbank öffnet. Im Formular wird die Eingabe der aktiven Datenbank-URL und des Benutzernamens und Passworts einer zugriffsberechtigten Person verlangt. Mit dem Bestätigen der Eingaben wird im Hintergrund das Datenbankschema aus Neo4j extrahiert, verarbeitet und visualisiert und ein XML-Request ausgeführt, um die in der XML-Datei hinterlegten Programmiersprachen an das Formular zur Auswahl der Sprache übergeben zu können. In diesem wählt der Benutzer eine der aufgelisteten Programmiersprachen, in der die zu betrachtende Software implementiert wurde. Mit dem Bestätigen der Auswahl wird ein erneuter XML-Request durchgeführt, um die Architekturrelevanten Komponenten der entsprechenden Sprache aus der XML-Datei zu erfassen. Die Ergebnisse werden an das Formular zur Zuordnung der Node Labels zu den Ansichten übergeben und eingebunden. Nun ordnet der Benutzer erst das Label für die Nodes, die die Commits darstellen, zu und anschließend die restlichen Node Label zu den Ansichten, wobei die Node Label aus dem Datenbankschema stammen und die Ansichten aus dem XML-Request. Durch Bestätigen der Zuordnung wird die Webseite vervollständigt, indem die Commits aus Neo4j extrahiert und durch die Zeitleiste visualisiert werden. An dieser Stelle hat der Benutzer mehrere Möglichkeiten, da es einige Buttons gibt, die er anwählen kann. Es ist unerheblich welchen Button der Benutzer betätigt oder in welcher Reihenfolge, in manchen Fällen passiert lediglich nichts.

Wählt der Nutzer einen der Ansicht-Buttons, wird die entsprechende Ansicht im letzten Commit visualisiert. Beim Anwählen einer der Commits wird dieser in der ersten erfassten Ansicht dargestellt. In beiden Fällen werden Daten in der entsprechenden Ansicht zum gewählten Zeitpunkt extrahiert und der Graph visualisiert. Von da aus kann wieder jeder beliebige Button angewählt werden. Wird nun ein Commit oder eine Ansicht gewählt, bleibt die aktuelle Ansicht bzw. der aktuelle Commit bestehen.

