# Visualisierung der Evolution von Softwarearchitektur

Masterarbeit im Fach Informatik

vorgelegt von

**Lynn von Kurnatowski**

geb. 14.01.1990 in Berlin

angefertigt am

**Department Informatik**
**Lehrstuhl für Informatik 2**
**Programmiersysteme**
**Friedrich-Alexander-Universität Erlangen–Nürnberg**
**(Prof. Dr. M. Philippsen)**

in Kooperation mit

**Deutsches Zentrum für Luft- und Raumfahrt e.V.**
**Simulations- und Softwaretechnik**
**Intelligente und verteilte Systeme**
**(Andreas Schreiber)**

Betreuer: Prof. Dr. Michael Philippsen
Betreuer: Dr.-Ing. Martin Jung

Beginn der Arbeit: {*22.03.2018*}
Abgabe der Arbeit: {*date*}

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Universität Erlangen-Nürnberg, vertreten durch die Informatik 2 (Programmiersysteme), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Masterarbeit einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den {*date*}

Lynn von Kurnatowski

# Masterarbeit

**Thema:** Visualisierung der Evolution von Softwarearchitektur

**Hintergrund:** Die Architektur, die sich im Source-Code großer, langlebiger Softwaresysteme ausprägt weicht häufig von der ursprünglich geplanten oder dokumentierten Architektur ab. Um den aktuellen Status der Softwarearchitekturen aus dem vorliegenden Quellcode zu extrahieren, gibt es bereits Verfahren, die beispielsweise mit statischer Analyse arbeiten. Speziell bei Software, die auf einem reichhaltigeren Komponenten-Framework (z.B. OSGi) aufbaut, lassen sich über die Analyse von Metainformationen noch weitere Informationen über die Struktur der Software ableiten. Bestehende Verfahren lassen allerdings den entscheidenden Aspekt der Evolution, also der Änderung der Architektur über die Zeit, unberücksichtigt.

**Aufgabenstellung:** Basierend auf dem Stand der Technik in der 2D- und 3D-Visualisierung von Softwarearchitekturen sind Sichten und Darstellungsformen auszuwählen bzw. zu definieren, die Aufschluss auf die zeitliche Veränderung der Softwarearchitektur geben. Um die Visualisierung der zeitlichen Veränderung einer Softwarearchitektur effizient zu gestalten, sind im zweiten Teil der Arbeit Kriterien festzulegen, wann eine Änderung im Quellcode-Repository als relevant für die Architektur zu bezeichnen ist. Im dritten Teil der Arbeit ist das Zusammenstellen der für die Visualisierung notwendigen Daten für ein OSGi-Softwaresystem zu konzipieren und aus den aufbereiteten Daten sind die Sichten zu generieren, die in Teil 1 der Arbeit festgelegt wurden. Basierend auf diesen Daten ist die Wirksamkeit der Effizienzverbesserungen aus dem zweiten Teil der Arbeit zu belegen.

**Meilensteine:**

- Zusammenfassen des Stands der Technik in der Visualisierung von Softwarearchitekturen
- Festlegen der Sichten für Evolution
- Festlegen der Kriterien für „architekturrelevante Änderungen“
- Generieren der Sichten
- Validierung der Ergebnisse
- Endvortrag im Kolloquium
- Anfertigung der Ausarbeitung

**Betreuung:** Prof. Dr. Michael Philippsen
Dr.-Ing. Martin Jung

**Bearbeiter:** Lynn von Kurnatowski

# Abstract

Softwarearchitektur wird klassischerweise mithilfe von UML-Diagrammen zum Beginn eines Projektes dokumentiert. Doch aufgrund neu formulierter Anforderungen durchläuft die Software während des gesamten Lebenszyklus eine Evolution, in der auch kontinuierlich eine Anpassung der Architektur erfolgt. Es besteht also keine Übereinstimmung mehr zwischen der anfänglich dokumentierten Softwarearchitektur und der implementierten. Somit sind die anfänglich erstellten UML-Diagramme besonders bei langjährigen und komponentenreichen Softwaresystemen schnell überholt und stellen keine ideale Lösung zur Kommunikation dar.
In dieser Arbeit wird ein Konzept vorgestellt, mit dem die gesamte Evolution komplexer Softwarearchitekturen verständlich dargestellt werden kann. Unter anderem wird die Frage beantwortet, wie Softwarevisualisierungen während des Entwicklungsprozesses helfen können und welche Vorteile sich daraus für Softwarearchitekten und Entwickler ergeben. Es wird gezeigt wie alle relevanten Daten für die Visualisierung durch Repository Mining auf dem gesamten Quellbaum und Data Mining auf Quellcodeebene aufbereitet werden. Durch die Implementierung der Softwarevisualisierung wird verdeutlicht wie die zeitliche Veränderung der Architektur greifbar gemacht und als Kommunikationsmittel eingesetzt werden kann, um technische Probleme zu identifizieren oder Qualitätsmerkmale zu beurteilen. Denn es ist von großer Wichtigkeit die gesamte Evolution zu betrachten und nicht nur einen Schnappschuss, um in der Lage zu sein, komplexe Systeme zu verstehen.

# Inhaltsverzeichnis


**1 Einleitung - The What and Why** 1

1.1 Arbeitsumgebung . . . 1
1.1.1 Deutsches Zentrum für Luft- und Raumfahrt e.V. . . . 1
1.1.2 Einrichtung Simulations- und Softwaretechnik . . . 1
1.2 Motivation . . . 2
1.3 Ziel der Arbeit . . . 3
1.4 Gliederung der Arbeit . . . 4

**2 Grundlagen - Things to Know in Advance** 5

2.1 Softwareanalyse . . . 5
2.1.1 Architekturanalyse . . . 5
2.2 Softwarearchitektur . . . 7
2.2.1 Evolution von Software . . . 9
2.3 Refactoring . . . 9
2.4 Java . . . 10
2.4.1 Struktur . . . 10
2.5 OSGi . . . 10
2.5.1 Modularität . . . 11
2.5.2 Framework . . . 12
2.6 Software-Measurement . . . 16
2.6.1 Metriken . . . 16
2.7 Graphdatenbanken . . . 19
2.7.1 Neo4j . . . 21

**3 Softwarevisualisierung - The Visualization Basics** 23

3.1 Datenvisualisierung . . . 23
3.1.1 Interaktionsmethoden . . . 24
3.2 Definition und Ziele . . . 25
3.2.1 Visualisierungs-Pipeline . . . 26
3.3 Klassifizierung . . . 26
3.3.1 Aspekte von Software . . . 26

3.3.2 Granularität ... 27
3.3.3 Anzahl der Sichten ... 27
3.3.4 Dimensionen ... 28
3.3.5 Virtual Reality ... 28
3.4 Herausforderungen ... 29
3.5 Metapher ... 30
3.5.1 Visualisierung der Architektur von Software ... 30
3.5.2 Visualisierung der Evolution von Software ... 34
3.6 Vorhandene Visualisierungstools ... 36

**4 Konzept - From the Model to the Visualization** 39
4.1 Zielgruppe ... 39
4.1.1 Nutzungsszenarien ... 40
4.2 Datenbankmodell ... 40
4.3 Visualisierungsansatz ... 45
4.3.1 Visuelle Metapher ... 51
4.4 Implementierungsansatz ... 55

**5 Umsetzung - Visualize the architecture** 57
5.1 Extraktion ... 57
5.1.1 Aufbau ... 57
5.1.2 Konfiguration ... 58
5.1.3 Persistierung ... 58
5.2 Architekturrelevante Softwareänderungen ... 61
5.2.1 Identifizierung architekturrelevanter Änderungen ... 62
5.2.2 Speicherung architekturrelevanter Änderungen ... 66
5.2.3 Mögliche Erweiterungen ... 68
5.3 Visualisierung ... 69
5.3.1 Plattform ... 69
5.3.2 Herausforderung ... 70
5.3.3 KeyLines ... 73
5.3.4 Mögliche Erweiterungen ... 79

**6 Evaluation - Proof of Concept** 81
6.1 Remote Component Environment ... 81
6.2 Modellerstellung ... 82
6.3 Evaluation der Visualisierung ... 83

**7 Fazit - To be continued... 87**

7.1 Zusammenfassung . . . . . 87

7.2 Ausblick . . . . . 88

# Abbildungsverzeichnis


2.1 Worfklow einer Architekturanalyse [17] . . . . . 6
2.2 Soll-Architektur ungleich der Ist-Architektur . . . . . 8
2.3 Bundles und Services im OSGi-Framework [56] . . . . . 14
2.4 Aufbau des OSGi-Frameworks [1] . . . . . 15
2.5 Abfrage von Beziehungen mit Hilfe von JOIN-Klauseln . . . . . 20
2.6 Reaktionszeit relationale Datenbank versus Graphdatenbank [53] . . . . . 21

3.1 Verhältnis zwischen Code-Lesen und Code-Schreiben . . . . . 25
3.2 (a) Beispiel einer 2D-Visualisierung [9] (b) Beispiel einer 3D-Visualisierung [57] . . . . . 29
3.3 Mapping zwischen der zu beantwortenden Problemstellung und der umgesetzten Visualisierung [40] . . . . . 31
3.4 (a) Baummodell [9] (b) Treemap [9] . . . . . 32
3.5 (a) Sunburst-Visualisierung 2D [9] (b) Sunburst-Visualisierung 3D [9] . . 32
3.6 (a) Geon-Diagramms [30] (b) Rufdiagramm [26] . . . . . 33
3.7 (a) City-Metapher [9] (b) Sonnensystem [43] . . . . . 34
3.8 (a) Gegenüberstellung zweier Versionen einer Architektur [27] (b) Matrixdarstellung [34] . . . . . 35
3.9 Anzahl der Paper nach Typ sortiert [40] . . . . . 36
3.10 Visualisierung mit Gource . . . . . 37

4.1 Datenbankmodell . . . . . 42
4.2 Modellierung der Historie im Datenbankmodell . . . . . 43
4.3 Modellierung von Packages und Package Fragmenten . . . . . 44
4.4 Modellierung der Services . . . . . 45
4.5 Verschiedene Ebenen des Interesses an Software Visualisierung - Quelle suchen . . . . . 49
4.6 Ausgewählte Darstellungsart für Bundles . . . . . 52
4.7 Ausgewählte Darstellungsart für Service-Components und Services . . . . 54
4.8 grober Ablaufplan der Applikation . . . . . 55

5.1 Workflow . . . . . 59
5.2 Aufbau der Neo4j-OGM Library und Spring Data Neo4j Library . . . . . 61
5.3 Beispielhafter Datenbankausschnitt mit Informationsverlust . . . . . 67
5.4 Beispielhafter Datenbankausschnitt ohne Informationsverlust . . . . . 67
5.5 Beispielhafter Datenbankausschnitt architekturrelevanter Änderungen . . 68

5.6 Darstellung eines Graphen mit D3.js . . . . . 71
5.7 Ausgewählte Darstellungsart für Bundles . . . . . 73
5.8 Ausgewählte Darstellungsart für Bundles . . . . . 73
5.9 Darstellung der Architektur [8] . . . . . 74
5.10 Darstellung der gesamten Architektur . . . . . 75
5.11 Darstellung des Bundle-Graphen . . . . . 76
5.12 Darstellung des Service-Graphen . . . . . 77
5.13 Darstellung der Query-Funktion . . . . . 77
5.14 Darstellung der Zoomleiste in der Visualisierung . . . . . 78
5.15 Darstellung der Zeitleiste . . . . . 79

6.1 Screenshot einer RCE-Instanz (Workflow zur Optimierung des Thermal Management Systems eines SpaceLiners) [12] . . . . . 82
6.2 Use Case - Übersicht über die zeitliche Veränderung erhalten . . . . . 84
6.3 Use Case - Überblick über die gesamte Architektur bekommen . . . . . 85

# Tabellenverzeichnis


2.1 typische Sichten auf die Software Architektur . . . . . 8
2.2 Klassifikation von Softwaremetriken (nach Fenton [18]) . . . . . 17
2.3 Bewertung der Kopplungsarten zwischen Bundles nach Beck und Diehl [4] 19

48table.41 Entscheidungsmatrix, wenn keine Architekturänderungen vorliegen 63
5.2 Entscheidungstabelle, wenn keine Architekturänderungen vorliegen trotzdem sich eine Komponente scheinbar verändert hat . . . . . 64
5.3 Entscheidungstabelle, wenn eine Komponenten gelöscht oder hinzugefügt wurden . . . . . 64
5.4 Entscheidungstabelle, wenn Architekturänderungen vorliegen . . . . . 65
5.5 Entscheidungstabelle, wenn Architekturänderungen vorliegen . . . . . 65
5.6 Vor- und Nachteile einer eigenständig entwickelten Anwendung . . . . . 69
5.7 Vor- und Nachteile einer webbbasierten Anwendung im Browser . . . . . 69

6.1 Zeiten für die Speicherung der gesamten Architektur für jeden Commit . 83
6.2 Zeiten für die Speicherung der Architekturänderungen . . . . . 83

# 1 Einleitung - The What and Why

Dieses Kapitel beginnt mit einer kurzen Beschreibung der Arbeitsumgebung, legt die Motivation der Arbeit dar und erläutert das Ziel. Abschließend wird der Aufbau der Arbeit beschrieben.

## 1.1 Arbeitsumgebung

Diese Arbeit wurde in Kooperation mit dem Deutschen Zentrum für Luft- und Raumfahrt e.V. in der Abteilung Intelligente und verteilte Systeme der Einrichtung Simulations- und Softwaretechnik verfasst.

### 1.1.1 Deutsches Zentrum für Luft- und Raumfahrt e.V.

Das Deutsche Zentrum für Luft und Raumfahrt e.V. (DLR) ist das Forschungszentrum in Deutschland für Luft- und Raumfahrt. Es werden Forschungs- und Entwicklungsarbeiten in den Bereichen Raumfahrt, Luftfahrt, Energie, Verkehr, Sicherheit und Digitalisierung an 40 Instituten und Einrichtungen an 20 Standorten durchgeführt.
Die Mission des DLR ist die Erforschung der Erde und des Sonnensystems, die Entwicklung umweltfreundlicher Technologien für die Energieversorgung, Mobilität, Kommunikation und Sicherheit. Um diese Aufgaben zu bewerkstelligen, folgt sie verschiedenen Richtlinien wie „One DLR“, Exzellenz in Wissenschaft und Professionalität, Aufmerksamkeit für Präzision und Zuverlässigkeit und vieles mehr. Des weiteren ist das DLR der größte Softwarehersteller in Deutschland.

### 1.1.2 Einrichtung Simulations- und Softwaretechnik

Zu den Aufgaben der Einrichtung Simulations- und Softwaretechnik gehört die Forschung und Entwicklung auf dem Gebiet innovativer Software-Engineering-Technologien und die Bereitstellung und Anwendung dieses Wissens.
Zu den derzeitigen Themenschwerpunkten gehören die Softwareentwicklung für verteilte und mobile Systeme, Software für eingebettete Systeme, Visualisierung und High Performance Computing. Durch die stetige Zusammenarbeit mit anderen Instituten im DLR beteiligt sich die Einrichtung an anspruchsvollen Softwareentwicklungsprojekten und lässt somit das gewonnene Know-How mit in die Projekte einfließen.
Die Einrichtung ist an den Standorten Köln, Braunschweig und Berlin vertreten und

gliedert sich in die Abteilungen „Intelligente und verteilte Systeme“, „High Performance Computing“ und „Software für Raumfahrtsysteme und interaktive Visualisierung“.
Die vorliegende Masterarbeit „Visualisierung der Evolution von Softwarearchitektur“ wurde in der Abteilung Intelligente und verteilte Systeme erarbeitet. In der Abteilung wird unter anderem das OSGi-basierende Softwareprogramm *Remote Component Environment* entwickelt.
Remote Component Environment, kurz RCE, ist ein Open Source Framework, welches Ingenieuren und Wissenschaftlern beim Design und der Simulation komplexer Systeme, durch die Verwendung und Integration ihrer eigenen Design- und Simulationswerkzeuge, unterstützt [15]. RCE soll dieser Arbeit als Anwendungsbeispiel dienen.

## 1.2 Motivation

„The data is out there. The problem is making practical use of it.“ [6]

Software-Systeme nehmen immer mehr an Komplexität zu. Deshalb ist, wie auch in den Ingenieursdisziplinen, für die Softwareentwicklung eine Form von „Bauplan“ notwendig, der die wesentlichen Eigenschaften eines Systems darstellt. Die Softwarearchitektur wird oft als solch ein Bauplan eines Software-Systems gesehen und dient als wesentliche Beschreibung um den Aufbau und das Verhalten eines Systems zu verstehen. Gerade für große und komplexe Softwareprojekte ist die Softwarearchitektur als Kommunikationsvehikel wichtig, als Basis für ein wechselseitiges Verstehen zwischen allen beteiligten Personen, zur Festlegung von Entscheidungen sowie zur Kommunikation über das System.
Doch der ursprüngliche Bauplan kann im Laufe der Zeit nicht mehr zur Kommunikation und Verständigung über das entwickelnde System eingesetzt werden. Denn während des gesamten Lebenszyklus durchläuft die Software eine Evolution, in der aufgrund neu formulierter Anforderungen auch eine Anpassung der Architektur erfolgen müsste. Infolgedessen weicht die Architektur, die sich im Source Code über den gesamten Entwicklungsprozess hinweg ausprägt häufig von der initial geplanten und dokumentierten Architektur ab. Ein Verständnis von den bisherigen und aktuell bestehenden Strukturen zu erhalten ist somit ein schwieriges Unterfangen, da keine Übereinstimmung zwischen dem Modell und der Implementierung vorhanden ist.
Um dieser Problematik entgegenzuwirken ist es von Vorteil den Quellcode, die aktuellste und detaillierteste Quelle eines Softwareprojektes, hinzuzuziehen und die fehlende Übereinstimmung zu revidieren. Jedoch eignet sich Quellcode bei der Betrachtung und Analyse nur im begrenztem Maße als Darstellungsart, um ein Überblick über die Architektur und den Aufbau des Systems zu erlangen. Denn das menschliche Gehirn ist nicht dafür ausgelegt, effektiv hunderttausend Zeilen verschränkten Codes zu verstehen.
Geeignete Visualisierungen sind leistungsstark, wenn Sie große Datenmengen erfassen müssen. Unser menschliches Gehirn ist eine erstaunliche Muster-Matching-Maschine,

die eine erstaunliche Menge an visuellen Informationen verarbeitet kann.
Doch die wenigen bisher bestehenden Visualisierungsmethoden lassen den Aspekt der Evolution, also der Änderung der Architektur über die Zeit, unberücksichtigt. Dabei enthält der Quellcode im Repository alle wesentlichen Informationen über die zeitliche Veränderung. Jede Änderung am System, die jemals durchgeführt wurde, wurde aufgezeichnet.
Dabei ist es von großer Wichtigkeit die gesamte Evolution zu betrachten, denn der Mensch wird niemals in der Lage sein, komplexe Systeme zu verstehen, wenn er sich nur einen Schnappschuss der Software betrachtet. Durch eine entsprechende Visualisierung der Evolution einer Architektur lassen sich, neben der Dokumentation, Schlussfolgerungen aus dem bisherigen Entwicklungsprozess ziehen, mit deren Hilfe sich die weiteren und zukünftigen Entwicklungsarbeiten optimieren lassen.