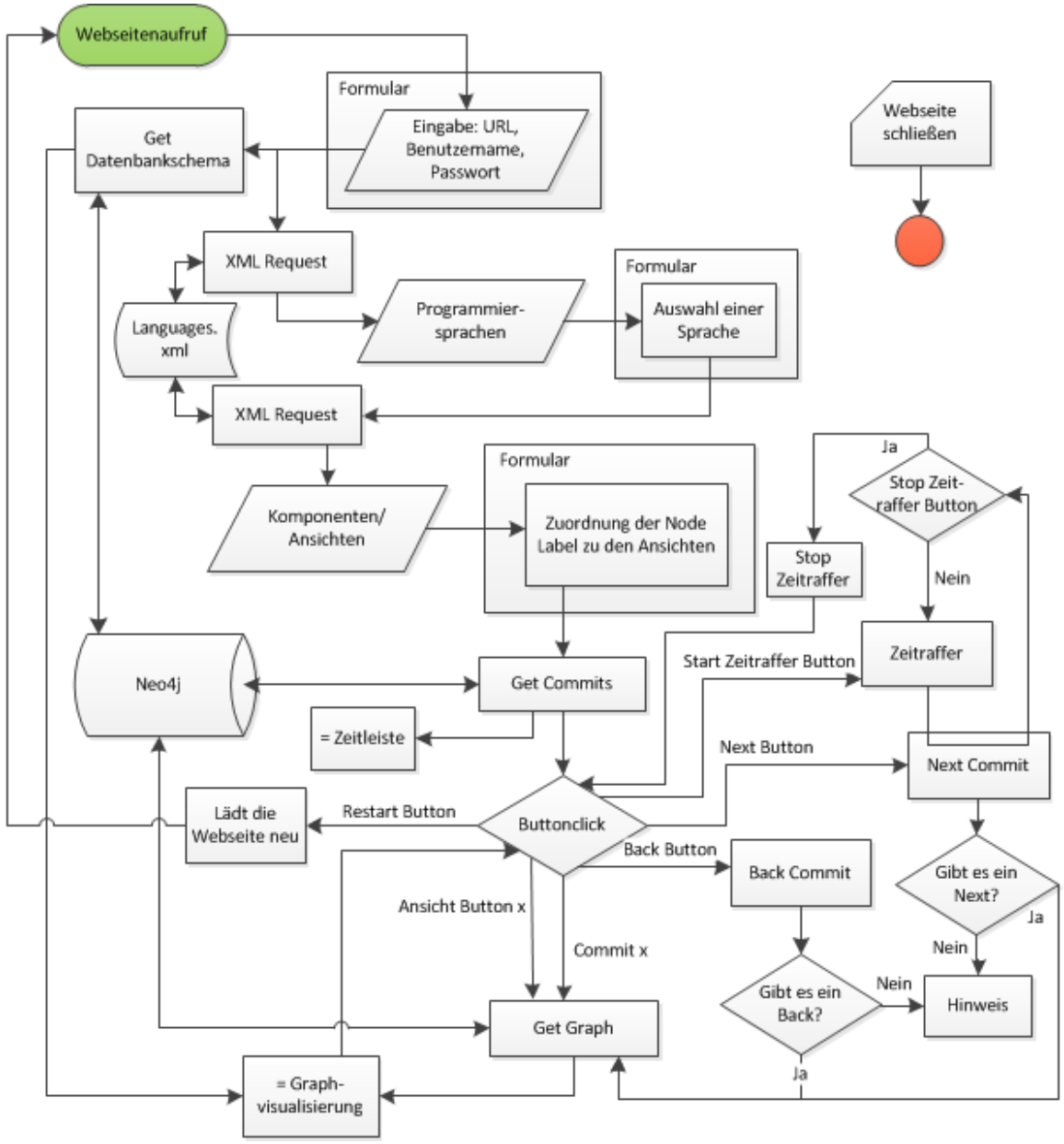


Abbildung 15: Programmablaufplan des generalisierten Systems

Mithilfe des Next- und Back-Buttons lässt sich einfach zwischen den Commits navigieren. Werden diese jedoch als erstes angewählt, noch bevor ein Graph visualisiert wurde und somit weder Ansicht noch Commit gegeben sind, werden der letzte bzw. erste der Commits ausgewählt und die erste erfasste Ansicht. Zusätzlich gibt es die Buttons zum Starten und Stoppen des Zeitraffers, um durch die Commits zu navigieren. Dieser läuft ab der aktuell angewählten Stelle solange durch die Commits, bis der letzte erreicht ist oder der Stop-Button betätigt wird.

Der letzte Button ist der Restart-Button zum Neuladen der Webseite, welcher das System neu startet, damit von vorne begonnen werden kann. Das System wird beendet, wenn die Webseite geschlossen wird.
Wie sich dem Programmablaufplan im Vergleich zu dem der Ausgangssituation in Abbildung 3 entnehmen lässt, wurden die einzelnen Funktionen zum Auswählen einer Ansicht entfernt und somit der statische Aspekt A08 aus Tabelle 2 aufgehoben. Die Navigation zwischen den Ansichten erfolgt über ein Event, das jedem der aus den Ansichten generierten Buttons übergeben wurde. Durch Auslösen wird nun direkt die Ansicht ausgewählt und der entsprechende Graph visualisiert. Des Weiteren wurden die vier statischen Funktionen für die Visualisierung der Graphen der Ansichten, welche die Aspekte A12 zur Namensgebung dieser und A13 zur Anzahl dieser in Tabelle 2 darstellen, zu einer Funktion zusammengefasst.

Die Antwort auf die zentrale Fragestellung dieser Arbeit, wie die Visualisierung flexibel auf verschiedene Schemata reagieren kann, ist die Arbeit selbst und die im Rahmen dieser erarbeitete Lösung. Mit Sicherheit kann die Frage an dieser Stelle nicht beantwortet werden, da dazu die Evaluation der Umsetzung nötig ist, die im anschließenden Kapitel 5 durchgeführt wird.

4.4 Erweiterungsmöglichkeiten

Die erste nachträglich zu implementierende Funktion wäre die Warnung bei falscher Eingabe der Login-Daten. Das nächste Formular zur Sprachauswahl wird ohnehin angezeigt und dem Benutzer wird lediglich durch das ausbleibende Visualisieren des Datenbankschemas im Hintergrund signalisiert, dass der Login fehlgeschlagen ist. Spätestens beim dritten Formular zum Zuordnen der Labels zu den Ansichten wird deutlich, dass zuvor etwas nicht funktioniert hat, da keine Node Label angezeigt werden und somit nichts zum Zuordnen. Die Seite muss neu geladen werden, um von Neuem beginnen zu können.

Eine weitere nicht komplett dynamische Funktion ist der Colorhandler. Dieser ordnet ein Array von Labeln und ein Array von vordefinierten Farben desselben Index zu. Zudem sind im Array aktuell 15 Farben hinterlegt, die bewusst nicht zu dunkel und gut unterscheidbar gewählt wurden, da Eindeutigkeit sichergestellt werden sollte. Sobald ein Graph mehr als 15 Node Label enthält, müsste man händisch im Code weitere Farben in das Array schreiben. Eine zufällige Zuordnung der in HTML hinterlegten Farben müsste so implementiert werden, dass eine Dopplung nicht vorkommen kann. Des Weiteren sollten die Farben nicht zu dunkel oder zu hell gewählt werden, damit es für den Benutzer nicht unangenehm oder schwer lesbar wird. Es könnte ein bestimmter Bereich festgelegt und die im Farbcode benachbarten Farben ausgeschlossen werden, sodass keine Verwechslungen der Knoten wegen zu großer Ähnlichkeiten entstehen. Bei der Farbauswahl könnte ebenfalls auf psychologisch sinnvolle Farben und Kombinationen Rücksicht genommen werden, sowie auf Menschen mit einer Rot-Grün-Schwäche. Diese Aspekte wurden in dieser Arbeit bisher völlig außer Acht gelassen.

Der „Project title“ am oberen Rand der Webseite ist statisch im HTML-Code für jedes Projekt. Die Funktion dahinter den tatsächlichen Titel des Projekts einzutragen, fehlt noch. An dieser Stelle könnte beispielsweise der Titel des Graphen aus Neo4j entnommen und als Projekttitel verwendet werden. Dadurch kann allerdings nicht der korrekte Projekttitel sichergestellt werden, da das wieder von der Person abhängt, die beim Erstellen und Integrieren des Graphen den Titel für diesen gewählt hat. Eine sichere Methode wäre, den Benutzer in einem der bereits existierenden Formulare den Projekttitel eingeben zu lassen.

5 Evaluation

Im Folgenden wird der zuvor entwickelte Prototyp evaluiert, in dem die Software mit ihren Funktionen Tests unterzogen wird. Als Anwendungsbeispiel wird ein zweiter Datensatz verwendet, der sich in Programmiersprache und Schema vom bereits integrierten unterscheidet. Mit diesem werden alle Funktionen der Software getestet, um zu beweisen, dass diese Möglichkeit der Generalisierung eine universelle Lösung darstellt, um verschiedene Architekturen, unabhängig von der Sprache und dem Schema, zu visualisieren.

In der Abnahme wird der Erfolg oder Misserfolg des Prototyps erfasst und daraus ein Fazit gezogen. Im Anschluss folgt eine erneute Beantwortung der zentralen Fragestellung, die mit der vorherigen Antwort übereinstimmt oder diese revidiert.

5.1 Datensatz

Zum Testen wird eine Software benötigt, deren Architektur sich von der bereits integrierten absetzt. Dazu wurden die Daten aus dem Git eines C#-Projekts zur Visualisierung eines dynamischen Graphen entnommen, das mit *Unity* implementiert wurde. Dieses wurde in Neo4j integriert, sodass es im Folgenden direkt zum Testen verwendet werden kann.