## 1.3 Ziel der Arbeit

Ziel dieser Arbeit ist es, die zeitliche Veränderung von Architekturen einer OSGi-basierenden Java-Software zu visualisieren, um ein Verständnis des Projektes zu vermitteln und somit einen Mehrwert für den Entwicklungsprozess zu schaffen.
Das reichhaltige Komponenten-Framework OSGi beinhaltet besonders viele Aspekte, die bei der Betrachtung der Architektur eine wichtige Rolle spielen. Dafür muss zunächst definiert werden, welche Aspekte bei der Betrachtung einer Architektur von Interesse sind. Um eine geeignete Auswahl der Granularitätsebene zu finden, muss die zuvor festgelegte Zielgruppe und die Nutzungsszenarien berücksichtigt werden.
Da in dieser Arbeit vorrangig große und komplexe Softwareprojekte betrachtet werden, entstehen bei der Extraktion der Historie einer Architektur aus dem Repository eine große Menge an Daten, in Form von Komponenten und Beziehungen in allen Revisionen. Um die Visualisierung dieser Daten effizient zu gestalten, sollen im ersten Teil dieser Arbeit diese Daten auf die Wesentlichen beschränkt werden. Dafür muss zunächst der Begriff Softwarearchitektur definiert werden und Kriterien dafür gefunden werden, wann eine Änderung als relevant für die Architektur betrachtet werden kann. Anschließend muss auf Grundlage der Extraktion, das heißt auf Grundlage der Datenbank, ein Vorgehen entwickelt werden, diese Kriterien entsprechend anzuwenden. Somit soll die Datenmenge in der Datenbank auf die wichtigsten Bestandteile beschränkt, und die Visualisierung der zeitlichen Veränderung der Architektur effizienter gestaltet werden.
Im zweiten Teil der Arbeit sollen diese Daten visuell dargestellt werden. Die Ausarbeitung der Konzeption der Visualisierung stellt einen entscheidenden Teil dar. Darstellungsarten sind auszuwählen beziehungsweise zu definieren, für die sich die Visualisierung der Evolution einer Softwarearchitektur eignet. Besonders die Visualisierung des Zeitverlaufs muss geschickt in die Darstellung integriert werden und die architekturrelevanten Änderungen hervorgehoben werden. Des Weiteren ist zu klären, inwiefern Interaktionsmöglichkeiten, wie zum Beispiel das Filtern von Informationen oder das Her-

vorheben und Selektieren von bestimmten Elementen, für den Betrachter von Interesse beziehungsweise notwendig sein könnten.
Bei der Implementierung sollen die im Konzept ausgearbeiteten Darstellungsarten und Interaktionsmethoden umgesetzt werden. Dazu gehört zunächst die Beantwortung technischer Fragen. Es muss geklärt werden in welcher Umgebung die Visualisierung laufen soll und wie die Schnittstelle zwischen der Analyse und der Visualisierung aussehen soll.

## 1.4 Gliederung der Arbeit

In Kapitel 2 werden die für diese Arbeit relevanten theoretischen Grundlagen beschrieben, um einen einheitlichen Konsens der Begriffswelt zu schaffen. In Kapitel 3 werden Grundlagen der Softwarevisualisierung und der Stand der Technik in diesem Forschungsbereich erläutert. Basierend auf den Grundlagen wird im vierten Kapitel ein Konzept für die Analyse und Visualisierung von OSGi-Anwendungen vorgestellt. In Kapitel 5 wird die Umsetzung des Konzepts, gegliedert in drei Teilen (Extraktion, Analyse und Visualisierung), vorgestellt. Anhand des Anwendungsbeispieles RCE, welches im Kapitel 1.1.2 kurz beschrieben wurde, wird in Kapitel 6 die Evaluation der Implementierung beschrieben, welche durch eine Effizienzbetrachtung der Analyse und anhand von User Stories durchgeführt wurde. Zum Schluss wird in Kapitel 7 ein Fazit der Arbeit gezogen und ein Ausblick auf mögliche Erweiterungen gegeben.

# 2 Grundlagen - Things to Know in Advance

In diesem Kapitel werden für die Arbeit relevante theoretische Grundlagen vorgestellt und Begriffe definiert, damit später beim Entwurf des Konzeptes und der Umsetzung von einer einheitlichen Grundlage und Begriffswelt ausgegangen werden kann.
Zunächst soll ein Einblick in die Architekturanalyse und der eingesetzten Technik, Repository Mining, gegeben werden. Im Hinblick auf die zu erstellenden Kriterien architekturrelevanter Änderungen soll eine Einführung in die Softwarearchitektur im allgemeinen und das Konzept und die Eigenschaften der Programmiersprache Java und des OSGi-Frameworks gegeben werden.

## 2.1 Softwareanalyse

Die Analyse von Daten, welche aus dem Softwareentwicklungsprozess hervorgehen, stellen einen erheblichen Reiz für Forschung und Entwicklung dar. Vor, während und nach dem Entwicklungsprozess eines Softwareprojektes werden Gigabytes an Daten produziert. Im Bereich der Softwareanalyse wird immer intensiver an dem Thema gearbeitet, diese Daten sowohl nutzbar zu machen als auch für Analysezwecke zu verwenden, mit der Zielsetzung den allgemeinen Entwicklungsprozess von Softwaresystemen zu optimieren [6].
Eine allgemeingültige Definition für den Begriff Softwareanalyse aufzustellen, bedeutet ebenso wie die Arbeit an dem Thema eine Herausforderung. Die Softwareentwicklung ist ein sehr komplexer Prozess mit einer Vielzahl an Zuständigkeiten und beteiligten Personen. Für jedes Team ergeben sich andere domänenspezifische Anforderungen an die Softwareanalyse.
In dieser Arbeit liegt der Schwerpunkt auf der Architekturanalyse, welche mithilfe der aus dem Softwareentwicklungsprozess entstandenen Daten umgesetzt wird.

### 2.1.1 Architekturanalyse

Im Softwareentwicklungsprozess ist der Architekturentwurf die früheste Softwaredesign-Entscheidung. Er dient der Kommunikation, Dokumentation und der Verständigung über das zu entwickelnde Programm. Doch im Laufe der weiteren Entwicklung findet häufig keine Überprüfung bezüglich der Übereinstimmung zwischen dem Modell mit

der tatsächlichen Implementierung statt, was fatale Folgen nach sich ziehen kann. Ein falsches Modell kann einem Entwickler ein falsches Verständnis der Ursachen des Programmverhaltens vermitteln. Das Fehlen einer Übereinstimmung kann dazu führen, dass das Modell mit der Zeit immer unbrauchbarer beziehungsweise irreführend wird, bis der Entwickler sich nicht mehr auf die Aussage des Modells verlassen kann [17]. Daher kann die Architekturanalyse ein wichtiges Werkzeug im Entwicklungsprozess sein.
Die Architekturanalyse besteht aus drei ineinandergreifende Vorgänge, siehe Abbildung 2.2:

**Extraktion** Wesentliche Strukturinformationen werden aus den Quellcode- und Dokumentationsdateien extrahiert und in die richtigen Beziehungen gesetzt.

**Abstraktion** Informationen werden durch die Kombination und Transformation der gegebenen Beziehungen abgeleitet.

**Präsentation** Die Komponenten und Beziehungen werden grafisch dargestellt.

Zu beachten ist, dass der Vorgang der Extraktion, Abstraktion und Präsentation projekt- und sprachspezifisch ist. Konkret bedeutet dies, dass sie von der im Projekt verwendeten Programmiersprache und den jeweiligen Architekturregeln abhängig sind.

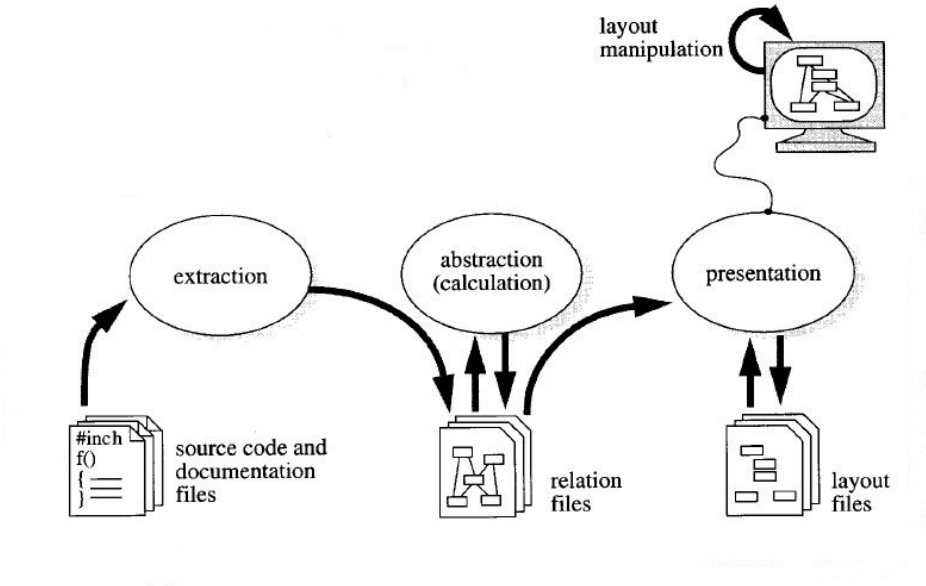



Abbildung 2.1: Worfklow einer Architekturanalyse [17]

Zusammenfassend lässt sich sagen, dass mithilfe einer Architekturanalyse eines gesamten Repository, eine lückenlose und stets aktuelle Dokumentation der Architektur über den gesamten Entwicklungszeitraum gewährleistet werden kann. Dafür werden die benötigten Daten mithilfe von Repository Mining extrahiert.

### Mining Software Repository

Repository Mining ist die empirische und systematische Untersuchung von Software-Repositories. Das Ziel besteht darin, Daten durch Codeanalyse zu erhalten und unentdeckte Informationen, Beziehungen oder Trends im zu analysierenden Repository zu identifizieren. Das Untersuchungsartefakt sind die Depots, welche die Historie und die vollständige Änderungsnachverfolgung der jeweiligen Projekte enthalten. Daher ist mithilfe von Repository Mining eine gründliche Analyse des Quellcodes und seiner Metadaten möglich.
Grundvoraussetzung für Repository Mining ist jedoch der Umstand, dass heutzutage für die kollaborative Zusammenarbeit in Softwareprojekten Versionsverwaltungssysteme, wie zum Beispiel Git, vermehrt zum Einsatz kommen. Womit diese Depots mit den nötigen Daten gefüllt werden, welche für den Prozess von Nöten sind.

## 2.2 Softwarearchitektur

Wie zu Beginn der Arbeit schon erläutert liegt das Hauptaugenmerk bei der Analyse auf der Architektur. Da es mehr als 50 verschiedene Definitionen von Softwarearchitektur gibt, die jeweils bestimmte Aspekte von Architekturen hervorheben, soll im folgenden eine Definition festgelegt werden, an der sich die weitere Arbeit orientiert:

Das Gebiet der Softwarearchitektur beschäftigt sich vordergründig mit der Strukturierung von Softwaresystemen. Entsprechend dem IEEE-Standard 1471-2000 wird Softwarearchitektur folgendermaßen definiert [29]:

> „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.“

Die Softwarearchitektur ist laut dem IEEE-Standard die grundlegende Organisation eines Systems. Dabei können zwei Aspekte einer Architektur betrachtet werden: zum einen die Beschreibung der Dynamik, das heißt der Verlauf des Kontrollflusses wird beschrieben und zum anderen die Abbildung der Komponenten, was Thema dieser Arbeit ist - die Komponenten und die Verbindungen zwischen diesen. Eine gute Softwarearchitektur hat einen großen Einfluss auf den Erfolg eines Softwareprojektes. Es ergibt sich die Frage, wie große Systeme so strukturiert werden können, dass letztendlich alle Anforderungen und die gewünschte Qualität erreicht werden. Je größer, umfangreicher, langwieriger und komplexer ein Softwareprojekt ist, desto wichtiger ist die Softwarearchitektur [55].
Zusätzlich kann die Softwarearchitektur in mehrere Sichten unterteilt werden, beispielweise Conceptual Architecture View, Module View, Execution View und Code View [25].

| Sichten | Beschreibung |
|---|---|
| Conceptual Architecture View | dient als High-Level-Karte, die anzeigt wie das System das tut, was es machen soll |
| Module View | zeigt, wie Module Maschinen und Netzwerke zugeordnet sind |
| Execution View | zeigt, wie die Software in Module und Subsysteme gegliedert ist |
| Code View | zeigt, wie Quellcode und Konfigurationen in Paketen organisiert sind, einschließlich deren Abhängigkeiten |

Tabelle 2.1: typische Sichten auf die Software Architektur

Die Tabelle 2.1 dient als Übersicht. In dieser Arbeit werden die letzten drei aufgelisteten Sichten aus der Tabelle nicht berücksichtigt. Ausschließlich die Conceptual Architecture View soll Bestandteil dieser Arbeit sein, Komponenten und deren Abhängigkeiten und Beziehungen.

### Soll- und Ist-Architektur

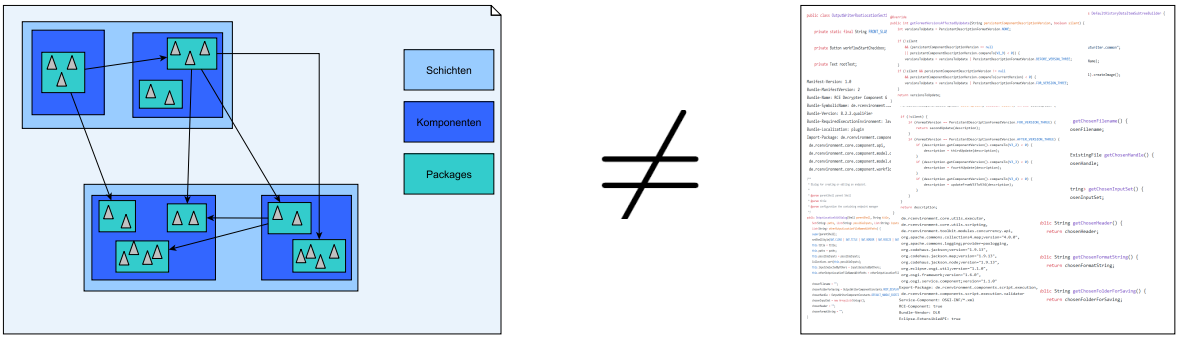



Abbildung 2.2: Soll-Architektur ungleich der Ist-Architektur

Wie im Kapitel 2.1.1 schon angedeutet, weicht im Laufe eines Entwicklungsprojektes häufig die Ist-Architektur von der Soll-Architektur ab.
Dabei ist die Soll-Architektur die vom Entwicklungsteam geplante Architektur und die im Sourcecode implementierte Architektur die Ist-Architektur.

Folgende Faktoren können das Auseinanderlaufen der Ist-Architektur und der Soll-Architektur beeinflussen [51]:

1. Funktionale und qualitative Ziele
2. Organisatorische Bedingungen
3. Technische Rahmenbedingungen

### 2.2.1 Evolution von Software

> „All successful software gets changed.“ [7]

Die ständige Veränderung spielt auch eine entscheidende Rolle bei der erfolgreichen Entwicklung von Software. Ein Softwaresystem muss sich fortwährend an die veränderten Anforderungen anpassen. Meir M. Lehman definierte die Gesetzte „Laws of Program Evolution“, die diese Behauptung untermalen [35]. Die für diese Arbeit wichtigsten Gesetzte werden im Folgenden aufgelistet und beschrieben:

**Law of Continuing Change** Ein Softwaresystem wird kontinuierlich während des gesamten Lebenszyklus weiterentwickelt und verändert. Bis es von einem anderen System abgelöst wird.

**Law of Increasing Complexity** Durch die ständige Veränderung, dem ein Softwaresystem unterliegt, nimmt es an Komplexität zu.

Aus den beiden Gesetzen lässt sich ableiten, dass Größe, Funktionaliät und Komplexität mit der Zeit in einem Softwaresystem zunehmen, während die Qualität abnimmt. Die im Laufe des Entwicklungsprozess zunehmende Größe und Komplexität, also die stetige Veränderung der Ist-Architektur, soll im zweiten Teil der Arbeit durch die Visualisierung der Evolution auf einer höheren Abstraktionsebene dem Betrachter ersichtlicher gemacht werden.

## 2.3 Refactoring

In den Laws of Program Evolution wird die Aussage getroffen, dass sich ein Sofwtaresystem fortwährend verändert und dadurch an Komplexität zunimmt. Daher ist es dringend erforderlich, die zunehmende Komplexität einer Software zu reduzieren, indem die interne Qualität verbessert wird. Den Prozess des Refactoring beschrieb William Opdyke folgendermaßen [39]:

> „The process of changing a software system in such a way that it does not alter the external behaviour of the code, yet improves its internate structure.“

Im Bezug zu der Evolution einer Software, siehe Abschnitt 2.2.1, werden Refactorings zum Verbessern der Qualität der Software eingesetzt.
Martin Fowler hat in seinem Buch Refactoring. Improving the design of existing code. [21] einen Katalog von verschiedenen Refactoringarten zusammengestellt. Da in dieser Arbeit ausschließlich die Architektur einer Software betrachtet wird, sind nur eine begrenzte Anzahl an Refactorings bei der weiteren Architekturanalyse relevant. Auf diese wird bei der Betrachtung der architekturrelevanten Änderungen (Kapitel 5.2) genauer eingegangen.

## 2.4 Java

Java ist eine statisch typisierte, objektorientierte Programmiersprache.
Der Java-Quellcode wird in maschinenlesbaren Bytecode kompiliert. Zur Laufzeit wird der Bytecode von der Java Virtual Machine (JVM) ausgeführt [3]. Das bedeutet, Java-Anwendungen können auf jedem beliebigen Gerät geschrieben, in Bytecode kompiliert und auf jedem beliebigen Gerät ausgeführt werden. Einzige Voraussetzung ist, auf dem Zielsystem muss eine JVM vorhanden sein. Somit ist Java eine plattformunabhängige Programmiersprache und kann auf verschiedener Hardware und unter verschiedenen Betriebssystemen ausgeführt werden.

### 2.4.1 Struktur

Software-Programme werden in Java als eine Menge von Packages organisiert. Die Gliederung in Packages dient als zentrales Konzept für die Modularisierung. Dies bietet die Möglichkeit, verwandte Klassentypen zusammenzufassen oder auch Sub-Packages anderen Packages unterzuordnen. Packages stellen unter anderem einen hierarchischen Namesraum für Klassen bereit und ordnen die Klassen wie Dateien in Verzeichnissen. Das bietet den Vorteil, dass Projekte besser strukturiert, Klassen durch die Verwendung von Namesräumen systematisch geordnet werden und deren Sichtbarkeit eingeschränkt werden kann.
Der Source Code wird in Java in Dateien mit der Endung .java gespeichert und entspricht einer Compilation Unit. Dies ist eine Einheit, die der Java-Compiler als Eingabe verarbeitet. Eine Compilation Unit enthält immer eine Typ-Deklaration, die eine Klasse definiert. Eine Klasse wiederum kann beliebig viele weitere Typ-Deklarationen (auch Nested Classes) enthalten.

## 2.5 OSGi

In dem Kapitel 2.4.1 wurde erwähnt, dass Packages in Java ein zentrales Konzept für die Modularisierung von Code darstellen. Doch ist diese Art der Modularisierung gerade für große und lang andauernde Projekte oftmals zu schwach.

Daher wurde mit der OSGi Spezifikation die führende Modularitätslösung für Java-Anwendungen entwickelt [37]. OSGi ist die Spezifikation eines Java-Frameworks, dass zur Erstellung und Ausführung dynamischer, modularer Systeme benutzt werden kann und ein komponenten- und serviceorientiertes Entwicklungsmodell unterstützt. Bei der Spezifikation handelt es sich um ein modulares Laufzeitsystem und dient als Laufzeitumgebung für Softwarekomponenten [56]. Java-Applikationen werden in Module unterteilt, wobei das Framework den Lebenszyklus dieser Module verwaltet und deren Abhängigkeiten. Zudem stellt es Mittel zum Suchen und Veröffentlichen von Services zur Verfügung.
Es gibt diverse Implementierungen der OSGi-Spezifikation, Laufzeitumgebungen, die als OSGi-Container bezeichnet werden. Zu den bekanntesten gehören: *Apache Felix* [1], *Knopflerfish* [2] und *Equinox* [3], welcher unter anderem als Grundlage für die Eclipse IDE dient.

Wie in diesem Abschnitt schon erwähnt, ist die OSGi Spezifikation eine Modularitätslösung für Java-Anwendungen. Deshalb soll im folgenden Kapitel der Begriff Modularität, im Kontext OSGi-basierender Java-Anwendungen, genauer erläutert werden.