C# verwendet Klassen und Interfaces, die in Namensräumen strukturiert werden. Durch die Entwicklung in Unity wurden diese in Ordnern und Dateien organisiert. In jeder Datei, die an der Endung `.cs` zu erkennen sind, befindet sich genau eine Klasse bzw. ein Interface, das denselben Namen trägt. Klassen oder Interfaces können zu einem Namensraum gehören. Ein weiterer signifikanter Unterschied zum vorherigen Datensatz ist die Vererbung, die C# mit sich bringt. Diese wurde durch Beziehungen zwischen Klassen und Interfaces dargestellt. Des Weiteren wird von Unity für jeden Ordner und jede Datei automatisch eine `.meta`-Datei erstellt, welche im Folgenden aber unberücksichtigt bleiben.

Für die Evaluation wurden lediglich relevante Daten aus dem Git übernommen. Aus diesem Grund bleiben die ersten beiden Commits unberücksichtigt, da diese nur eine `READ ME`- und `Gitignore`-Datei enthielten. Somit wurden letztendlich sechs Commits in die Datenbank integriert.

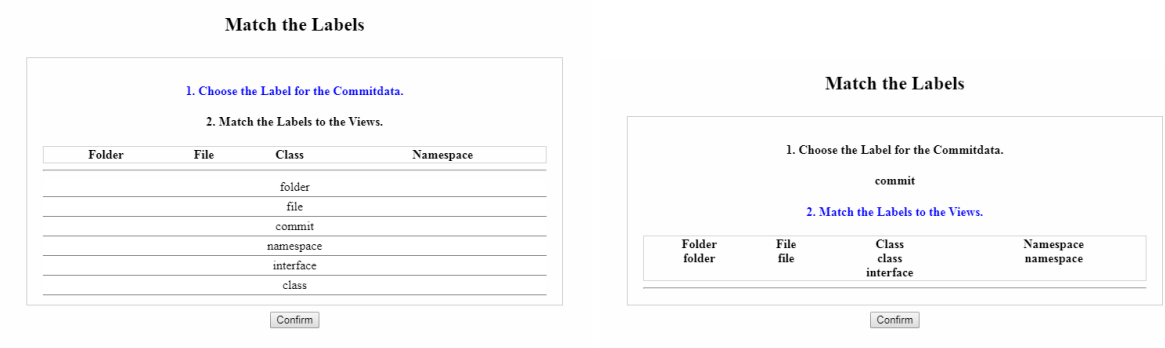
Der Graph in Neo4j enthält 440 Nodes, mit den Labeln `class, commit, file, folder, interface` und `namespace`, und 626 Edges, mit den Types `BELONGS_TO, CONTAINS` und `INHERITS_FROM`. Des Weiteren wurden die Property keys `message, name, time, timestamp` und `unixtime` hinzugefügt. Die Keys unixtime und timestamp enthalten beide die Unix-Zeitkomponente und wurden lediglich zu Testzwecken unterschiedlich benannt.

5.2 Testing

Zuerst wurde die Sprache C# zur language.xml hinzugefügt, indem die Ansichten entsprechend der Software definiert wurden, nämlich als Folder, File, Class und Namespace, und anschließend integriert wurden. Die Datenbank wurde aktiviert und die Webseite aufgerufen.

Beim Login werden die Datenbank-URL, Benutzername und Passwort übergeben, die von denen des vorherigen Datensatzes differieren. Der Verbindungsaufbau verlief erfolgreich, was durch die Visualisierung des Datenbankschemas im Hintergrund zu erkennen ist.

Die hinzugefügte Programmiersprache erscheint im Formular zur Sprachauswahl zusammen mit dem Radiobutton und nach der Auswahl, werden im Formular zur Zuordnung die entsprechenden Ansichten dargestellt sowie die Node Label aus der aktiven Datenbank. In Abbildung 16 ist zu erkennen, dass die Sprache und das Label korrekt, wie für den Test zugeordnet, übergeben wurden.



(a) Leeres Formular

(b) Zuordnungen

Abbildung 16: Zuordnung der Labels zu den Ansichten (Evaluation)

Nach der Bestätigung der Zuordnung vervollständigt sich die Webseite, sodass die Legende, die Zeitleiste und die Ansicht-Buttons zu sehen sind. Das Datenbankschema wird visualisiert und der entsprechende Button hervorgehoben, wie in Abbildung 17 deutlich wird. Außerdem wird aus der Abbildung ersichtlich, dass die „Berechnung“ des Start- und Endpunkts der Zeitleiste nicht für jeden Datensatz geeignet ist. Die Commits liegen in diesem Fall zu nah bei einander, sodass nicht alle Commit-Beschriftungen lesbar und somit teilweise nicht anwählbar sind. Außerdem werden im Datenbankschema die von Neo4j automatisch generierten Beziehungen mit angezeigt, was dieses unübersichtlicher wirken lässt, als es sollte.

Durch die Auswahl eines der Ansicht-Buttons werden die Knoten und Kanten, dem Default entsprechend im letzten Commit, visualisiert. So wird auch durch die erste Auswahl eines Commits eine Ansicht gewählt und entsprechend visualisiert. In diesem Fall ist die erste generierte Ansicht, die Folder-Ansicht. Die Ansichten werden korrekt visualisiert, was sich mithilfe von Vergleichen zwischen der Webseite und Neo4j beweisen lässt. Dafür wurden Anzahl und Namen der Knoten sowie das Vorhandensein der Kanten überprüft.

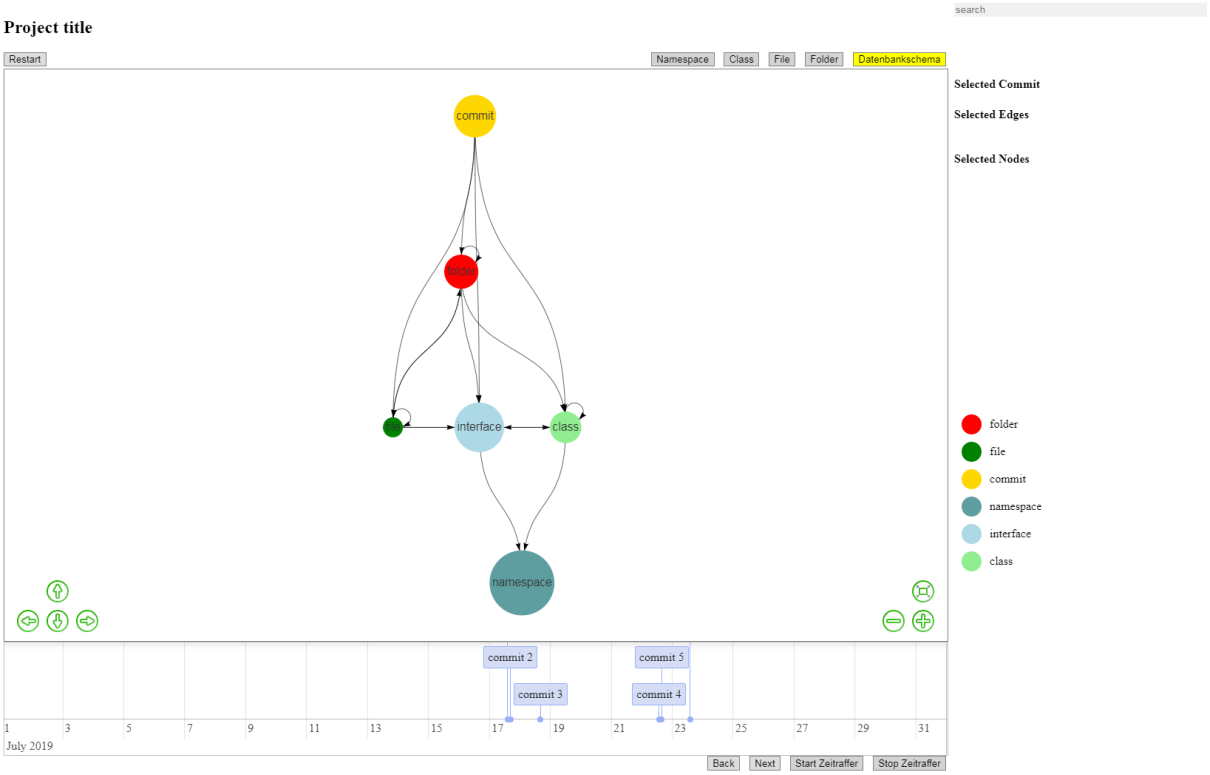


Abbildung 17: Vollständige Webseite (Evaluation)