### 2.5.1 Modularität

Modularität und folgende drei Prinzipien stehen in direktem Zusammenhang:

- Entkopplung
- Einfachheit
- Erweiterbarkeit

Diese drei nicht-funktionalen Eigenschaften spielen bei der Umsetzung modularer Systeme eine entscheidende Rolle.
Um eine stetige Weiterentwicklung ohne Gefährdung und Beeinflussung des Gesamtsystems zu ermöglichen, muss das System modular aufgebaut sein. Besonders bei der Entwicklung von großen und lang andauernden Projekten ist es entscheidend, die Prinzipien der Entkopplung und der Erweiterbarkeit zu berücksichtigen. Dadurch steht das Prinzip der Einfachheit oftmals in Kontrast zu den beiden anderen genannten Prinzipien. Oft müssen zusätzliche Abstraktionsschichten eingeführt werden, um ein System entkoppelt und erweiterbar umzusetzen. Daher ist es umso wichtiger, die richtige Balance zwischen den drei Prinzipien zu erreichen [20].
Wie der Begriff Modularität schon impliziert, ist das Modul die entscheidende Entität,

[1] http://felix.apache.org/
[2] https://www.knopflerfish.org/
[3] http://www.eclipse.org/equinox/

um die Wiederverwendung von Code zu fördern und die Komplexität bei der Entwicklung durch Zerlegung zu verringern. Module sind abgeschlossene funktionale Einheiten und bieten eine klare Trennung zwischen Schnittstelle und Implementierung. Sie stellen sozusagen die Strukturierungseinheit für Software dar. Zudem stellen Module nach außen Services für andere Module über eine definierte Schnittstelle zur Verfügung. Was zu einer weiteren Eigenschaft von Modulen führt: dem Black-Box Paradigma, womit lose Kopplung erreicht wird. Durch das Anbieten explizit definierter Schnittstellen bleibt der innere Aufbau und die innere Funktionsweise nach Außen hin verborgen, daher können intern beliebig viele Änderungen vorgenommen werden. In den folgenden Kapiteln wird das Wort Komponente benutzt, wenn diese umfassende Bedeutung gemeint ist.
Doch auch das Design von Modulen muss betrachtet und bewertet werden. Für die Analyse wurden zwei Metriken entwickelt, die Kohäsion und die Kopplung.
Ein fundamentales Konzept in der objektorientierten Programmierung ist es, eine geringe Kopplung zwischen Modulen und hohe Kohäsion innerhalb von Modulen zu erreichen. Dabei präsentiert Kopplung das Maß dafür, wie stark die Abhängigkeit zwischen Modulen ist. Kohäsion dagegen ist ein Maß dafür, wie stark die Abhängigkeit zwischen Funktionen innerhalb eines Moduls ist. Sowohl Kopplung als auch Kohäsion lassen sich in verschiedene Arten klassifizieren [51]. Da die Klassifizierungsmöglichkeiten keine relevante Rolle in dieser Arbeit spielen, wird an dieser Stelle nicht näher auf deren Arten eingegangen. Die OSGi-Serviceplattform basiert auf diesem Konzept der geringen Kopplung und hohen Kohäsion.

### 2.5.2 Framework

Der gezielte Einsatz von Access Modifier und Packages kann dazu beitragen, Java-Source Code modular zu gestalten. Wenn eine Klasse jedoch als public deklariert wurde, ist diese global zugänglich. Es ist nicht möglich, öffentliche Klassen eines Packages nur für bestimmte andere Packages freizugeben. Abhilfe schafft das übergeordnete Modulkonzept von OSGi. Wie im Kapitel 2.5 schon erwähnt, werden mithilfe des OSGi-Frameworks Java-Anwendungen in Module unterteilt. In OSGi werden diese Module als Bundles bezeichnet.

#### Bundles

Ein Bundle ist eine zusammenhängende Einheit von Klassen und anderen Ressourcen, deren Abhängigkeit zu anderen Modulen und Services explizit definiert sind. Um der zuvor beschriebenen Problematik, der globale Zugriff öffentlicher Klassen, entgegenzusetzen, kann ein Bundle selbst eine API auf Basis von Packages definieren. Öffentliche Klassen sind somit nur noch in Packages sichtbar, die expliziert exportiert wurden [60].
Ein Bundle wird anhand seiner Manifest-Datei definiert. Eine Manifest-Datei kann, wie in Listing 2.1 dargestellt, aufgebaut sein. Diese ist in dem zugehörigen JAR-Archiv enthalten, somit sind alle Metainformationen über ein Bundle in dem JAR-

Archiv verfügbar. Die Import-Package-Anweisung gibt an, von welchen Packages das Bundle abhängt. Es wird erkannt, welche Packages zu einer einwandfreien Ausführung eines Bundles benötigt werden. Für die Auflösung der Abhängigkeiten ist der OSGi-Container zur Laufzeit zuständig. Das Pendant zur Import-Package-Anweisung ist die Export-Package-Anweisung. Durch die Verwendung der Bundle-ManifestVersion-Anweisung ist es möglich, zu ein und derselben Komponente gleichzeitig mehrere Versionen laufen zu lassen [1].

Listing 2.1: Beispiel einer Manifest-Datei aus dem RCE-Projekt

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: RCE Components DOE Common
Bundle-SymbolicName: de.rcenvironment.components.doe.common
Bundle-Version: 8.1.0.qualifier
Bundle-RequiredExecutionEnvironment: JavaSE-1.7
Bundle-Vendor: DLR
Export-Package: de.rcenvironment.components.doe.common
Import-Package: de.rcenvironment.core.datamodel.api,
 de.rcenvironment.core.utils.common,
 org.apache.commons.csv;version="1.1.0",
 org.apache.commons.logging;version="1.1.1",
 org.codehaus.jackson;version="1.9.13",
 org.codehaus.jackson.map;version="1.9.13",
 org.codehaus.jackson.node;version="1.9.13"
```

### Services

OSGi Services werden durch Declarative Service Components deklariert, die durch je eine XML-Datei definiert werden (siehe Listing 2.2). Eine Service-Component kann auf mehrere Services verweisen (reference) und diese auch bereitstellen (provide). Interfaces dienen dazu Services zu definieren, die dann durch Bundles realisiert werden.
Mit dem Einsatz von OSGi Service Registry können Services leichter ausgetauscht werden, da sie von der konkreten Implementierung entkoppelt sind. Die Komponenten verstecken ihre Implementierungen, somit findet die Kommunikation über Services statt. Damit beispielsweise andere Bundles diese Services nutzen können, müssen diese importiert oder exportiert werden [60].

Listing 2.2: Deklaration eines Service-Components

```
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
activate="activate" name="Service Remote File Connection Factory">
        <implementation class="de.rcenvironment.core.communication.file.
        service.legacy.internal.ServiceRemoteFileConnectionFactory" />
        <service>
                <provide interface="de.rcenvironment.core.communication.
                fileaccess.spi.RemoteFileConnectionFactory"/>
        </service>
        <reference
                name="Communication Service"
                interface="de.rcenvironment.core.communication.api.
                CommunicationService"
                cardinality="1..1"
                bind="bindCommunicationService"
                policy="dynamic"
        />
</scr:component>
```

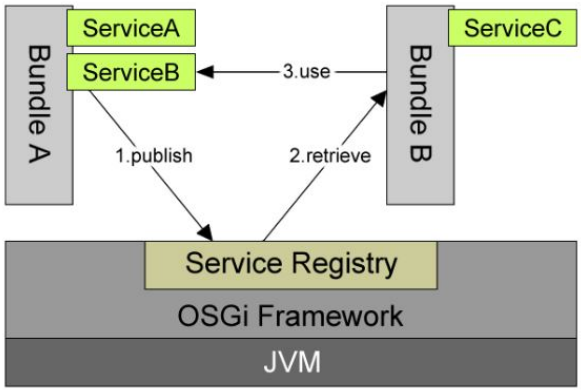



Abbildung 2.3: Bundles und Services im OSGi-Framework [56]

Das OSGi-Framework ist, wie in Abbildung 2.4 zu sehen, in vier konzeptionelle Ebenen unterteilt:

- Services Layer
- Life Cycle Layer
- Modules Layer

- Security Layer

Im Life Cycle Layer wird der Lebenszyklus von Modulen spezifiziert und kontrolliert. Module können dynamisch in ein laufendes System geladen und dessen Abhängigkeiten zur Laufzeit auflösen. Der dynamische Aspekt von OSGi wird im Rahmen dieser Arbeit aber nicht berücksichtigt. Genauso wie das Security Layer an dieser Stelle nicht weiter relevant ist [56].

### Modules Layer

In dem OSGi Modules Layer werden Bundles als Modularisierungseinheit definiert. Die Metadaten der installierten Bundles werden an dieser Stelle verarbeitet [1].

### Services Layer

Der OSGi Service Layer unterstützt eine serviceorientierte Architektur. Somit wird eine zuverlässige Kopplung zwischen Bundles über Interfaces ermöglicht, ohne dass eine explizite Abhängigkeit zwischen diesen vorhanden sein muss. In dem Service Layer können Services systemweit über ein Service Registry verfügbar gemacht werden [1].

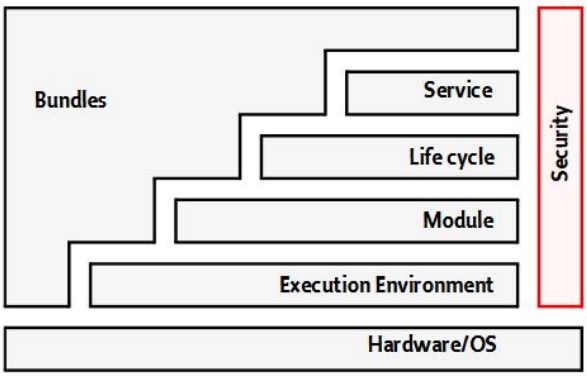



Abbildung 2.4: Aufbau des OSGi-Frameworks [1]

### Classloading

Wie unter dem Kapitel 2.5.2 schon erläutert, besitzt jedes Bundle in einem OSGi-Container einen eigenen Classloader. Dieser wird von der OSGi-Implementierung bereitgestellt und ist dafür zuständig, die Klassen des Bundles zu laden. Wird eine Klasse aus einem anderen Bundle benötigt, wird dazu der entsprechende Classloader angefragt, die Klassen werden somit von verschiedenen Classloadern geladen [60]. Durch diesen Classloading-Mechanismus werden die Klassen der einzelnen Bundles voneinander isoliert und der Zugriffsschutz wird durchgesetzt. Daher ist es möglich, dass in einer Anwendung mehrere Versionen ein und derselben Komponente zur gleichen Zeit ausgeführt

werden und in einer Anwendung mehrere Klassen existieren können, deren vollständig qualifizierte Namen identisch sind.
Diese Tatsache muss bei der Metamodellierung in Kapitel 4.2 berücksichtigt werden.

## 2.6 Software-Measurement

Auf die aus der Analyse gewonnenen Daten können Funktionen für die Bestimmung von Softwareeigenschaften angewandt werden. Dieser Prozess wird „Software-Measurement“, zu Deutsch „Softwaremessung oder -bewertung“, genannt und von Fenton folgendermaßen definiert [19]:

> „Measurement is concerned with capturing information about attributes of entities. (...) Measurement assigns numbers or symbols to these attributes of entities in order to describe them.“

Die visuelle Darstellung, siehe Kapitel 3, kann eine unterstützende Funktion einnehmen, um die Attribute der Entität zu bewerten. Im Rahmen dieser Arbeit handelt es sich bei der Entität um die Softwarearchitektur.
Es wird zwischen internen und externen Attributen unterschieden. Interne Attribute können meistens direkt gemessen werden, ohne auf andere Attribute zurückgreifen zu müssen. Softwareentwickler setzen sich hauptsächlich mit internen Attributen auseinander, zu denen unter anderem Größe, Modularität und Portierbarkeit gehören.
Externe Attribute sind vor allem für Endbenutzer und Manager von Interesse. Diese können nur indirekt durch die Betrachtung von internen Attributen bestimmt werden. Die Visualisierung kann dabei eine unterstützende Funktion einnehmen, beispielsweise kann aus dem internen Attribut der Komplexität eine Aussage über die Wartbarkeit der Software getroffen werden. Der Betrachter kann mithilfe der Visualisierung Rückschlüsse auf externe Attribute ziehen. Weitere externe Attribute sind beispielsweise: Zuverlässigkeit, Funktionalität und Benutzbarkeit.
Doch werden im Folgenden ausschließlich interne Attribute als Basis für die Visualisierung betrachtet.

### 2.6.1 Metriken

Es gibt eine große Anzahl von Softwaremetriken, die Kenngrößen der Software oder des Softwareentwicklungsprozesses messen, siehe Tabelle 2.2. Dabei handelt es sich um die Herleitung einer Zahl, welche die Eigenschaften von Software Code charakterisiert oder die Qualität der Software vorhersagen. Auch die quantitativen Aspekte der Qualitätskontrolle, wie das Erfassen und Überwachen von Fehlern während der Entwicklung fallen unter den Sammelbegriff Softwaremetriken [18]. Zudem gibt die Tabelle einen Einblick in die Attribute, welche am bekanntesten sind (die externen Attribute) und

diejenigen, welche direkt gemessen werden können (die internen Attribute). In den folgenden Abschnitten werden ausschließlich für diese Arbeit relevante Metriken, wie Größe, Kopplung und Kohäsion, genauer erläutert, also interne Attribute des Produkts.

| **Entitäten** | **Attribute** | |
|---|---|---|
| | **Intern** | **Extern** |
| **Produkte** | | |
| Spezifikation | Größe, Wiederverwendung, Modularität, Redundanz, Funktionalität, Syntaktische Korrektheit,... | Verständlichkeit |
| Design | Größe, Wiederverwendung, Modularität, Kopplung, Kohäsion, Funktionaliät, ... | Komplexität, Wartbarkeit, ... |
| Code | Größe, Wiederverwendung, Modularität, Kopplung, Funktionalität, algorithmische Komplexität, Strukturiertheit des Kontrollflusses, ... | Zuverlässigkeit, Benutzbarkeit, Wartbarkeit, Wiederverwendbarkeit |
| Testdaten | Größe, Testabdeckung, ... | Qualität, Wiederverwendbarkeit, ... |
| ... | ... | ... |
| **Prozesse** | | |
| Spezifikation | Zeit, Aufwand, Anzahl von Anforderungsänderungen, ... | Qualität, Kosten, Stabilität, ... |
| Design | Zeit, Aufwand, Anzahl der spezifizierten Fehler, ... | Kosten, Kosteneffizienz, ... |
| ... | ... | ... |
| **Ressourcen** | | |
| Personal | Alter, Preis, ... | Produktivität, Erfahrungen, Intelligenz |
| Team | Größe, Kommunikationslevel, Strukturiertheit, ... | Produktivität, Qualität |
| ... | ... | ... |

Tabelle 2.2: Klassifikation von Softwaremetriken (nach Fenton [18])

### Größe

Das Attribut Größe hat zwei unterschiedliche Bedeutungen: es wird zwischen funktionaler und technischer Größe unterschieden.

**Funktionale Größe** Die funktionale Größe gibt die Größe von Komponenten aus Benutzersicht an und stellt eine quantitative Bewertung der Funktionalität dar. Sie ist unabhängig von technischen Rahmenbedingungen. Ein verbreitetes Verfahren zur Bestimmung der funktionalen Größe ist die *Function-Point-Analyse* (FPA), welche in der ISO/IEC 20926 international normiert wurde. Bei der Function-Point-Analyse handelt es sich um eine Methode, um die Größe einer Software objektiv und unabhängig von der Programmiersprache, der Entwicklungsumgebung oder der Laufzeitumgebung zu messen. Die Größe der Software stellt in diesem Zusammenhang den Leistungsumfang, das heißt die Menge an Funktionen, die eine Software dem Anwender bietet, dar [22]. Da es sich hierbei aber lediglich um die Aufwandsschätzung in Softwareprojekten handelt, stellt die funktionale Größe der gesamten Software in dieser Arbeit keinerlei Bedeutung dar. Im Gegensatz zu der funktionalen Größe der einzelnen Komponenten, welche bei der Visualisierung von Interesse sein kann.

**Technische Größe** Bei der technischen Größe ist die wohl bekannteste Metrik die *Line of Code* Metrik (LOC). Die Anzahl der Code-Zeilen (LOC) sollte einen ersten Rückschluss auf die Qualität und Leistung ermöglichen. Zudem kann die Kennzahl helfen, den Entwicklungs- und Wartungsaufwand der Klassen besser einschätzen zu können [10]. Weitere Kennzahlen um die Größe von Klassen zu messen sind zum Beispiel die Anzahl von Attributen oder Methoden.
Doch in dieser Arbeit spielen neben Klassen und Methoden weitere Entitäten auf höherer Ebene, wie Packages und Bundles, eine wichtige Rolle. Hamza et al. [24] setzten bei der Betrachtung von OSGi-Anwendungen Metriken ein, deren kleinste Granularität die Klasse ist, zum Beispiel den Größenaspekt der einzelnen Komponenten. Diese können bezogen auf Bundles folgende Komponenten sein:

- Anzahl der enthaltenen Packages
- Anzahl der enthaltenen Klassen
- Anzahl der enthaltenen abstrakten Klassen und Interfaces

### Kopplung

Die Kopplung ist ein Maß, welches eine Aussage über die Abhängigkeit von Software-Komponenten trifft. Es besagt, wie stark zwei Komponenten zusammenhängen. Es gibt verschiedene Arten der Kopplung, die sich auf einer Skala von starker Kopplung bis hin zu schwacher Kopplung unterscheiden. Dabei steht die schwache Kopplung für die größtmögliche Unabhängigkeit zwischen Komponenten, was bei der Umsetzung einer Software angestrebt wird.
Im Zusammenhang mit dem Prinzip der geringen Kopplung und hohen Kohäsion wird der Begriff Kopplung nur für die modulübergreifenden Beziehungen eingesetzt. Beck und Diehl [4] betrachten den Begriff in ihrer Arbeit im allgemeinen für Beziehungen und

erläutern wie die verschiedenen Arten von Kopplung zwischen zwei Klassen bewertet werden können. Dabei unterscheiden sie zwischen, die für diese Arbeit relevanten, Arten : strukturelle Abhängigkeit und evolutionäre Kopplung. Mit struktureller Abhängigkeit ist die direkte Abhängigkeit zwischen zwei Klassen gemeint, eine Methode ruft eine andere Methode auf, eine Klasse erweitert eine andere Klasse oder eine Klasse aggregiert Objekte einer anderen Klasse. Bei der evolutionären Kopplung werden zwei Klassen häufig gemeinsam verändert. Dieses Vorgehen lässt sich auch auf die Kopplung zwischen Bundles, siehe Kapitel 2.5.2, übertragen. Dazu siehe folgende Tabelle 2.3:

| **Kopplungsart** | **Bewertung** |
|---|---|
| Strukturell | Anzahl der importierten Packages des anderen Bundles<br>Anzahl der Bundles<br>Anzahl der benutzen Services |
| Evolutionär | Anzahl der gemeinsamen Änderungen |

Tabelle 2.3: Bewertung der Kopplungsarten zwischen Bundles nach Beck und Diehl [4]

### Kohäsion

Die Kohäsion definiert das Maß der Abhängigkeiten zwischen Funktionen innerhalb eines Moduls. Ob die einzelnen Fragmente einer Komponente zusammenhängen und sich vorwiegend auf eine Funktion konzentrieren. Auch bei der Kohäsion gibt es unterschiedliche Arten, die sich auf einer Skala von zufälliger Kohäsion bis hin zur funktionalen Kohäsion unterscheiden [49]. Die einzelnen Stufen der Kohäsion spielen in dieser Arbeit nur eine untergeordnete Rolle, lediglich die Größe der Komponenten kann bei der Visualisierung für den Betrachter von Interesse sein.

## 2.7 Graphdatenbanken

Viele Jahre wurden in *relationalen Datenbankmanagementsystemen* (RDBMS) jegliche Arten von Daten abgespeichert. Ganz nach dem Motto „One Size Fits All“. Doch nicht für jeden Anwendungszweck und jede Art von Daten sind relationale Datenbanken geeignet. Klassische relationale Datenbanken verlieren an Bedeutung, da sie bei den Mengen an Daten heutzutage vermehrt an ihre Grenzen stoßen. Die Daten sind oftmals nicht in das klassische Raster der relationalen Datenbanken zu integrieren [32].
Die Informationen von Relationen sind in dieser Arbeit mindestens genauso wichtig wie die Entitäten selbst. Die Beziehungen der Objekte können untereinander mit Hilfe von gerichteten und ungerichteten Kanten spezifiziert werden. Denn Datenbanken, die Beziehungen als ein Kernaspekt ihres Datenmodells umfassen, sind in der Lage, Verbindungen effizient zu speichern, zu verarbeiten und abzufragen. Diese Eigenschaften sorgen dafür,

dass Graphdatenbanken einfach in der Bedienung sind. Viele Datensätze, so auch die Architektur eines Softwareprojektes, lassen sich auf natürliche Art und Weise auf Graphen abbilden. Zudem stehen Graphdatenbanken für eine gewisse Agilität. Sie unterliegen keiner Beschränkung oder starren Datenschemata [53].
Obwohl relationale Datenbanken im Vergleich zu Graphdatenbanken durch ihren jahrelangen Einsatz in Industrie und Forschung erprobter und dadurch auch ausgereifter sind, ist ihr Datenschema festgelegt, was die Erweiterbarkeit und Zusammenführung von Daten erschwert. Zudem werden Relationen zwischen Objekten erst zur Abfragezeit berechnet, siehe Abbildung 2.5. Für die Abfrage komplizierter Beziehungsgeflechte bedarf es einer großen Anzahl an Join-Klauseln, welche teuer zu berechnen sind. Im Gegensatz zu den relationalen Datenbanken weisen Graphdatenbanken mehr Flexibilität auf und sind in der Lage, relevante Informationsnetzwerke transaktional zu speichern und besonders schnell und effizient abzufragen.

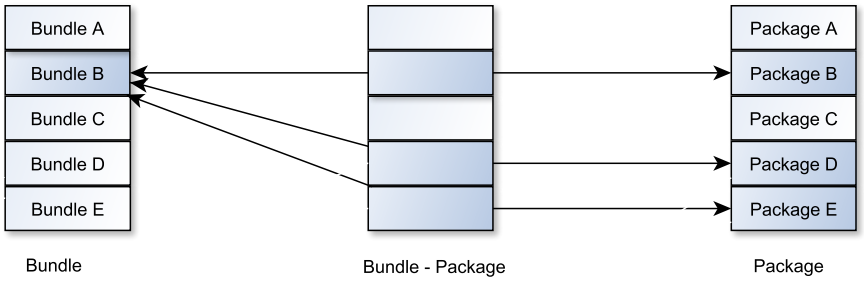



Abbildung 2.5: Abfrage von Beziehungen mit Hilfe von JOIN-Klauseln

Relationale Datenbanken sind ohne Zweifel ausgereifter, doch Graphdatenbanken bestechen durch ihre Flexibilität und Agilität und weisen eine bessere Performance im Umgang mit verknüpften Daten auf.

**Flexibilität** Mit dem Einsatz von Graphen besteht die Möglichkeit, einer vorhandenen Struktur neue Arten von Beziehungen, neue Knoten, neue Beschriftungen und neue Untergraphen hinzuzufügen, ohne vorhandene Abfragen und Anwendungsfunktionen verändern zu müssen. Somit sind Strukturen und Schemata nicht von Beginn an festgesetzt. Die Folge ist, dass aufgrund der Flexibilität die Domäne nicht im Voraus detailliert festgeschrieben werden muss.

**Agilität** Die schemafreie Eigenschaft des Graphdatenbankmodells und die Abfragesprache einer Graphdatenbank bieten die optimale Möglichkeit, ein Projekt systematisch weiterzuentwickeln und anzupassen.

**Performance** Bei relationalen Datenbanken verschlechtert sich die Abfragezeit mit größer werdendem Datensatz. Im Gegensatz dazu bleibt die Leistung der Graphdatenbanken relativ konstant. Die Ausführungszeit für jede Abfrage ist nur proportional zu

der Größe des Teils des Graphen, der durchquert wird, um die Abfrage zu erfüllen, und nicht der Größe des Gesamtgraphen [53].

Die Flexibilität und Agilität spielen besonders im Hinblick auf die Anwendung architekturrelevanter Regeln eine große Rolle. Denn durch den Architekturvergleich von zwei Systemversionen werden immer wieder Beziehungen und Untergraphen hinzugefügt und gelöscht, siehe Kapitel 5.2. Der Inhalt der Graphdatenbank befindet sich in einer stetigen Veränderung und ist nicht vorhersehbar.

Da die Reaktionszeit der Graphdatenbank eine wichtige Anforderung ist, wurde von Partner und Vukotic anhand eines Experimentes gezeigt, dass die Performance der Graphdatenbank Neo4j im direkten Vergleich mit einer relationalen Datenbank im Umgang mit verbundenen Daten wesentlich besser ist [53].