Durch Anwählen der Knoten und Kanten lassen sich im Kontextinformationsbereich die Label und Types auf Richtigkeit überprüfen. Eine lesbare Zeitangabe des aktuell angewählten Commits sollte ebenfalls in diesem Bereich angezeigt werden. Da hier jedoch eine Zeitangabe um 1970 angezeigt wurde, musste die Ursache dafür im Programmcode gefunden und überarbeitet werden. In der Stelle wurde übersehen, dass die Unix-Zeitkomponente in diesem Datensatz nicht schon mit 1000 multipliziert hinterlegt worden war. Das bedeutet, dass bei der Konvertierung eine entsprechend kleine Zahl ein entsprechend zurückliegendes Datum liefert. Durch eine einfache Abfrage der Länge an der Stelle der Übergabe an den Kontextinformationsbereich konnte das Problem gelöst werden.

Beim Commitwechsel wird der entsprechend nächste Commit visualisiert, was sich beim Wechsel vom ersten zum zweiten, anhand der Anzahl der File-Nodes erkennen lässt, da zwei hinzugekommen sind. Die Back- und Next-Buttons erlauben die Navigation zwischen aufeinanderfolgenden Commits und der Zeitraffer läuft die Commits der Reihenfolge nach durch.

Über die Suchleiste lässt sich mithilfe von automatischer Vervollständigung nach Knoten in der aktuellen Visualisierung suchen. Durch Anwählen des Restart-Buttons wird die Webseite neu geladen.

Zusammenfassend lässt sich sagen, dass die Software, bis auf einen schnell korrigierbaren Fehler zur Zeitangabe, wie erwartet funktioniert und somit erfolgreich war. Sowohl die Konfiguration als auch der Visualisierungsprozess werden ordnungsgemäß mit den vorgesehenen Ergebnissen ausgeführt. Die Berechnung des Start- und Endpunktes auf der Zeitleiste sollte überarbeitet werden, damit diese optisch und funktional korrekt dargestellt wird. Es könnte eine komplexe Berechnung durchgeführt werden, die prozentual den Abstand berechnet, sodass alle Commit-Boxen auf die Zeitleiste passen.

5.3 Abnahme

Ziel war es, eine Generalisierung der Visualisierungsplattform zu erarbeiten und umzusetzen und die zentrale Fragestellung zu beantworten: Wie kann die Visualisierung flexibel auf verschiedene Schemata reagieren?

Wie vorgenommen wurde die Bindung an das eine Datenbankschema und die eine Programmiersprache aufgehoben. Diese werden lediglich noch als Parameter zur Konfiguration übergeben, um Webseite und Funktionen anpassen zu können.

Wie vorgesehen wurden alle statischen Aspekte der Tabelle 2 behandelt und aus vorangegangenem Kapitel zum Testing lässt sich die Vollständigkeit und Funktionalität des Systems entnehmen.

Die Antwort auf die Fragestellung, ist das entwickelte System selbst. Damit die Visualisierung flexibel auf verschiedene Schemata reagieren kann, müssen Relationship Types, Node Label und Property Keys aus der Datenbank variabel extrahiert, verarbeitet und visualisiert werden. Relationships wurden über die Nodes abgefragt, Node Label vom Benutzer zugeordnet und Property Keys nach bestimmten Annahmen passenden Eigenschaften zugeordnet.

Die Ziele und Aufgaben dieser Arbeit wurden erreicht und die Generalisierung erfolgreich umgesetzt, weshalb das System abgenommen wird.

6 Fazit

Dieses Kapitel liefert eine Zusammenfassung dieser Arbeit und einen Ausblick in die Zukunft. Es erfolgt ein Überblick über die vorangegangen Kapitel sowie ein Fazit über den Erfolg der Arbeit. Im Anschluss werden im Ausblick Erweiterungs- und Verbesserungsmöglichkeiten für die Umsetzung dieser Arbeit geliefert.

6.1 Zusammenfassung

Die Visualisierung von Softwarearchitektur in ihrer Evolution hilft, Verständnis für die vorliegende Software zu entwickeln, und kann als Kommunikationsgrundlage im Team dienen. Die Architektur hängt von der Programmiersprache ab, in der die Software entwickelt wurde, die somit bei jeder Sprache unterschiedlich sein kann. Aus diesem Grund wurde eine universelle Lösung zur Visualisierung verschiedener Architekturen gesucht.