Wie in der Abbildung 2.6 zu sehen ist, stehen die relationale Datenbank und die Graphdatenbank bei einer Abfrage der Abfragetiefe zwei sich nichts nach. Eine Abfrage der Tiefe zwei könnte zum Beispiel der Namen eines Freundes von einem Freund sein. Doch schon bei einer Abfrage der Tiefe drei stellt sich heraus, dass die relationale Datenbank die Anfrage nicht mehr in einem vernünftigen Zeitraum bearbeiten kann. Im Gegensatz zu der Graphdatenbank, deren Reaktionszeit sehr gering geblieben ist. Spätestens bei einer Abfrage der Tiefe fünf lässt sich eindeutig erkennen, dass Graphdatenbanken im Umgang mit verbundenen Daten besser geeignet sind.

| Depth | RDBMS execution time(s) | Neo4j execution time(s) | Records returned |
|---|---|---|---|
| 2 | 0.016 | 0.01 | ~2500 |
| 3 | 30.267 | 0.168 | ~110,000 |
| 4 | 1543.505 | 1.359 | ~600,000 |
| 5 | Unfinished | 2.132 | ~800,000 |

Abbildung 2.6: Reaktionszeit relationale Datenbank versus Graphdatenbank [53]

Wie im Experiment kommt auch in dieser Arbeit die Graphdatenbank Neo4j zum Einsatz.

### 2.7.1 Neo4j

Es gibt eine Vielzahl verschiedener Graphdatenbanken. Zu den bekanntesten Open Source Graphdatenbanken gehören unter anderem ArangoDB, Neo4j, GraphBase, Titan und HypergraphDB. Von den genannten Graphdatenbanken besitzt ausschließlich Neo4j eine eigene, eigens von Neo4j selbst entwickelte Abfragesprache - Cypher. Die leistungsstarke

Abfragesprache Cypher ist besonders für den zweiten Teil der Arbeit, der Visualisierung, von großer Wichtigkeit.

Bei Neo4j handelt es sich zudem um eine quelloffene ACID-transaktionale Graphdatenbank, welche in der Community-Version unter der General Public License v3 veröffentlicht wurde. Des weiteren ist die Graphdatenbank über eine integrierte REST-Web-API-Schnittstelle erreichbar und unterstützt Hot-Backups. Daher gehört Neo4j zu den meist verwendetsten Graphdatenbanken [28].

### Cypher

Cypher ist eine deklarative Abfragesprache spezifisch für Neo4j. Mit Hilfe von Cypher kann die Datenbank aufgefordert werden, Daten zu finden, die einem bestimmten Muster entsprechen. Die Anfragen werden auf Basis des American Standard Code for Information Interchange (ASCII) formuliert [48].
Wie die meisten Abfragesprachen besteht Cypher aus Klauseln. Die grundlegende Abfrage besteht aus einer *MATCH*-Klausel gefolgt von einer *RETURN*-Klausel. Dabei gibt die *RETURN*-Klausel an, welche Knoten, Beziehungen und Eigenschaften an den Client zurückgegeben werden soll [42].

# 3 Softwarevisualisierung - The Visualization Basics

> 'Es gibt Dinge, die sind schwer zu erkennen, nicht etwa weil sie für unsere Augen zu klein waren, sondern weil sie zu complex sind.' V. Braitenberg [17]

Bei der Softwarearchitektur handelt es sich um ein unsichtbares und immaterielles Konzept, sie ist nicht direkt greifbar. Quellcode als Darstellungsart ist hierbei keine Alternative. Ein Entwickler kann durchschnittlich 30 000 Zeilen Code überblicken, doch heutzutage bewegen sich die Softwaresysteme in der Größenordnung zwischen 200 000 und 100 Millionen Zeilen Code [36]. Daher ist eine Betrachtung auf höherer Abstraktionsebene nötig. Eine geeignete Visualisierung der Softwarearchitektur ist somit sowohl für Softwarearchitekten als auch für Entwickler, Tester und für das Projektmanagement gewinnbringend. Mit dem Ziel Softwaresysteme zu verstehen und die Produktivität des Entwicklungsprozesses zu verbessern.
Trotzdem die Softwarevisualisierung mittlerweile ein großes Forschungsgebiet ist, mangelt es weiterhin an praxistauglichen Werkzeugen. Im folgenden Kapitel soll eine Übersicht über den aktuellen Stand der Technik im Bereich der Softwarevisualisierung gegeben werden.

## 3.1 Datenvisualisierung

Die Datenvisualisierung umfasst viele verschiedene Teilbereiche, dazu gehört auch die Softwarevisualisierung, genauso wie zum Beispiel die Informationsvisualisierung, wissenschaftliche Visualisierung oder medizinische Visualisierung. Sie stellt eine Art der Kommunikation von Informationen mittels grafischer Darstellungen dar, mit dem Ziel den Entscheidungsprozess und die Erkenntnisgewinnung eines Menschen zu unterstützen [59].
Visualisierungen werden schon seit Jahrhunderten als Kommunikationsmittel verwendet. Sie enthalten eine Fülle an Informationen, welche vom Menschen schneller als beim Lesen von Wörtern aufgenommen werden können. Denn im menschlichen Wahrnehmungssystem finden Bildinterpretationen parallel statt,während durch den sequentiellen Prozess des Lesens die Textanalyse begrenzt ist [59].

#### Grafische Semiologie

In der Semiologie wird versucht, bestimmte Bedeutungen auf grafische Symbole abzubilden. Die größte Herausforderung dabei ist die grafische Darstellung entsprechend auszuwählen, dass sie eindeutig interpretierbar ist, ohne sie vorher definieren zu müssen. Dies bedeutet, dass zum einen die Darstellung des einzelnen Objektes an sich und zum anderen die Darstellung aller Objekte zusammen betrachtet werden muss [5].
Des Weiteren können die einzelnen Komponenten einer Visualisierung mit grafischen Variablen versehen werden, um den Informationsgehalt zu erhöhen beziehungsweise verständlicher zu gestalten. Grafische Variablen sind:

- Größe
- Form
- Position
- Textur
- Helligkeit
- Richtung
- Farbe

Die Variablen Form und Textur können zur Unterscheidung von Komponenten eingesetzt werden, wenn keinerlei Ordnungsrelationen der Komponenten zueinander vorliegen. Helligkeit und Größe dagegen können eingestezt werden, um Ordnungsrelationen hervorzuheben.

### 3.1.1 Interaktionsmethoden

Die Interaktion ist ein wichtiges Werkzeug in der Datenvisualisierung. Denn mit Hilfe folgender Interaktionsmethoden wird dem Betrachter die Möglichkeit gegeben, die Visualisierung anzupassen [59]:

**Filterung** Die angezeigten Daten können reduziert werden, indem zum Beispiel Daten ausgeblendet werden.

**Navigation** Mit Hilfe von Zoomen, Rotieren oder Verschieben können Blickwinkel und Ansichten verändert werden.

**Selektion** Bestimmte Elemente oder Teile der Darstellung, die von besonderem Interesse sind, lassen sich auswählen.

**Verbindung** Ermöglicht das Aufzeigen von Verbindungen und Beziehungen.

**Kodierung** Die grafische Darstellung kann verändert werden.

**Rekonfigurierung** Ermöglicht das Verändern des Layouts oder die Darstellung der Elemente.

**Abstraktion** Der Detailgrad kann mit Hilfe der Abstraktion angepasst werden.

## 3.2 Definition und Ziele

Es ist besonders zeitaufwendig und teuer große und komplexe Softwaresysteme anhand des Quellcodes zu verstehen [9]. Die Abbildung 3.1 macht das typisches Verhältnis zwischen Code-Lesen und Code-Schreiben deutlich, welches bei 70% Code verstehen, 20% Problem lösen und 10% Code schreiben liegt [36]. Um diesen Zeit- und Arbeitsaufwand zu minimieren, können Kosten durch die Verwendung von Visualisierungswerkzeugen reduziert werden, damit eine effiziente Sicht der Software geschaffen wird.

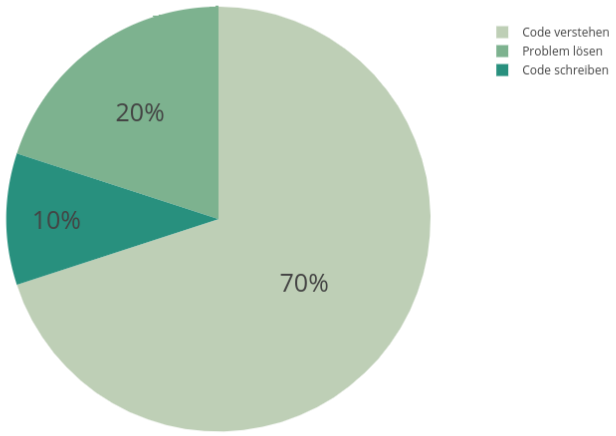



Abbildung 3.1: Verhältnis zwischen Code-Lesen und Code-Schreiben

In einer Umfrage [33] wurden 82 Forscher bezüglich des Themas Softwarevisualisierung befragt. Auf die Frage wie wichtig sie die Visualisierung von Software einstufen, antworteten 42% mit sehr wichtig, 40% mit wichtig und nur 1% mit unwichtig. Die Umfrage zeigt, dass Softwarevisualisierung ein wichtiger Bereich in der Softwareentwicklung ist.
Sie kann durch folgende Aspekte den Entwickler und Architekten unterstützen [46]:

1. Verständnis
2. Debugging
3. Design-Entscheidung

Gershon [14] definierte die Visualisierung folgendermaßen:

> 'Visualization is more than a method of computing. Visualization is the process of transforming information into a visual form, enabling users to observe the information. The resulting visual display enables the scientist or engineer to perceive visually features which are hidden in the data but nevertheless are needed for data exploration and analysis.'

### 3.2.1 Visualisierungs-Pipeline

Der Prozess der Softwarevisualisierung umfasst weit mehr als die bloße Darstellung von Grafiken und Bildern. Die Visualisierungs-Pipeline beschreibt den Prozess von der Datenerfassung bis hin zu der graphischen Darstellung [14].

**Datenerfassung** Wie in Kapitel 2.1.1 schon erwähnt, gibt es verschiedene Quellen an Informationen. Die Methoden um relevante Daten aus den Quellen zu extrahieren und zu sammeln, sind so unterschiedlich wie die Quellen selbst.

**Analyse** Verschiedene Analyseverfahren können genutzt werden, um die Datenmenge auf die wichtigsten Bestandteile zu beschränken. Analyseverfahren können Filter, statische Programmanalysen oder statistische Methoden sein.

**Visualisierung** Die aus der Datenerfassung und Analyse resultierenden Daten werden auf ein visuelles Modell abgebildet, das heißt in geometrische und graphische Informationen transformiert.

In dieser Arbeit wird die gesamte Visualisierungs-Pipeline durchlaufen. Von der Extraktion der Daten aus dem Source Code über die Filterung von architekturrelevanten Änderungen bis hin zu der Visualisierung der Evolution einer Softwarearchitektur.

## 3.3 Klassifizierung

Softwarevisualisierungen werden häufig nach den folgenden Kategorien klassifiziert.

### 3.3.1 Aspekte von Software

Visualisierungen neigen dazu, bei besonders komplexen Softwaresystemen nur bestimmte Aspekte anzusprechen [58]. Dabei lassen sich die Aspekte in folgende drei Arten einteilen:

1. Statisch
2. Evolution
3. Dynamisch

Im Gegensatz zu den dynamischen Aspekten, wo eine Programmausführung notwendig ist, um Informationen eines bestimmten Programmlaufs zu erfassen und ein Verständnis des Ausführungsverhaltens zu erhalten, werden statische Aspekte aus Quellen vor der Ausführung abgeleitet, wie zum Beispiel vom Quellcode. Die Visualisierung der Evolution zeigt, wie sich die statischen Aspekte einer Software im Laufe der Zeit verändern. Im Rahmen dieser Arbeit sollen nur die ersten beiden Punkte betrachtet werden, daher werden im Folgenden bei der Visualisierung ausschließlich die statischen und evolutionären Aspekte berücksichtigt.

### 3.3.2 Granularität

Die Softwarevisualisierung kann auf drei verschiedenen Abstraktionsebenen erfolgen. Caserta und Zendra [9] klassifizieren sie nach ihrer Granularität, basierend auf der Abstraktionsebene, die sie darstellen:

- Source Code-/Instruktionsebene
- Package-, Klassen und Methodenebene
- Architekturebene

Die niedrigste Abstraktionsebene, die Source Codeebene, befasst sich mit dem Quellcode und ist für jedes Projekt sehr bedeutend, da die IDE auch als Low-Level-Form der Visualisierung angesehen werden kann. Die mittlere Ebene gibt Einblick in die Funktionsweise einer Klasse oder einer Methode. Auf der obersten Ebene wird die Gesamtheit der Softwarearchitektur visualisiert und gehört damit zu dem wichtigsten Bereich der Softwarevisualisierung.
Diese Arbeit beschränkt sich auf die Architekturebene, sie gibt Einblick in die zugrunde liegende hierarchische Komponentenstruktur und deren Beziehungen.

### 3.3.3 Anzahl der Sichten

Eine Softwarevisualisierung kann aus einer oder mehreren Sichten bestehen. Dabei verwendet jede Sicht einen anderen Ansatz und kann sich somit auf verschiedene Aspekte der Software konzentrieren. Dieser sogenannte Multi-View-Ansatz stellt eine Vielzahl an Informationen von unterschiedlicher Granularität dar. Mehrere Interessengruppen mit unterschiedlichen Anforderungen an die Visualisierung werden angesprochen, die jedoch durch die Vielzahl an Informationen auch eine erhebliche kognitive Belastung für den Benutzer sind. Zudem wird die Kommunikation zwischen den Interessengruppen erschwert, da keine gemeinsame Grundlage vorhanden ist. Denn jeder verwendet in erste Linie eine andere Sichtweise.
Neben dem Multi-View-Ansatz gibt es den Single-View-Ansatz, welcher einfacher zu

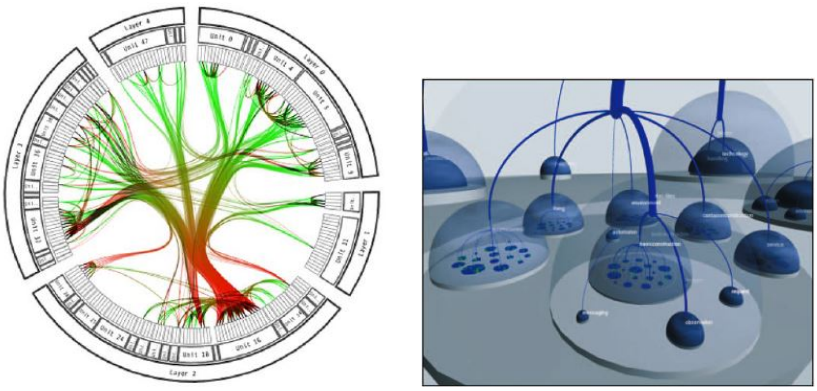
navigieren ist und zu einem kollektiven Verständnis der Software beiträgt, da alle Beteiligten mit der gleichen Sicht arbeiten und somit alle die gleichen Informationen als gemeinsame Grundlage vor sich liegen haben [45]. Allerdings kann als Folge dessen nur eine verminderte Menge an Informationen dargestellt werden.

### 3.3.4 Dimensionen

Visualisierungen können für den zweidimensionalen oder dreidimensionalen Raum entwickelt werden, siehe Abbildung 3.2. Da die Benutzer heutzutage mit einer 2D-Desktop-Umgebung vertraut sind, wird die zweidimensionale Visualisierung als leicht zu bedienen eingestuft. Die Navigation und Interaktion sind bekannt und somit einfacher. Wie im vorherigen Abschnitt 3.3.3 schon erwähnt, wird bei komplexen Projektsystemen oder schnell wachsenden Datensätzen teilweise der Mulit-View-Ansatz verfolgt. Dieser wird gerne in einer zweidimensionalen Umgebung eingesetzt, um eine überladene Ansicht zu vermeiden. Dadurch wird hingegen auch die Visualisierung komplexer und unübersichtlicher, was zu einer kognitiven Überbelastung führen kann. Um dem entgegenzuwirken, kann das Hinzufügen einer dritten Dimension das Raumproblem verbessern. Dadurch wird die Informationsdichte in der Visualisierung erhöht. Da hierfür das Wahrnehmungssystem beansprucht wird, führt dies zu keiner kognitiven Belastung [52]. Die Benutzer können einen besseren Bezug zu Metaphern aus der realen Welt herstellen und dadurch Informationen besser aufnehmen. 3D-Visualisierungen stellen diese realen Metaphern genauer dar wie 2D-Visualisierungen, was als weiterer Vorteil berücksichtigt werden muss. Jedoch sollen auch folgende Nachteile gegenüber der Visualisierung in einem zweidimensionalen Raum genannt werden. Die Rechenkomplexität ist bei einer 3D-Visualisierung wesentlich höher. Zudem ist die Navigation und Interaktion in einem dreidimensionalen Raum wesentlich komplexer, da den Benutzern hauptsächlich der Umgang mit 2D-Desktop-Umgebungen gebräuchlich ist [57]. Des Weiteren sind nicht alle Informationen auf den ersten Blick ersichtlich, Objekte können für den Betrachter teilweise verborgen erscheinen, was für diesen eine Verzerrung des Informationsgehaltes zur Folge hätte.

### 3.3.5 Virtual Reality

Wie im vorherigen Kapitel schon erwähnt, können 3D-Visualisierungen gegenüber einem zweidimensionalen Raum mehr Informationen darstellen. Zudem bietet die 3D-Visualisierung die Möglichkeit des Einsatzes von Virtual Reality. Mit Hilfe von Virtual Reality steigt der Betrachter direkt in die Metapher ein. Dadurch wird dem Betrachter eine noch tiefergehende Auseinandersetzung mit der Softwarevisualisierung ermöglicht. Allerdings ist für die Umsetzung zusätzliche Hardware erforderlich. Zwar ermöglichen VR-Brillen eine intuitive Interaktion, indem die Kopfbewegung des Betrachters mit Hilfe von Sensoren erfasst werden und sich somit die Darstellung automatisch an diese anpasst. Doch seien an dieser Stelle auch Begriffe wie Simulator Sickness oder Disorientierung erwähnt, an denen einige Menschen unter der Verwendung von VR-Brillen leiden.



(a) (b)

Abbildung 3.2: (a) Beispiel einer 2D-Visualisierung [9] (b) Beispiel einer 3D-Visualisierung [57]

Trotzdem wird immer mehr im Bereich der Softwarevisualisierung unter Einsatz von Virtual Reality geforscht. Doch äußern Softwarearchitekten und Entwickler immer wieder Kritik, dass gemeinschaftliches Arbeiten innerhalb eines Teams mit einer VR-Brille nicht möglich ist.
An dieser Stelle wird darauf hingewiesen, dass die Visualisierung in Virtual Reality nicht Bestandteil dieser Arbeit ist. Daher werden in dieser Arbeit ausschießlich Visualisierung als Desktopanwendung betrachtet.