Dafür wurden zuerst die Ausgangssituation und der Stand der Technik erläutert, anschließend wurde ein Konzept für eine solche Lösung erarbeitet, welches darauf hin umgesetzt und zuletzt evaluiert wurde.

Die Ausgangssituation für diese Arbeit stellte eine bestehende web- und graphdatenbankbasierte Visualisierungsplattform [2], die statisch an einen Datensatz und somit an seine Architektur gebunden entwickelt worden war.

In der Konzeption einer geeigneten universellen Lösung für die Extrahierung, Verarbeitung und Visualisierung von verschiedenen Datenbankschemata aus Neo4j wurden Ansätze verglichen, um den geeignetsten unter ihnen zu erfassen. Die Erkenntnis war die Umsetzung einer Kombination mehrerer Lösungsansätze. Außerdem wurden statische Aspekte in der Software definiert, um diese im Verlaufe der Arbeit beseitigen oder aufheben zu können.

Bei der prototypischen Umsetzung wurden alle zuvor definierten statischen Aspekte behandelt, sodass die Software überwiegend dynamisch oder automatisch reagiert. Es wurde unterteilt, in die Konfiguration durch den Benutzer und den Visualisierungsprozess. Nach dem der Benutzer die nötigen Angaben zum Datenbank-Login gemacht, die Programmiersprache gewählt und die Node Label aus der Datenbank den Ansichten zugeordnet hat, werden die Webseite und die Funktionen automatisch generiert und ausgeführt.

Die vorangegangene Antwort auf die zentrale Fragestellung, wie die Visualisierung flexibel auf verschiedene Schemata reagieren kann, lässt sich nach der Evaluation untermauern. Diese Umsetzung stellt eine geeignete Lösung für eine Generalisierung im Kontext dieser Arbeit dar, dennoch nicht für jede Architektur. Es müssen bestimmte Eingrenzungen und Vorgaben für die Verwendung der Visualisierungsplattform definiert werden.

Zum einen können nur komponentenbasierte Programmiersprachen eingebunden werden und zum anderen muss der Datensatz in der Graphdatenbank für erfolgreiche Ergebnisse bestimmte Vorraussetzungen erfüllen. Dabei ist es wichtig, dass die Benennung der Property Keys sinnvoll und in englischer Sprache erfolgt. Sinnvoll bedeutet hier, dass Zeitangaben das Wort „time“ und Namen das Wort „name“ enthalten. Außerdem sollte jeder Knoten die Unix-Zeitkomponente als Eigenschaft enthalten. An dieser Stelle ist allerdings anzumerken, dass die Software insofern optimiert werden könnte, damit dies nicht länger nötig ist. Dazu müsste stattdessen der Pfad des Graphen aus Neo4j abgefragt und so die zugehörige Zeit ermittelt werden. Des Weiteren muss die Programmiersprache in der XML-Datei gemäß des Formats der bisher integrierten eingebunden werden, um korrektes Auslesen zu gewährleisten.

6.2 Ausblick

Die in dieser Arbeit bereits genannten Erweiterungs- oder Optimierungsmöglichkeiten umfassen hauptsächlich die Verbesserung für den Nutzer, wie die Prüfung und Handlung im Falle eines erfolglosen Datenbank-Login, das Ersetzen des Project titles durch echte Projektinformation, die Optimierung der Zuordnung von Farben zu Knoten oder der Berechnung des Start- und Endpunktes der Zeitleiste.

Aufwändige, aber sinnvolle Erweiterungen für das System wären die Möglichkeit zum Erweitern auf über- und untergeordnete Knoten und die Visualisierung der Veränderung in der Evolution.

Die Erweiterung um über- und untergeordnete Knoten würde die Beziehungen zwischen Knoten verschiedener Ansichten aufzeigen. Im Neo4j Browser gibt es beim Anwählen eines Knoten die Option `Expand child relatonships`. Dadurch werden alle Knoten, die mit diesem durch eine direkte Kante verbunden sind, angezeigt. Der Vorteil ist hier das Wissen über den Kontext des Knotens und seine Verbindungen im System.

Die Visualisierung der Veränderungen in der Evolution der Software würde es dem Nutzer ermöglichen, die Entwicklung auf einen Blick zu sehen. Im aktuellen Stand des Systems müsste man die Knoten oder Kanten zählen und anschließend die Namen vergleichen, um die Veränderungen identifizieren zu können. Deshalb wäre es sinnvoll, einen Vergleich zwischen den Commits durchzuführen und darauf basierend neue Knoten oder Kanten farblich hervorzuheben sowie entfernte Knoten oder Kanten auszugrauen.

Des Weiteren könnte die Software insofern erweitert oder verbessert werden, dass die definierten Bedingungen und Begrenzungen des Systems aufgehoben werden.

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

[1] L. Merino, E. Kozlova, O. Nierstrasz, D. Weiskopf. *VISON: An Ontology-Based Approach for Software Visualization Tool Discoverability*. 2019.

[2] C. Rapp. *(unpublished) Praktikumsbericht - Visualisierung von Software-Evolutionsgraphen*. 2019.

[3] Carnegie Mellon University Software Engineering Institute. *Software Architecture*. URL: `https://www.sei.cmu.edu/research-capabilities/all-work/display.cfm?customel_datapageid_4050=21328`. (Zugriff: 19.09.2019).

[4] ISO/IEC/IEEE 42010. *Systems and software engineering - Architecture description*. 2011.

[5] H. Balzert. „Was ist eine Softwarearchitektur?“ In: *Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb* (2011).

[6] M.M. Lehmann. „Programs, life cycles, and laws of software evolution“. In: *Proceedings of the IEEE* (1980).

[7] D. Gračanin, K. Matković, M. Eltoweissy. „Software visualization“. In: *Innovations in Systems and Software Engineering* (2005).

[8] H. Schumann, W. Müller. *Visualisierung - Grundlagen und allgemeine Methoden*. 2000.

[9] Y. Ghanam, S. Carpendale. *A Survey Paper on Software Architecture Visualization*. 2008.

[10] T. Panas, R. Berrigan, J. Grundy. *A 3D Metaphor for Software Production Visualization*. 2003.

[11] M. Misiak, A. Schreiber, A. Fuhrmann, S. Zur, D. Seider, L. Nafeie. *IslandViz: A Tool for Visualizing Modular Software Systems in Virtual Reality*. 2018.

[12] H. Graham, H. Y. Yang, R. Berrigan. *A Solar System Metaphor for 3D Visualisation of Object Oriented Software Metrics*. 2004.

[13] DSi Digital. *Vizz*. URL: `https://vizz3d.com/`. (Zugriff: 31.08.2019).

[14] B. Sasaki, J. Chao, R. Howard. *Graph Databases for Beginners*. 2018.

[15] Neo4j. *What is a Graph Database?* URL: `https://neo4j.com/developer/graph-database/`. (Zugriff: 01.09.2019).

[16] Neo4j. *Neo4j Website*. URL: `https://neo4j.com/`. (Zugriff: 18.08.2019).

[17] Neo4j. *Cypher for Neo4j*. URL: `https://neo4j.com/developer/cypher-query-language/`. (Zugriff: 18.08.2019).

[18] Neo4j. *Neo4j JavaScript Driver 1.7.2*. URL: `https://neo4j.com/developer/javascript/`. (Zugriff: 23.03.2019).

[19] Visjs Community. *Vis.js*. URL: `https://visjs.org/`. (Zugriff: 18.08.2019).

[20] M. Knauß. *Nutzung von Software-Architekturvisualisierungen in der Praxis*. 2009.

Eidesstattliche Erklärung

Ich versichere, dass ich die vorliegende Abschlussarbeit selbstständig und ohne unerlaubte Hilfe Dritter verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die inhaltlich oder wörtlich aus Veröffentlichungen stammen, sind als solche kenntlich gemacht. Diese Arbeit lag in gleicher oder ähnlicher Weise noch keiner Prüfungsbehörde vor und wurde bisher nicht veröffentlicht.

Berlin, 20.09.2019 Cynthia Rapp