## 3.4 Herausforderungen

Die größte Herausforderung in der Softwarevisualisierung liegt darin, ein wirkungsvolles Mapping-Schema zwischen den Komponenten der Softwarearchitektur und der visuellen Metapher zu finden. Dabei sind zwei Aspekte bei der Ausarbeitung eines wirkungsvollen Mapping-Schema von entscheidender Bedeutung.
Zunächst hat jede an einem Softwareprojekt beteiligte Person einen anderen Anspruch an die Visualisierung und deren Informationsgehalt. Daher ist es unwahrscheinlich, dass ein einziges Softwarevisualisierungstool allen Ansprüchen gleichzeitig gerecht werden kann [47]. Um so wichtiger ist eine klare Definition der Zielgruppe, um die, für deren Aufgabengebiet, am besten geeignete Visualisierung zu finden.
Zum anderen muss eine geeignete Lösung für den Maßstab der Visualisierung entwickelt werden. Neben der Datenkomplexität, welche durch die große Anzahl an Komponenten und Beziehungen entsteht, gibt es noch den wichtigen Aspekt der visuellen Komplexität. Das menschliche Gehirn hat nur beschränkte Fähigkeiten und ist daher nicht in der Lage, extrem komplexe Datenmodelle mit vielen einzelnen Komponenten zu verarbeiten [23]. Daher ist, „das einfache Umpacken von massiven Textinformationen in eine

massive grafische Darstellung nicht hilfreich.“ [47]. Durch die Datenkomplexität sind die zu visualisierenden Informationen zu groß, um sie alle vollständig in einer Ansicht zu erfassen. Daher ist die Auswahl einer effektiven und nutzbringenden Sichtweise für eine bestimmte vordefinierte Aufgabe entscheidend.

## 3.5 Metapher

Im folgenden Kapitel sollen bekannte Visualisierungstechniken im Bereich der Architekturvisualisierung und der Visualisierung der Evolution vorgestellt werden.
Leonel Merino et al. [40] betrachteten 368 Paper, welche auf der SOFTVIS- und VISSOFT-Konferenz veröffentlicht wurden. Dabei analysierten sie 86 Paper bezüglich der Art der eingesetzten Visualisierungstechnik genauer. Die Abbildung 3.3 zeigt eine Zuordnung zwischen der zu beantwortenden Problemstellung und der Art der eingesetzten Visualisierungstechnik, welche dafür vorrangig eingesetzt wurden. Bei der Betrachtung der Felder History und Architecture fällt besonders auf, dass beide häufig mit Hilfe geometrischer Formen visualisiert werden.

### 3.5.1 Visualisierung der Architektur von Software

Die Visualisierung auf Architekturebene ist eines der größten Gebiete der Softwarevisualisierung. Im folgenden Abschnitt sollen bekannte 2- und 3-Dimensionale Visualisierungstechniken vorgestellt werden.

Baummodelle dienen oftmals zur Visualisierung von Organisationen und Hierarchien. Allerdings ist das klassische Baummodell, siehe Abbildung 3.4 a, bei der Datenkomplexität großer und langlebiger Softwaresysteme ungeeignet, da die große Anzahl an Blättern schnell unübersichtlich und ausufernd wirken würde.
Für eine Darstellung auf höherer Abstraktionsebene eignet sich die von Johnson und Shneiderman entwickelte Treemap-Visualisierungstechnik, siehe Abbildung 3.4 b. Nachteil an dieser Form der Visualisierung ist zum einen die schwere Unterscheidbarkeit der Elemente und zum anderen die fehlende Visualisierung der hierarchischen Strukturen [9]. Daher sind sowohl das Baummodell als auch die Treemap-Visualisierung für die Darstellung der Komponenten und Beziehungen einer großen Software ungeeignet.

Mit Hilfe der Sunburst-Visualisierung wurden die Problematiken der Unterscheidbarkeit der Elemente und der Darstellung hierarchischer Strukturen angegangen, siehe Abbildung 3.5 a. Der Kreis in der Mitte symbolisiert das Wurzelelement und die Ringe stellen die Hierarchieebene dar. Um so tiefer sich die Komponente auf der Hierarchieebene in der Software befindet, desto weiter ist es vom Wurzelelement in der Mitte entfernt. Doch Experimente haben gezeigt, dass kreisförmige Treemaps in einem dreidimensionalen Raum besser die unterschiedlichen Hierarchieebenen und die Unterscheidbarkeit

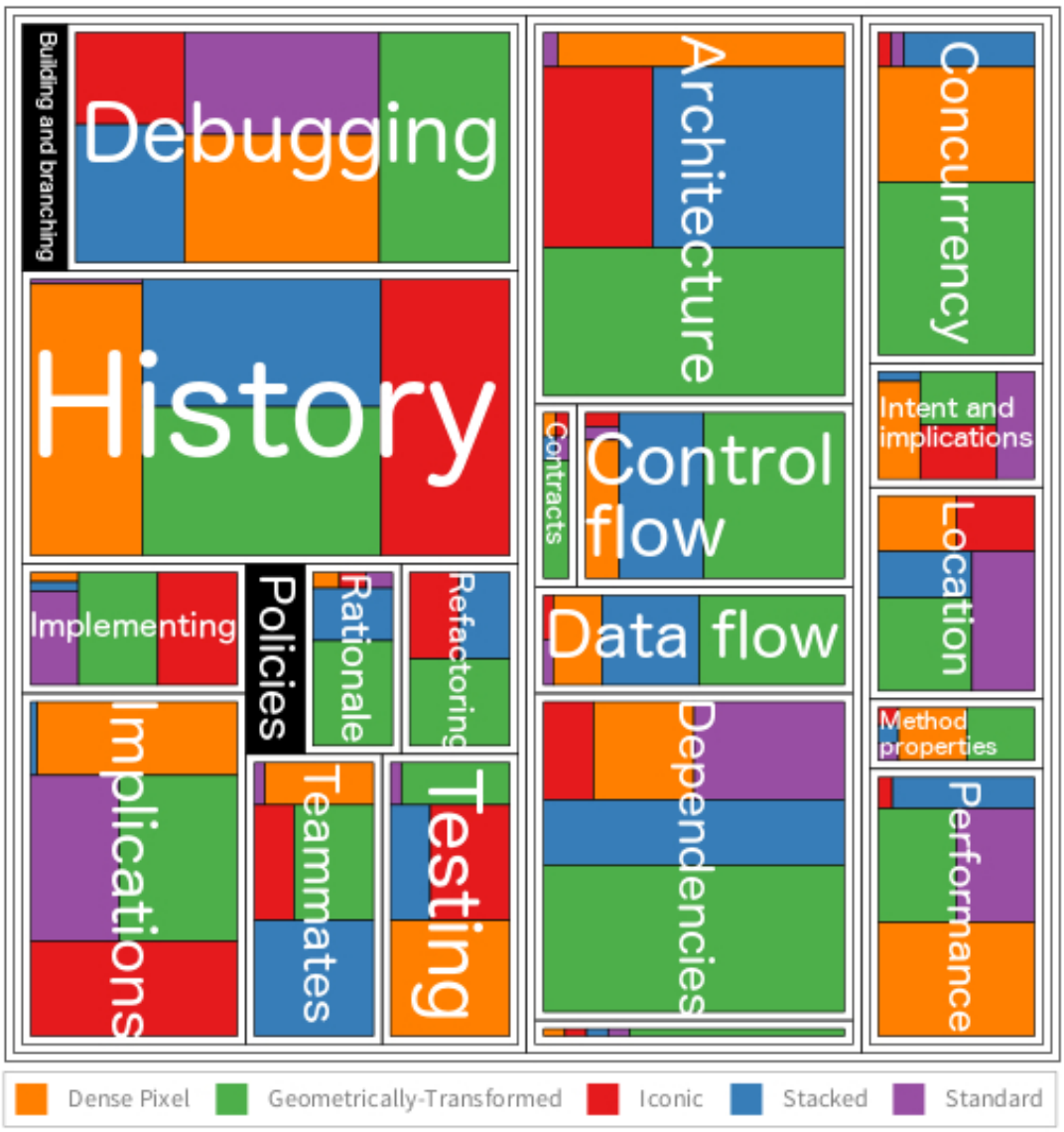
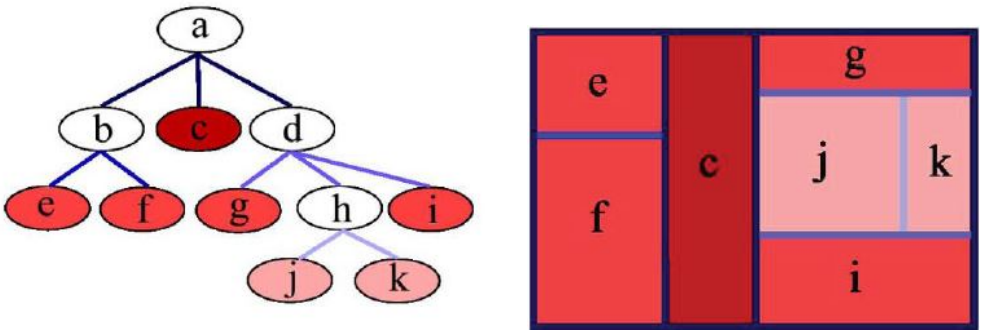



Abbildung 3.3: Mapping zwischen der zu beantwortenden Problemstellung und der umgesetzten Visualisierung [40]

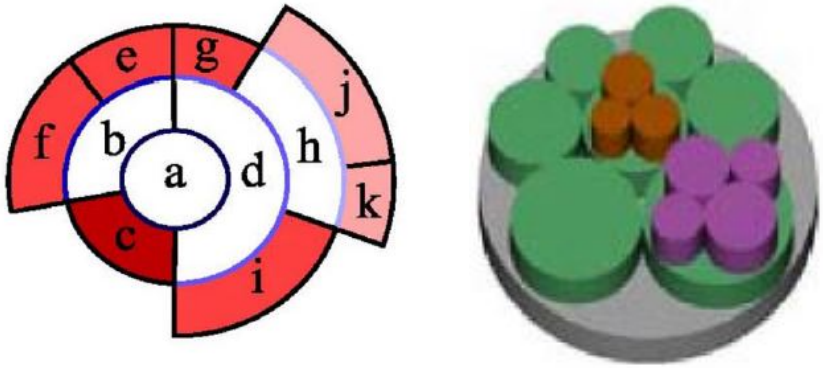
der einzelnen Komponenten darstellen wie Treemaps in 2D [9]. Da mit Hilfe der dritten Dimension zusätzlich mit unterschiedlichen Höhenniveaus gearbeitet werden kann, woraus sich für den Betrachter besser die Hierarchie einer Komponente ableiten lässt, siehe Abbildung 3.5 b.

Aufgrund der zahlreichen Beziehungen und Abhängigkeiten in einem großen und langlebigen Softwaresystem stellt deren Visualisierung eine große Herausforderung dar. Überlappungen und Verdeckungen sind die größte Problematik. Abgeleitet von der be-




(a) (b)

Abbildung 3.4: (a) Baummodell [9] (b) Treemap [9]



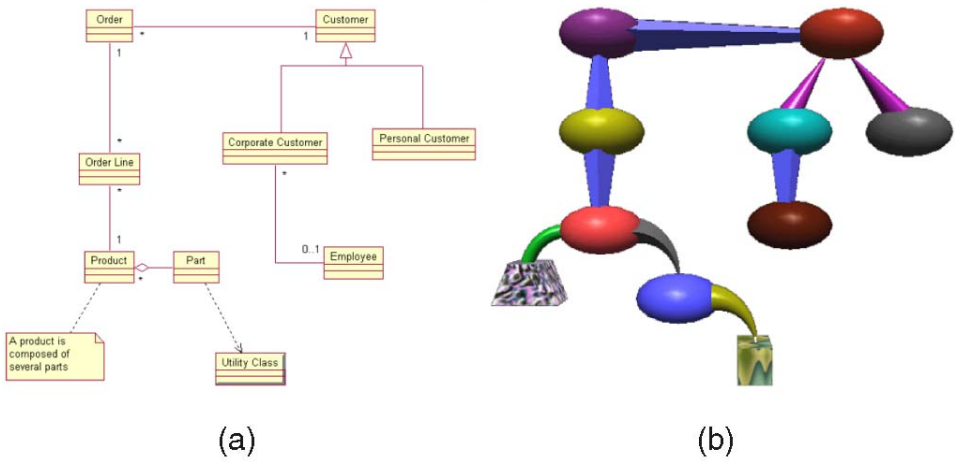

(a) (b)

Abbildung 3.5: (a) Sunburst-Visualisierung 2D [9] (b) Sunburst-Visualisierung 3D [9]

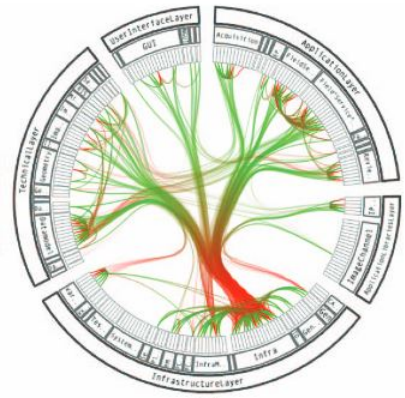
kanntesten Softwaredarstellung, dem UML-Diagramm, leiteten Iranie et al. [30] die Visualisierung unter Verwendung des Geon-Diagramms ab, siehe Abbildung 3.6 a. Mit Hilfe von 3D-Körpern werden die einzelnen Komponenten dargestellt und liefern Informationen, wie sie miteinander verbunden sind. Als weitere Technik zur Darstellung der Beziehungen sei das Rufdiagramm genannt [26], siehe Abbildung 3.6 b. Zum Rufdiagramm sei gesagt, dass die Anzahl der Komponenten beschränkt ist und zudem sind Beziehungen bei der Darstellung einer großen Software schlecht nachvollziehbar.

### Reale Metapher

Globale Softwarestrukturen und komplexe Situationen werden laut Dos Sontas et al. [54] unter Verwendung einer realen Metapher schneller und besser erkannt, wenn der Betrachter einen vertrauten Kontext wiederfindet. Reale Metaphern sind Visualisierungen, die




(a)

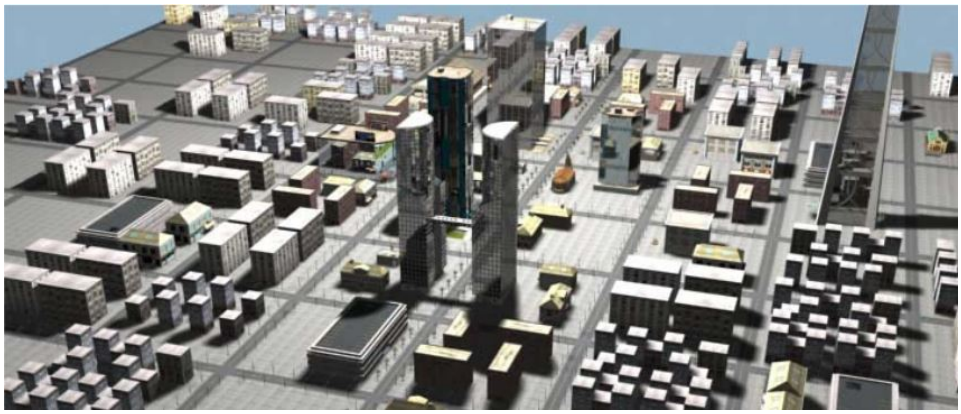


(b)

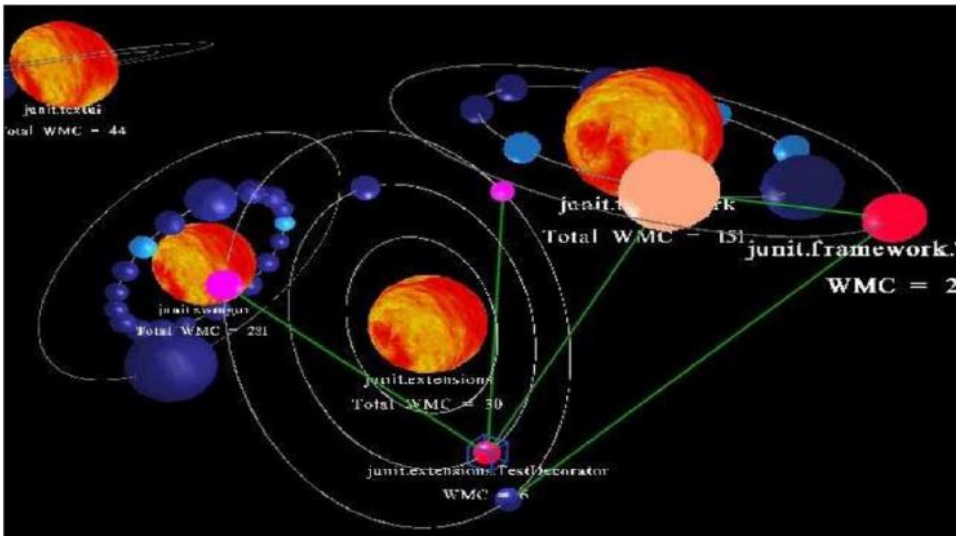
Abbildung 3.6: (a) Geon-Diagramms [30] (b) Rufdiagramm [26]

aus der physischen Welt abgeleitet wurden.
Visualisierungen basierend auf realen Metaphern machen sich das für den Betrachter Bekannte zunutze, wie zum Beispiel die City-Metapher, siehe Abbildung 3.7 a. Damit wird eine intuitive Interpretation von Software geschaffen. Große Städte sind zumeist so strukturiert, dass Elemente in Unterelemente zerlegt werden können. Diese unterschiedlichen Granularitätsebenen können hervorragend genutzt werden, um die Hierarchie abzubilden. Beispielweise kann eine Stadt ein Softwaresystem repräsentieren, Bezirke Bundles, Gebäude Packages und die einzelnen Etagen stellen die zugehörigen Klassen dar. Dabei steht jedoch immer die Visualisierung der Hierarchie im Vordergrund. Abhängigkeiten und Beziehungen zwischen den Komponenten sind in einer City-Metapher schwer darstellbar, denn Straßen würden nicht ausreichen, um die Zusammengehörigkeit übersichtlich darstellen zu können.

Eine weitere sehr bekannte Visualisierungstechnik ist das Sonnensystem. Wie in Abbildung 3.7 b zu sehen ist, wird die Software als virtuelle Galaxie präsentiert. Ein Package wird durch eine Sonne dargestellt, während mehrere Planeten, die für die einzelnen Klassen stehen, die Sonne auf verschiedenen Umlaufbahnen umkreisen. Allerdings ist die Visualisierung der Beziehungen untereinander schwierig, da die Metapher keine „natürliche“, dem Sonnensystem entsprechende Zuordnung dafür bietet.



(a)



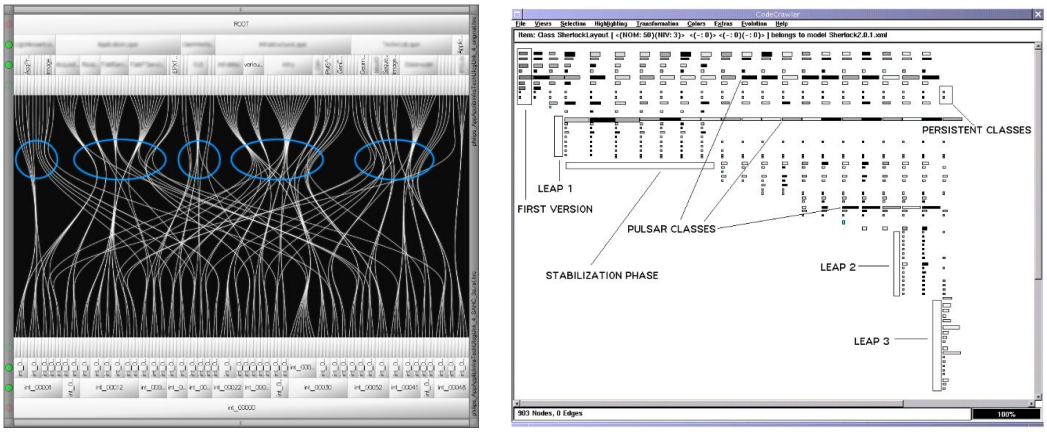

(b)

Abbildung 3.7: (a) City-Metapher [9] (b) Sonnensystem [43]

### 3.5.2 Visualisierung der Evolution von Software

Ein Softwaresystem durchläuft eine Evolution, in der sie sich kontinuierlich verändert, siehe Kapitel 2.2.1. Daher sollen in der Visualisierung einer Softwarearchitektur nicht

nur Komponenten, Beziehungen und Hierarchien dargestellt werden, sondern auch die Entwicklung der Software über den gesamten Lebenszyklus hinweg [11]. Doch sobald die zeitliche Dimension zur Visualisierung hinzugefügt wird, nimmt der Informationsgehalt signifikant zu. Was wiederum schnell zu einer visuellen Komplexität führen kann, siehe Kapitel 3.4. Caserta und Zendra liefern in ihrem Paper [9] einen umfassenden Überblick über den aktuellen Stand im Bereich der Softwarevisualisierung. Sie präsentieren ein breites Spektrum der verschiedensten Visualisierungstechniken, von der Visualisierung statischer Aspekte zu einem bestimmten Zeitpunkt, bis hin zu der Visualisierung von Metriken. Doch gleichzeitig betonen die oben Genannten: „Note that we are not aware of any visualization technique that focuses on visualizing how relationships change across software versions.“ . Die Aussage stellt für diese Arbeit eine besondere Bedeutung dar, da die Visualisierung der Architekturänderung als Zeitraffer ausschlaggebend für diese Arbeit ist.
Holten und Wijk [27] stellten eine alternative Visualisierungstechnik vor um die Evolution einer Architektur zu visualisieren. Darin wird die Hierarchie zweier Versionen einer Software gegenüber gestellt, siehe Abbildung 3.8a. Damit ist ein Vergleich sehr leicht ablesbar, doch werden damit immer noch „ nur“ zwei Versionen abgebildet und verglichen. Doch die gesamte Historie lässt sich auch mit dieser Visualisierungstechnik nicht darstellen. Eine weitere alternative Darstellung ist die Evolution einer Software in Form einer Matrix [34], siehe Abbildung 3.8b.
Allerdings konnte bei den durchgeführten Recherchearbeiten keine Visualisierungstechnik ausfindig gemacht werden, mit der die Evolution der Softwarearchitektur in einer Art Zeitraffer dargestellt wird.




(a) (b)

Abbildung 3.8: (a) Gegenüberstellung zweier Versionen einer Architektur [27] (b) Matrixdarstellung [34]

## 3.6 Vorhandene Visualisierungstools

Allgemein fiel bei Recherchearbeiten auf, dass es nur eine ganz geringe Anzahl an frei-verfügbaren Visualisierungstools gibt, die direkt auf das eigene Projekt angewendet werden können. Beim Großteil handelt es sich um akademische Lösungen oder Veröffentlichungen, in denen ausschließlich ein Prototyp entwickelt wurde und somit nicht in der Praxis einsatzbereit sind.
Leonel Merino et al. [40] veröffentlichte eine Studie, in der eine Auswahl an Visualisierungspaper, welche auf der SOFTVIS- oder VISSOFT-Konferenz veröffentlicht wurden, kritisch begutachtet wurden. Dabei war ein häufiger Kritikpunkt die fehlende Evaluation, wie anhand der Abbildung 3.9 erkennbar. Zu sehen ist, dass nur wenige bis gar keine Veröffentlichungen eine aussagekräftige Evaluierung enthalten. Vorrangig beschäftigen sich Paper mit Designstudien oder einem bestimmten System. Es bleibt also weiterhin die Frage, wie nutzbringend die bisher veröffentlichten Ergebnisse sind.

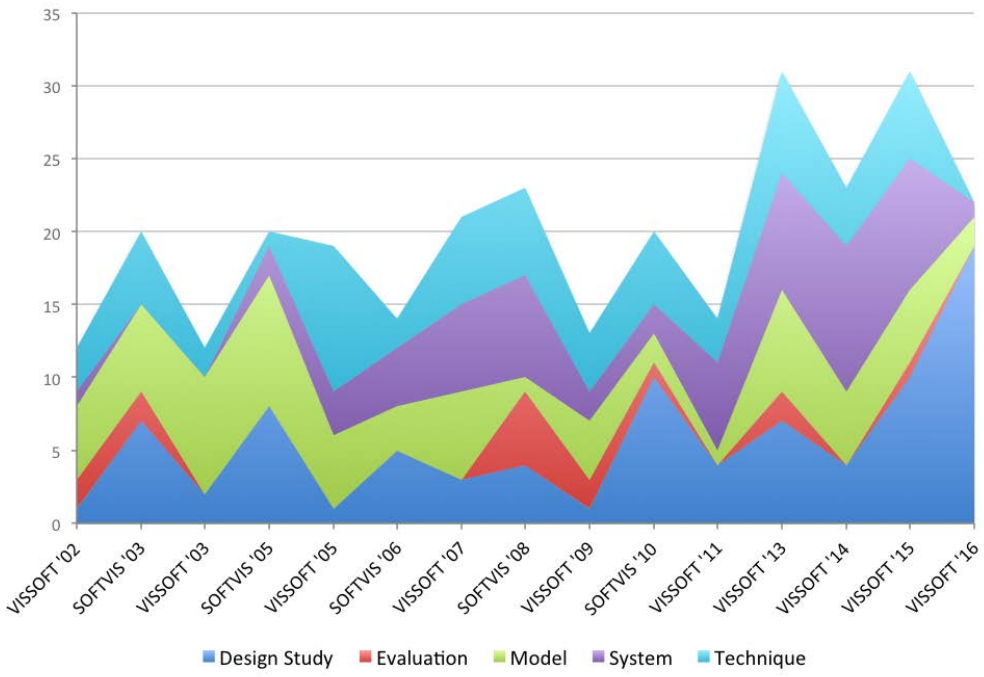



Abbildung 3.9: Anzahl der Paper nach Typ sortiert [40]

Das einzige Programm, welches Open Source verfügbar ist und auf das eigene Projekt angewendet werden kann, ist das Visualisierungstool Gource, siehe Abbildung 3.10 [2]. Doch für den Einsatz in dieser Arbeit bietet die Visualisierung mit Gource keinen großen Mehrwert. Das Ergebnis ist eine Video, welches eine geringe Aussage über den Inhalt

der zu analysierenden Daten vermittelt und keinerlei Interaktionsmöglichkeiten bietet.

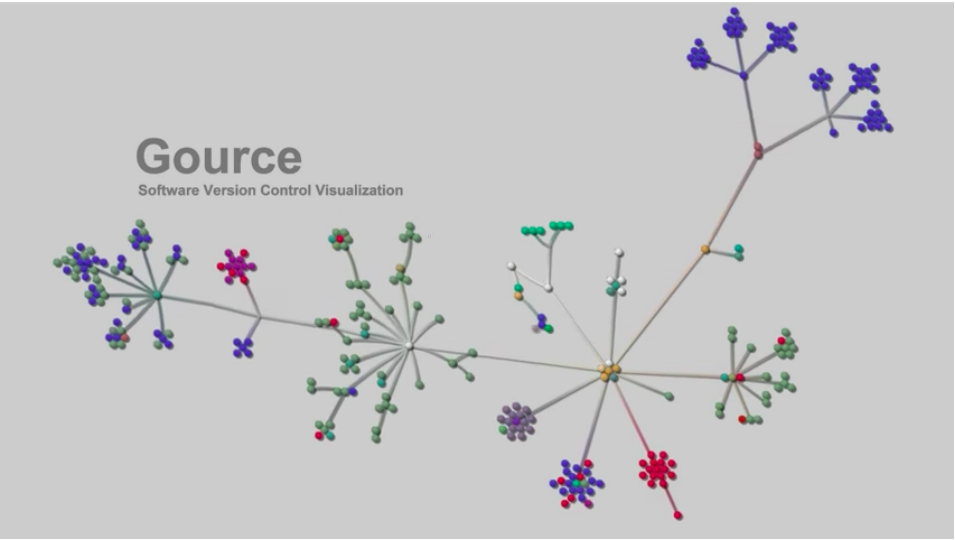



Abbildung 3.10: Visualisierung mit Gource

Einzig und allein die kommerzielle Lösung *KeyLines*[1] bietet die Möglichkeit, mit Hilfe eigener Daten individuell eine Visualisierung umzusetzen. Auf KeyLines wird im Kapitel 5.3 genauer eingegangen.

[1] `https://cambridge-intelligence.com/keylines/`

# 4 Konzept - From the Model to the Visualization

Im folgenden Kapitel wird das Konzept erläutert, welches als Grundlage für die im nächsten Kapitel beschriebene Umsetzung dient. Anhand der in der Einführung dargelegten Beschreibung dieser Arbeit wird der Fokus der Arbeit genauer erklärt. Die Zielgruppe wird eingeschränkt und Nutzungsszenarien definiert und anschließend wird das zur Verfügung stehende Datenmodell präsentiert. Zum Schluss des Kapitels wird der Visualisierungsansatz mit der angestrebten visuellen Metapher und ein erster Überblick des Implementierungsansatzes gegeben.

## 4.1 Zielgruppe

Um einen Nutzen aus der Visualisierung ziehen zu können, müssen sowohl die Ziele als auch die Zielgruppe klar definiert sein.
Die Zielgruppe kann von Softwarearchitekten und Entwicklern bis hin zu Managern und Kunden reichen. Da Entwickler und Architekten andere Kenntnisse und Erwartungen an die Visualisierung haben wie Manager und Kunden, ist es selten möglich, allen gleichermaßen gerecht zu werden. Deshalb ist an dieser Stelle eine Eingrenzung der Zielgruppe wichtig.
Die primäre Zielgruppe stellen in dieser Arbeit Softwarearchitekten und Entwickler dar, welche aktiv an der Entwicklung der Software beteiligt sind. Die Visualisierung der Architektur OSGi-basierender Systeme soll Architekten dabei unterstützen, bestehende Strukturen zu erkennen, zu verstehen und gegebenenfalls zu überdenken. Besonders im Hinblick auf Attribute wie Komplexität, Kopplung und Kohäsion, siehe Kapitel 2.6.
Entwickler sollen einen Überblick über die Softwarearchitektur und den Entwicklungsprozess bekommen, da dieser besonders bei großen und langjährigen Projekten wichtig ist. Doch nicht nur Entwickler, die aktiv an der Entwicklung beteiligt sind, können von der Visualisierung profitieren. Auch neue Mitarbeiter oder neue Teammitglieder, die in den Entwicklungsprozess integriert werden sollen, müssen sich zunächst mit der Software und deren Aufbau auseinandersetzen. Hier liegt der Fokus weniger auf dem Erkennen von Indikatoren, als viel mehr auf dem Verständnis, welches aus der Analyse gezogen werden kann. Zunächst soll die Visualisierung als eine gute Informations-Grundlage für Architekten und Entwickler dienen, mit der Möglichkeit immer die aktuellste Dokumentation vorliegen zu haben.

Zu der primären Zielgruppe der Architekten und Entwickler ist noch eine weitere Zielgruppe zu berücksichtigen. Manager wollen in der Regel das System nur auf einer hohen Ebene verstehen und einen Überblick über den Fortschritt des Systems bekommen. Die Visualisierung kann ihnen diesen Einblick gewähren. Forscher werden an dieser Stelle nur der Vollständigkeit halber als weitere Zielgruppe erwähnt. Durch die Forschung im Bereich der Nachvollziehbarkeit und Analyse von Softwareprozessen könne sie neue Trends erkennen und optimieren.

### 4.1.1 Nutzungsszenarien

Im folgenden Kapitel werden Nutzungsszenarien für die Visualisierung der Evolution einer Softwarearchitektur aufgelistet.

**Dokumentation**

Mithilfe des Tools soll es möglich sein, eine lückenlose und aktuelle Dokumentation der Architektur jederzeit zur Verfügung zu haben. Eine bessere Einsicht, wie neue Anforderungen in die bestehende Softwarearchitektur integrierbar sind, wird gewährleistet.
Somit kann Zeit- und Arbeitsaufwand minimiert werden, indem eine effiziente Sicht der Softwarearchitektur geschaffen wird und die Kosten durch die Verwendung der Visualisierung reduziert werden.

**Rückverfolgung**

Die Visualisierung der Evolution kann dazu beitragen, die bisher angehäuften technischen Schulden zu identifizieren und der weiteren Erosion der Architektur Schritt für Schritt entgegenzuwirken, wieder in den Korridor der geringen technischen Schulden zurückzubringen.
Der Begriff der technischen Schulden wurde 1992 von Ward Cunningham geprägt und besagt, dass bewusst oder unbewusst falsche oder suboptimale Entscheidungen getroffen wurden, die nicht mehr der, zu Beginn des Projektes festgelegten, Strukturvorgaben des Softwaresystems entsprechen [36].

**Qualitätsbeurteilung**

Das Analysetool kann Nutzer unterstützen, Qualitätsmerkmale von bereits bestehenden Systemen zu beurteilen und zu prüfen, ob alle Anforderungen erfüllt wurden. Qualitätsmerkmale können in diesem Zusammenhang Wartbarkeit, Modifizierbarkeit, Effizienz und Wiederverwendbarkeit sein.

## 4.2 Datenbankmodell

Bestandteil dieser Arbeit ist die Anwendung architekturrelevanter Kriterien und die Umsetzung einer visuellen Darstellung der Evolution einer Softwarearchitektur. Diesen

beiden Vorgängen liegt eine Graphdatenbank zugrunde, deren Aufbau im folgenden Kapitel beschrieben wird, um für die weitere Umsetzung ein grundlegendes Verständnis vom Aufbau der Daten zu bekommen.

In der Abbildung 4.1 ist die Gesamtansicht des Datenbankmodells dargestellt. Bei der Konzeption des Datenbankmodells musste abgewägt werden, wie umfassend und feingranular das Modell aufgebaut sein soll. Auf der einen Seite ist es von Vorteil, möglichst viele Architekturkomponenten abzuspeichern, um somit einen vielseitigen Einsatz der Analyse zu gewährleisten. Auf der anderen Seite ist besonders im Hinblick auf die darauf aufbauende Visualisierung eine zu feingranulare Speicherung der Daten unvorteilhaft. Da durch die Visualisierung zu kleiner Komponenten schnell der Überblick verloren gehen kann und somit nicht mehr der Zielsetzung dieser Arbeit entsprochen werden kann - mit Hilfe der Visualisierung einen Überblick und ein besseres Verständnis zu schaffen. Es ist zu erwähnen, dass die dynamischen Eigenschaften von OSGi an dieser Stelle nicht weiter berücksichtigt werden. Es handelt sich somit um eine statische Sicht auf die zu analysierende OSGi-Anwendung.

Das Modell lässt sich in drei grobe Teilbereiche gliedern, welche in den nächsten Abschnitten genauer erläutert werden:

- Historie
- Bundles und Packages
- Services

### Historie

In der Graphdatenbank ist die komplette Historie einer Softwarearchitektur abgespeichert. Durch die Abbildung der Historie im Modell entsteht eine große Datenmenge. Um trotz dieses Umstandes eine übersichtliche Datenbankstruktur zu schaffen und eine Unterscheidung der einzelnen Versionen der Architektur ausmachen zu können, wurde eine weitere Entität eingeführt, bei der es sich um keine Softwarearchitekturkomponente handelt. Das Versionskontrollsystem wird durch die Entität CommitID modelliert. Unter dieser Entität wird der Stand der Softwarearchitektur zu einem bestimmten Zeitpunkt abgespeichert. Somit ist es möglich, eine Unterscheidung der Architektur zwischen den einzelnen Versionen vorzunehmen. Durch die gerichtete Beziehung *HAS* von der Entität CommitID zu Bundle, können alle zu einem bestimmten Commit gehörenden Bundles bestimmt werden, siehe Abbildung 4.2. Dabei werden nur die Komponenten berücksichtigt, die in dem jeweiligen Commit existieren. Gelöschte Komponenten fallen somit weg.
Darüber hinaus wird zu jedem Commit der Autor als Attribut mit gespeichert, um zum

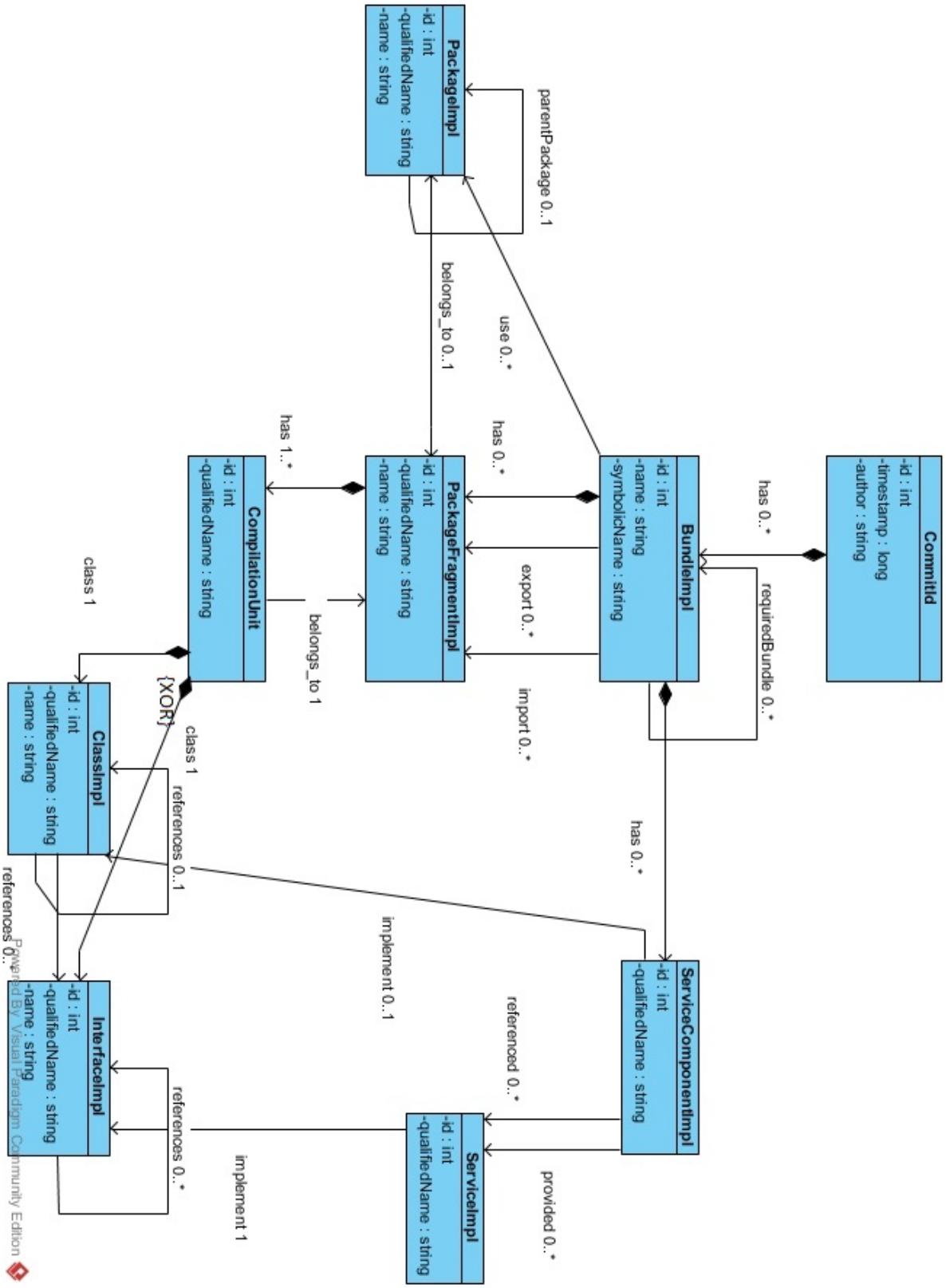



Abbildung 4.1: Datenbankmodell

Beispiel in darauf aufbauenden Analysen und bei der Visualisierung eine Aussage über die Wissensverteilung im Team treffen zu können.

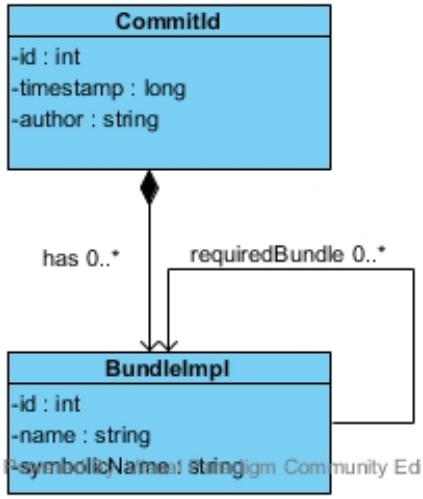



Abbildung 4.2: Modellierung der Historie im Datenbankmodell

### Bundles und Packages

OSGi-Bundles werden als Entität Bundle modelliert. In Kapitel 2.5.2 wurde beschrieben, dass bei der Datenbankmodellierung berücksichtigt werden muss, dass jedes Bundle einen eigenen Classloader besitzt. Ein bestimmtes Package kann somit gleichzeitig in mehreren Bundles vorhanden sein, siehe Kapitel 2.5.2. Das gleiche gilt für Compilation Units. Daher wurde die Unterscheidung zwischen den Entitäten Package und Package-Fragment eingeführt, siehe Abbildung 4.3. Damit soll ersichtlich werden, ob ein Package in mehreren Bundles verwendet wird (fragmentiert).
Ein Java-Package muss streng genommen immer eine Compilation Unit enthalten. Somit stellt die konkrete Ausprägung eines Packages in einem Bundle die Entität Package-Fragment dar. Die Entität Package dagegen repräsentiert den Namensraum und die zugrunde liegenden Hierarchieverhältnisse, durch die Beziehung *parentPackage* und *subPackage*. Die Hierarchie ist ausschließlich für die Umsetzung der Visualisierung nützlich. Zwischen den Entitäten Package und Package-Fragment besteht eine bidirektionale Beziehung. Ein Package, welches in einem Bundle vorhanden ist und mindestens eine Compilation Unit enthält, verweist auf genau ein Package-Fragment. Existiert dieses Package jedoch in mehreren Bundles, so verweist es entsprechend auf mehrere Package-Fragmente. Ein Package-Fragment kann mehrere Compilation Units enthalten. Diese definieren wiederum eine Klasse oder ein Interface.

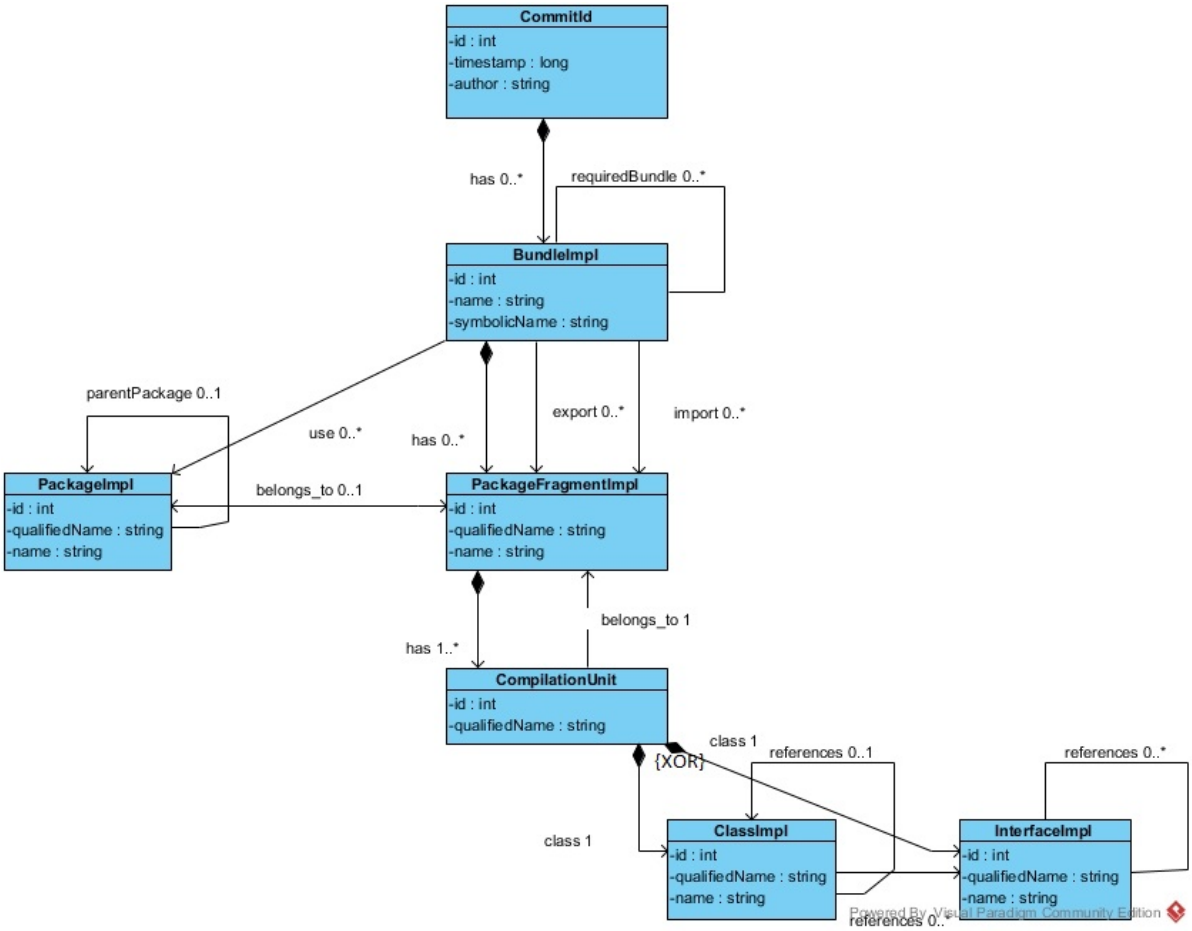



Abbildung 4.3: Modellierung von Packages und Package Fragmenten

### Services

Eine Service Component repäsentiert eine Service-Deklaration durch eine XML-Datei innerhalb eines Bundle, siehe Kapitel 2.5.2. Diese werden durch die Entität Service Component modelliert und durch eine bestimmte Klasse implementiert, siehe Abbildung 4.4. Eine Service Component kann Services zur Verfügung stellen (provide) und benutzen (reference). Bei einem Service handelt es sich letztendlich um ein Java-Interface, welches durch die Entitäten Service und Interface modelliert sind.

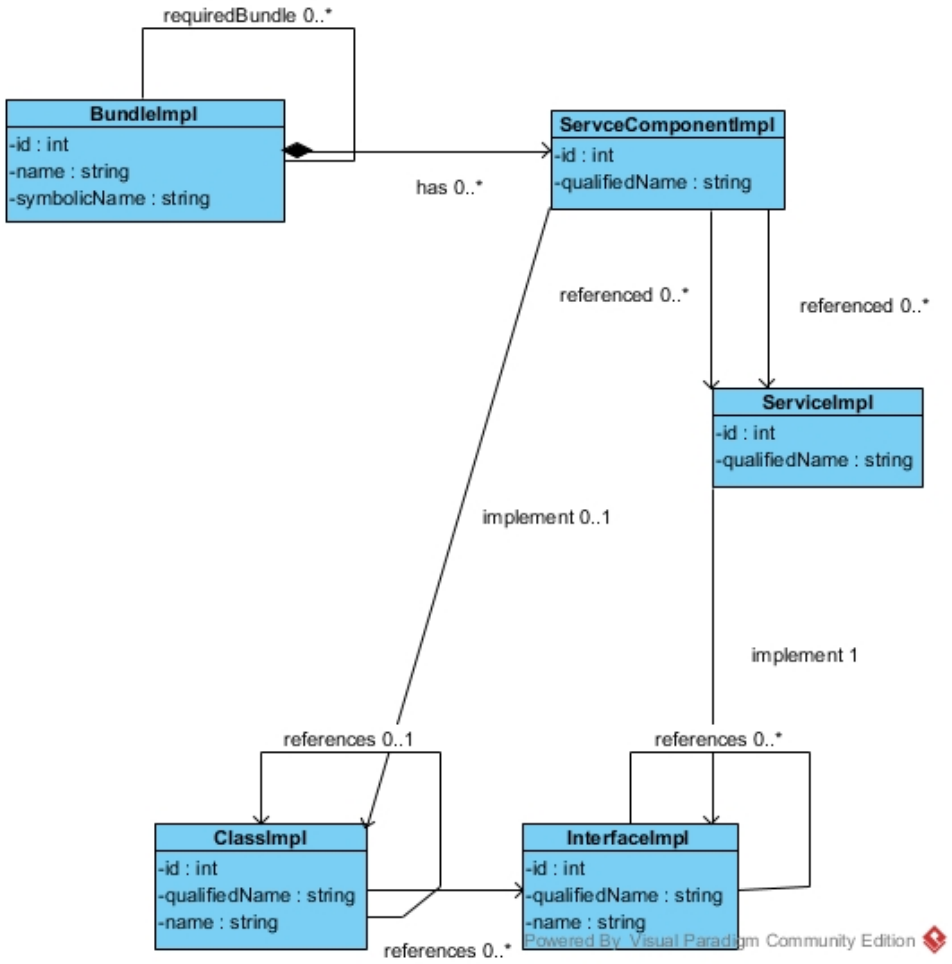



Abbildung 4.4: Modellierung der Services

In der eben beschriebenen Graphdatenbank sind alle Daten enthalten, um das folgende Konzept für die Visualisierung umsetzen zu können.

## 4.3 Visualisierungsansatz

Aufgrund der in Kapitel 4.1 festgelegten Zielgruppe und Nutzungsszenarien sollen als Basis für die Visualisierung ausschließlich die internen Attribute einer Software berücksichtigt werden, da sie für Softwarearchitekten und Entwickler einen größeren Mehrwert, im Gegensatz zu den externen Attributen, darstellen, siehe Kapitel 2.6.1.
Ein wichtiges internes Attribut ist die Modularität. Da das Hauptfeature des OSGi-Frameworks die komponenten- und serviceorientierte Entwicklung ist, also die Modula-

risierung, sollte der Fokus auf den Komponenten und deren Beziehungen liegen. Denn Modularisierung bedeutet auch immer, dass Module in Beziehung zueinander stehen.
Weitere Metriken wie Kopplung und Kohäsion können mit einbezogen werden. Jedoch liegt der Fokus auf der Visualisierung der gesamten Software und nicht auf der Präsentation von Metriken. Sie soll lediglich eine Unterstützung darstellen, um Schwachstellen selber ausfindig zu machen und nicht dem automatischen Erkennen und Präsentieren solcher eben genannten Stellen zu dienen (siehe Kapitel 4.1). Mithilfe des gezielten Einsatzes von Interaktionsmethoden, siehe Kapitel 3.1.1, kann der Betrachter beim Erkennen der eben genannten Metriken unterstützt werden.
Dennoch darf das Ziel nicht sein, so viele Informationen wie möglich in die Visualisierung zu integrieren. Die wesentlichen Informationen sollen geschickt zusammengefasst und anschließend zielgruppengerecht visuell aufbereitet werden.
Da in dieser Arbeit nicht der Stand zu einem bestimmten Zeitpunkt, sondern die Historie der zu analysierenden Software, betrachtet werden soll, kann es sich bei der visuellen Metapher keinesfalls um ein statisches Bild handeln. Auch mithilfe einer Darstellung der gesamten Historie in einem statischen Bild kann die Abfolge von Architekturänderungen, während des gesamten Entwicklungsprozesses, nicht aussagekräftig umgesetzt werden. Viel mehr soll in einer Art Zeitraffer durchgehend die gesamte Architektur dargestellt werden, in der sich ausschließlich Stellen, an denen eine Architekturänderung stattgefunden hat, verändern.

Bei der Erarbeitung eines wirkungsvollen Mapping-Schema soll die Beantwortung der von Ghanam et al. [23] empfohlenen Fragen als Unterstützung dienen.

**Wer** gehöhrt zu der Zielgruppe?

**Was** soll durch die Visualisierung beantwortet werden?

**Wie** können visuelle Metaphern genutzt werden, um die gestellten Fragen zu beantworten?

Denn die sorgfältige Ausarbeitung einer zielgruppengerechten Visualisierung ist von besonderer Wichtigkeit im Bereich der Softwarevisualisierung. Forscher stellten fest, dass einer der Gründe warum Softwarevisualisierungen in der Praxis immer noch sehr wenig eingesetzt werden, die Differenz zwischen den Ergebnissen der Visualisierung und den Bedürfnissen der Benutzer ist [40].

#### Wer?

McNair et al. [38] definierten drei verschiedene Zielgruppen. Zum einen Entwickler, welche mit Hilfe der Visualisierung eine Übersicht über den Aufbau des Systems erhalten wollen und somit ein besseres Verständnis der Software. Zum anderen Forscher, welche

mögliche Trends und Erkenntnisse erkennen und Manager, welche den Fortschritt des Projektes überwachen wollen.
Wie in Kapitel 4.1 bereits ausführlich definiert, gehören zu der primären Zielgruppe dieser Arbeit ausschließlich Softwarearchitekten und Entwickler. Welche andere Erkenntnisse aus der Visualisierung ziehen wollen wie zum Beispiel Manager und Forscher. [44].

### Was?

Priya et al. [50] fanden in ihrer Arbeit heraus, dass die an einem Softwareprojekt beteiligten Personengruppen, je nach Grad der Beteiligung, unterschiedliche Interessen bezüglich des Informationsgehalts eines Projektes haben, siehe Abbildung 4.5. Dies muss in der Visualisierung berücksichtigt werden. Denn wie anhand der Abbildung 4.5 zu erkennen ist, hat ein Software-Architekt ein größeres Interesse an einem hohen Detailgrad der Visualisierung als Manager oder Kunden. Architekten wollen einen Überblick über die gesamte Software erhalten und gewisse Charakteristika und Eigenschaften, wie Modularität, Komplexität oder Kopplung mithilfe der Visualisierung beurteilen.
Des Weiteren ist anhand der Abbildung zu erkennen, dass es selbst zwischen dem Architekten und dem Entwickler Interessensunterschiede gibt, wobei beide Personengruppen direkt an der Softwareentwicklung involviert sind. Während der Architekt eine Gesamtübersicht erhalten möchte um die Architektur betrachten zu können, ist der Entwickler an einer noch tiefgründigeren Visualisierung interessiert. Für den Entwickler liegt der Fokus auf Source-Code-Ebene, dabei spielen Metriken wie LOC, siehe Kapitel 2.6.1, eine große Rolle.
Eine Visualisierung auf Source-Code-Ebene wird nicht angestrebt, da der Fokus der Arbeit auf der Erstellung einer Visualisierung von Architekturänderungen über den gesamten Entwicklungsprozess hinweg liegt und somit eine Betrachtung auf höherer Abstraktionsebene nötig ist.

Auf Grundlage der, in der Graphdatenbank enthaltenen, Daten, siehe Kapitel 4.2, soll im Folgenden genauer beschrieben werden, welche Aspekte im Hinblick auf die Visualisierung von Interesse sind. Dabei liefert die Tabelle 4.1 einen ersten Überblick.

**Historie** Bei einer Visualisierung der gesamten Softwarearchitektur über den gesamten Lebenszyklus hinweg können einzelne Veränderungen in Bezug auf Komponenten und deren Beziehungen und Abhängigkeiten deutlich herauskristallisiert werden. Nicht die Architektur der einzelnen Commits soll mithilfe einer Visualisierung dargestellt werden, sondern die Veränderungen der Architektur die über den gesamten Entwicklungszeitraum stattgefunden haben. Schlussendlich soll in der Visualisierung durchgehend die gesamte Architektur dargestellt werden, ausschließlich die Stellen mit architekturrelevanten Änderungen sollen in der Visualisierung angepasst werden.
Hotspots können identifiziert werden, zu welchem Zeitraum besonders viele

| | Eigenschaften | Implementierung |
|---|---|---|
| Historie | Evolutionäre Abhängigkeiten<br>Architekturrelevante Veränderungen<br>Hotspots | ✓<br>✓<br>✓ |
| Abhängigkeiten | Abhängigkeiten zw Klassen<br>Abhängigkeiten zw Packages<br>Abhängigkeiten zw Bundles<br>Art der Abhängigkeiten | ✓<br>✓<br>✓<br>✓ |
| Modularität | Exports und Imports<br>interner und öffentlicher genutzter Code | ✓<br>✓ |
| Services | Services<br>Service-Nutzung<br>Service-Implementierung | ✓ (provide)<br>✓ (reference)<br>✓ (implements) |

Tabelle 4.1: Eigenschaften, die auf Grundlage der vorhandenen Daten dargestellt werden können. Dabei steht ✓für umgesetzt.

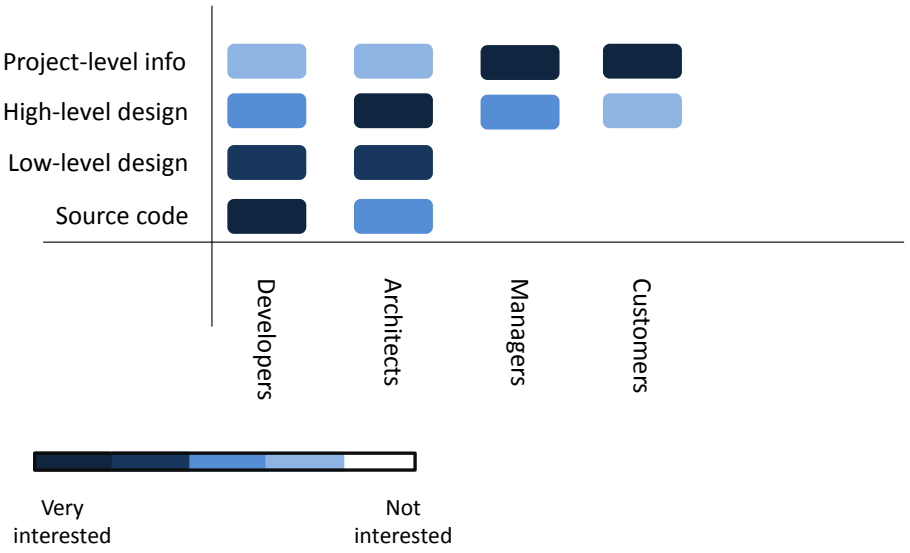



Abbildung 4.5: Verschiedene Ebenen des Interesses an Software Visualisierung - Quelle suchen

Änderungen stattgefunden haben. Diese Information kann als Indikator genutzt werden, um mit Hilfe weiterer Informationen nach der Ursache zu suchen und somit den Entwicklungsprozess zu optimieren, siehe Kapitel 2.1. Zum Beispiel kann ein Hotspot in der Maintenance-Phase ein Indikator dafür sein, dass die Komponente viele Fehler enthält. Weshalb es sinnvoll sein würde, weitere Test zu erstellen oder die Architektur zu überarbeiten. Mehrere Hotspots, die sich zusammen ändern, können ein Zeichen für instabilen Code sein. In einem gut durchdachten System wird erwartet, dass der Code im Laufe der Zeit stabiler wird. Doch wenn die Module instabil sind, beruht das auf Versagen einer geringen Kohäsion. Die einzelnen Subsysteme ändern sich laufend, weil zu viele Verantwortlichkeiten dazu führen.
Darüber hinaus wäre die zeitlich Einordnung in Entwicklungsphasen als Erweiterung wünschenswert.

**Abhängigkeiten** Im Fokus der Visualisierung steht die Modularität, das Hauptfeature des OSGi-Frameworks, siehe 4.3. Daher sollen die einzelnen Komponenten und de-

ren Abhängigkeiten auf unterschiedlichen Abstraktionsschichten visualisiert werden, auf Klassen-, Package- und Bundleebene.

**Modularität** Komponenten können drei verschiedene Arten von Beziehungen eingehen:

- Beziehungen zwischen internen Komponenten
- Eingehende Beziehungen
- Ausgehende Beziehungen

Mit Hilfe der Darstellung der Komponenten und den unterschiedlichen Beziehungen, lassen sich Metriken wie Kopplung und Kohäsion ableiten.
Des Weiteren kann es von Interesse sein, wie viel Code öffentlich, das heißt exportiert wurde und wie viel Code intern gekapselt ist. Die Manifest-Datei bietet durch die Import- und Export-Package-Deklaration, auf Packageebene, alle dafür notwendigen Informationen. Dies ermöglicht zusätzliche Informationen über die Modulstruktur.

**Services** Services spielen eine besondere Rolle in OSGi-Anwendungen, siehe Kapitel 2.5.2. Bei der Darstellung dieser, sollen die drei verschiedenen Arten von Beziehungen zwischen Bundle und Services berücksichtigt werden: Bereitstellung, Implementieren und Referenzieren.

### Wie?

Die Visualisierungstechnik einer Software kann durch folgende drei Aspekte klassifiziert werden: Anzahl der Sichten, Dimensionen und die Wahl der Metapher. Alle drei Aspekte können der Klärung dienen, wie die Visualisierung umgesetzt werden soll.

**Anzahl der Sichten** In der Forschung werden zwei unterschiedliche Ansätze im Umgang mit der Anzahl der Sichten verfolgt. Auf der einen Seite wird die Meinung vertreten, dass die Umsetzung mehrerer Sichten von Vorteil ist. Denn mit der Unterstützung mehrerer Sichten sollen mit einer Visualisierung gleichzeitig mehrere Zielgruppen mit unterschiedlichen Anforderungen angesprochen werden, der sogenannte Multi-View-Ansatz. Auf der anderen Seite wird die Meinung vertreten, dass es sinnvoller und effektiver ist, eine sorgfältig ausgearbeitete Einzelansicht zu implementieren [23].
Für die Darstellung von Architekturen wurde in der internationalen Norm IEEE 42010 [31] festgelegt, dass eine Architekturbeschreibung aus mehreren Architekturansichten bestehen soll, wenn die Darstellung mehr wie eine Problemstellung behandelt. Denn eine zu beantwortende Problemstellung soll genau mit einer Ansicht dargestellt werden. Da die Visualisierung in dieser Arbeit ebenfalls mehr wie eine Problemstellung beantwortet, ist es von Vorteil den Multi-View-Ansatz für die Umsetzung zu verfolgen, siehe Kapitel 4.3.

**Dimensionen** Durch die Komplexität der Daten und der Menge an Knoten und Kanten ist es sinnvoll eine 3D-Visualisierung in Betracht zu ziehen, da auf den ersten Blick somit viele Informationen eines großen und langjährigen Softwareprojektes mit einer komplexen Architektur gut darstellbar sind. Doch die Übersicht kann bei einer 3D-Visualisierung auch schnell verloren gehen.
Zudem kann die Verwendung von drei Dimensionen zu extremen Informationsverlust führen. Denn nicht immer sind alle Elemente im Modell, an denen eine Veränderung stattfindet, mit Hilfe eines Blickwinkels darstellbar. Für die Umsetzung der Visualisierung bedeutet dies, dass nicht immer alle architekturrelevanten Änderungen für den Betrachter auf einem Blick ersichtlich sind. Mit dem gezielten Einsatz von Interaktionsmöglichkeiten, wie der entsprechenden Navigation, könnte dem Problem Abhilfe geschaffen werden. Doch würde dies implizieren, dass der Betrachter nach jeder Anpassung der Visualisierung die Darstellung nach Architekturänderungen absuchen müsste. Das Hervorheben der entsprechenden Komponenten und Beziehungen könnte eine Unterstützung bei der Suche darstellen, trotzdem müsste der Betrachter wertvolle Zeit mit der Suche hervorgehobene Stellen verbringen. Dies würde bei großen und langlebigen Systemen ein großer Aufwand für den Betrachter darstellen.
Ziel dieser Arbeit ist es, dem Betrachter auf einen Blick die Änderungen der Architektur über den gesamten Lebenszyklus übersichtlich und verständlich darzustellen. Daher wird in dieser Arbeit die Visualisierung in 2-D erfolgen.

**Metapher** Bei der Wahl der visuellen Metaphern wurden bisher häufig geometrische Formen wie Kugeln oder Vierecke verwendet. Doch inzwischen wird bei der Visualisierung von Softwarearchitekturen vermehrt auf reale Metaphern zurückgegriffen. Im folgenden Kapitel 4.3.1 soll genauer auf die Auswahl der Metapher eingegangen werden.

### 4.3.1 Visuelle Metapher

Basierend auf den zur Verfügung stehenden Basisdaten und den, in Kapitel 4.1, definierten Stakeholdern wurde eine geeignete Darstellungsform für die Visualisierung ausgewählt. Diese bildete alle für den Betrachter interessanten Software-Komponenten ab, siehe Abschnitt 4.3, und stellt folgende Aspekte aussagekräftig dar:

1. Abhängigkeiten zwischen Bundles und Packages
2. Package- und Klassenstruktur
3. Service-Components und ihre Beziehungen (providing/ referencing)
4. Evolution der Architektur

OSGi ist ein sehr reichhaltiges Komponenten-Framework, daher werden reale Metaphern als Darstellungsart, für die Visualisierung großer und langlebiger Softwareprojekte, nicht in Betracht gezogen. Zum einen wurden bisher keine aussagekräftigen Evaluationen durchgeführt, die den Beweis erbracht haben, dass Entwickler und Architekten besser die Architektur eines System mithilfe realer Metaphern aufnehmen können. Zum anderen wirkt eine reale Metapher bei der großen Anzahl an Komponenten und Beziehungen sehr schnell überladen. Sowohl die City-Metapher als auch das Sonnensystem können nicht genug Elemente bereitstellen, um alle Komponenten, Beziehungen und Abhängigkeiten sinnvoll und strukturiert darzustellen (siehe Kapitel 3.5.1). Besonders die zeitliche Dimension lässt sich in diese visuellen Metaphern nicht sinnvoll integrieren.
Stattdessen soll durch die ausgewählte Darstellungsart gewährleistet werden, dass die Software auf verschiedenen Abstraktionsebenen (Levels-of-Detail), von Bundles über Packages und Klassen bis hin zu Services, betrachtet werden kann. Metriken und Eigenschaften, wie zum Beispiel die Größe der Komponenten oder die Stärke von Abhängigkeiten, sollen durch Anpassung der grafischen Variablen in die Darstellung einbezogen werden.

### Abhängigkeiten zwischen Bundles

Um Elemente der gleichen Informationskomponente und ihre Beziehungen untereinander darzustellen eignet sich eine Netzdarstellung. Ein gerichteter Graph mit Bundles als Knoten und Abhängigkeiten als gerichtete Kanten, eignet sich gut um die Abhängigkeiten zwischen Bundles zu visualisieren, siehe Abbildung 4.6. Neben der Darstellung der Komponenten und deren Abhängigkeiten mithilfe eines Graphen, können durch den Einsatz grafischer Variablen zusätzliche Informationen mit in die Visualisierung integriert werden. Durch die Größe der Knoten soll dargestellt werden, wie groß die Komponenten im Verhältnis zu den anderen Komponenten sind und durch die Dicke der Kante soll dargestellt werden, wie Stark die Abhängigkeiten sind.

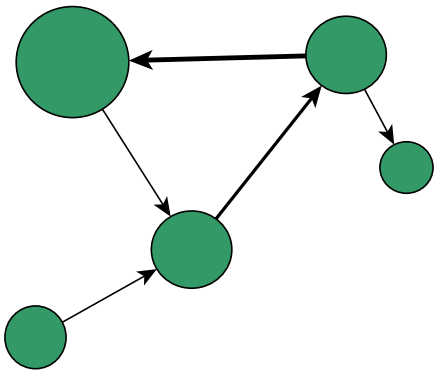


Abbildung 4.6: Ausgewählte Darstellungsart für Bundles

### Package- und Klassenstruktur

Für die Visualisierung der Packages und Klassen innerhalb eines Bundles sind vor allem die hierarchischen Strukturen und Größenverhältnisse von Interesse. Dafür eignet sich zum einen die Treemap, siehe Kapitel 3.5.1. Die Größe der einzelnen Blätter stellt die Größe der zugehörigen Klassen dar und durch die Farbe lassen sich die Packages zuordnen. Die Abhängigkeiten zwischen den Komponenten lassen sich durch das Hierarchical Edge Bundling darstellen, welches auch auf Treemaps anwendbar ist.
Doch das Wechseln der Darstellungsform innerhalb einer Visualisierung kann dazu führen, dass der Betrachter die Orientierung und Übersicht im Gesamtkontext verliert. Der Betrachter müsste sich zusätzlich zu der Informationsflut die er durch die Menge an Daten verarbeiten muss, noch zusätzlich während der Visualisierung in einer neue Darstellungsform einarbeiten und die Zusammenhänge neu interpretieren. Daher eignet sich, auf Grund der zuvor definierten Nutzungsszenarien (siehe Kapitel 4.1.1), ein Wechsel der Darstellungsform innerhalb einer Visualisierung nicht. Doch schon die Treemap an sich eignet sich nicht zur Darstellung der zeitlichen Veränderung der Architektur, da die zeitliche Dimension nicht in die Darstellungsform integrierbar ist.
Stattdessen soll auch an dieser Stelle ein gerichteter Graph eingesetzt werden und grafische Variablen dazu genutzt werden, zusätzliche Informationen anzuzeigen.

- Die Größe der Knoten soll die Größe der Komponenten darstellen.
- Durch die Farbe der Knoten sollen die unterschiedlichen Arten der Komponenten visualisiert werden.
- Die Dicke der Kanten soll die Stärke der Abhängigkeiten darstellen.
- Die Richtung der Kanten soll die Richtung der Abhängigkeiten visualisieren.

### Service-Components und ihre Beziehungen

Bei der Visualisierung der Service-Components und deren Beziehungen muss beachtet werden, dass an dieser Stelle zwischen zwei Arten von Komponenten unterschieden werden muss: Service-Components und Services. In diesem Fall soll ebenfalls ein gerichteter Graph als Darstellungsform dienen, wobei Service-Components und Services durch die Form der Knoten unterschieden werden sollen, siehe Abbildung 4.7. Bei der Visualisierung muss jedoch nicht nur zwischen zwei Arten von Komponenten unterschieden werden, sondern auch zwischen zwei Varianten von Beziehungen zwischen diesen Komponenten. Die Beziehungen *provide* und *reference* (siehe Kapitel 2.5.2) können durch die Form der Kanten dargestellt werden. Das Anbieten (provide) kann durch eine gestrichelte und die Benutzung (reference) durch eine durchgezogene Kante umgesetzt werden. Die Zuordnung zu den Bundles soll über die Farbe realisiert werden.

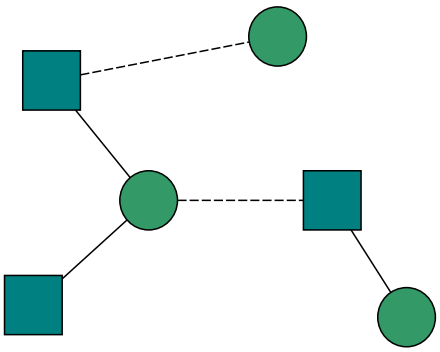


Abbildung 4.7: Ausgewählte Darstellungsart für Service-Components und Services

### Evolution der Architektur

Die Darstellung der Evolution stellt die größte Herausforderung bei der Umsetzung der Visualisierung dar. Denn durch die zeitliche Dimension nimmt der Informationsgehalt signifikant zu. Bisher ist keine Visualisierungstechnik bekannt mit der die gesamte zeitliche Veränderung einer Architektur in einer Art Zeitraffer umgesetzt wurde, siehe Abschnitt 3.5.2. Gegenstand dieser Arbeit ist nicht die Gegenüberstellung zweier Versionen einer Architektur oder die Gesamtansicht einer Architektur in der alle Änderungen abgebildet sind (siehe Kapitel 3.5.2), sondern die Visualisierung der zeitlichen Veränderung der gesamten Architektur.
Das soll in dieser Arbeit mithilfe eines Zeitraffers umgesetzt werden. Der Betrachter kann die Entwicklung der Architektur vom Anfang bis zum Ende des Entwicklungsprozesses in einer Art Video abspielen und dabei betrachten. Dabei sollen die Änderungen am Graphen nacheinander automatisch angepasst werden. Wurde zum Beispiel eine Komponente hinzugefügt soll in der Visualisierung ein neuer Knoten erkennbar sein und durch farbliche Hervorhebung dem Betrachter auf dem ersten Blick ersichtlich sein. Wurde dagegen eine Komponente gelöscht soll der dazugehörige Knoten in der Grafik ausgegraut werden, somit ist auf dem ersten Blick ersichtlich an welcher Stelle Komponenten gelöscht wurden. Würden die Knoten komplett aus der Grafik verschwinden, würde die Information bei der Menge an Knoten untergehen.
Durch zusätzliche Elemente, wie einer Timebar und einer Suchanzeige, sollen dem Betrachter zusätzliche Informationen und Interaktionsmöglichkeiten geboten werden. Mithilfe der Timebar am unteren Rand der Visualisierung soll die zeitliche Einordnung der im Bild dargestellten Architektur realisiert werden und durch Filterung und Selektion dem Betrachter die Möglichkeit geboten werden die Menge an Informationen zu reduzieren und auf die individuellen Bedürfnisse einzuschränken.

## 4.4 Implementierungsansatz

Die Implementierung soll in zwei überwiegend unabhängige Teile, die nacheinander ablaufen, aufgeteilt werden, siehe Abbildung 4.8. Zunächst werden die Daten des zu analysierenden Git-Projektes extrahiert, analysiert und in ein Modell transformiert. Dazu gehört auch die Einschränkung der analysierten Daten auf architekturrelevante Änderungen. Im zweiten Teil soll das Datenmodell durch eine interaktive Visualisierung dargestellt werden, siehe Kapitel 5.3.
Die Zweiteilung der Anwendung hat den Vorteil, dass sie unabhängig voneinander umgesetzt werden können. Dies kann hilfreich sein, da jeder Teil unterschiedliche technische Anforderungen hat. Somit können unterschiedliche Programmiersprachen und Laufzeitumgebungen eingesetzt werden. Des Weiteren können sie alleinstehend ausgeführt werden. Während die Analyse auf einem Server durchgeführt wird, kann die Präsentation der Visualisierung separat auf einem Client erfolgen. Durch die Aufteilung besteht auch keine direkte Abhängigkeit zwischen der Analyse und Visualisierung, denn beide Teile greifen ausschließlich auf das Datenmodell zu. Wodurch das Datenmodell als einzige Schnittstelle zwischen beiden Teilen fungiert. Somit wird der Wartungsaufwand verringert und die Möglichkeit verbessert, den Analyseteil jederzeit durch weitere Features zu erweitern oder andere Visualisierungstechniken auf Basis der gespeicherten Daten anzuwenden.

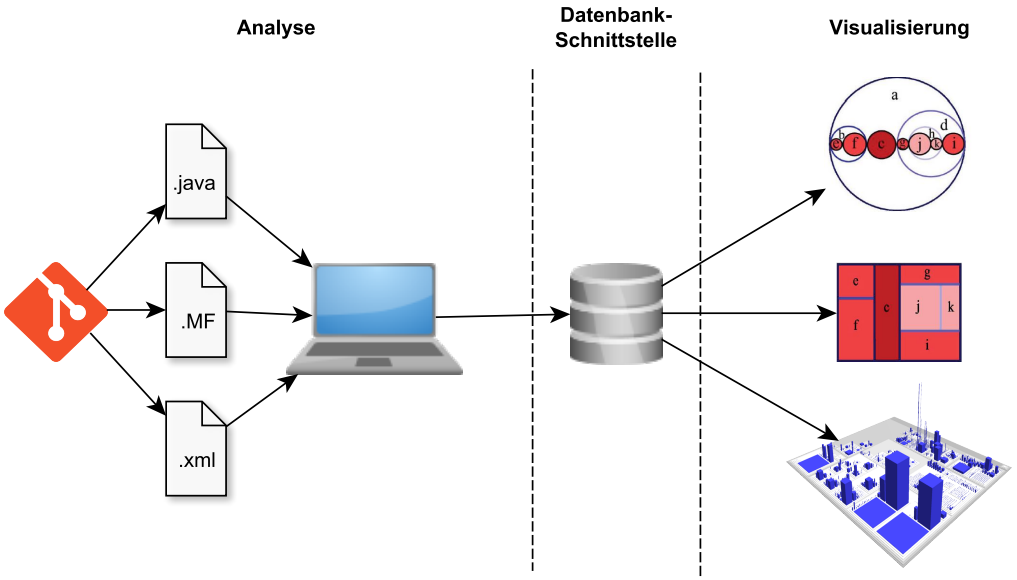



Abbildung 4.8: grober Ablaufplan der Applikation

# 5 Umsetzung - Visualize the architecture

In diesem Kapitel wird die Umsetzung erläutert, auf Grundlage des vorgestellten Konzeptes in Kapitel 4. Die Implementierung lässt sich in zwei separate Teile unterteilen: die Analyse und die Visualisierung. Es wird jeweils der Aufbau der Implementierung vorgestellt, technische Entscheidungen begründet und an ausgewählten Stellen detaillierter auf die Umsetzung und Herausforderungen eingegangen. Da die Realisierung aller Ideen den Rahmen dieser Arbeit sprengen würde, musste besonders im Bezug auf die Visualisierung ein Fokus gesetzt werden. Doch auf Basis des Konzeptes ist in Zukunft eine Erweiterung der Implementierung möglich.

## 5.1 Extraktion

Als Grundlage für diese Arbeit dient die Projektarbeit „Effiziente Verarbeitung von historisierten Software-Entwicklungsergebnissen“. Ziel dieser Arbeit war es, die Evolution der Softwarearchitektur mithilfe eines iterativen Vorgehens aus einem Git-Repository zu extrahieren und in einer Graphdatenbank abzuspeichern. Im Folgenden sollen die für diese Arbeit relevanten Ergebnisse des Analyseteils beschrieben werden.

### 5.1.1 Aufbau

Die Abbildung 5.1 stellt den Workflow des Analyseteils dar, welcher die Extraktion der Daten und die Beschränkung der Daten auf die architekturrelevanten Änderungen beinhaltet. Als Datenquelle wurde das Versionsverwaltungsprogramm *Git* verwendet. Ein Git-Repository enthält den kompletten historischen Verlauf einer Software und somit alle, für diese Arbeit, relevanten Daten.
Vor der ersten Durchführung müssen in der analysis- und config-Datei Angaben hinterlegt werden, siehe Kapitel 5.1.2. In der config-Datei werden unter anderem die Zugangsdaten des zu analysierenden Repository angegeben, in der analysis-Datei die Zugangsdaten zur Neo4j-Graphdatenbank. Im ersten Schritt wird eine lokale Repository-Kopie auf dem Rechner abgespeichert. Ist diese bereits vorhanden werden ausschließlich die fehlenden Commits hinzugefügt und das Repository auf den aktuellsten Stand gebracht. Mithilfe einer Datenbankabfrage wird überprüft, welche Daten bereits in der Graphdatenbank vorhanden sind. Dieses Vorgehen soll verhindern, dass immer das gesamte

Repository analysiert werden muss und Daten doppelt abgespeichert werden. Abschließend folgt die Extraktion der Daten und die Speicherung in einer Graphdatenbank. Um nur die relevanten Daten, also die Architekturänderungen in der Datenbank abzuspeichern, werden die extrahierten Daten zuvor auf architekturrelevante Änderungen hin gefiltert. Dieses Vorgehen wird in Kapitel 5.2 noch einmal genauer beschrieben.

### 5.1.2 Konfiguration

Der Benutzer hat durch die Konfigurationsdatei ServiceConfig die Möglichkeit, auf eine einfache Art und Weise das zu analysierende Repository anzugeben. Dafür muss die URI des zu analysierenden Git-Repository angegeben werden und da nicht alle Repositories öffentlich zugänglich sind, muss des weiteren der Benutzername und das Passwort des Git-Benutzerkontos angegeben werden.
Zudem kann die Ordner-Struktur bei der Analyse nicht komplett automatisiert erfasst werden, da diese von Projekt zu Projekt unterschiedlich sein kann. Daher muss in der analysis-Datei angegeben werden, wo die Bundles liegen und welche Bundles bei der Analyse gegebenenfalls ausgeschlossen werden sollen, siehe Beispieldatei 5.1. Des weiteren müssen Angaben zu der zur Speicherung verwendeten Graphdatenbank gemacht werden.

Listing 5.1: Beispieldatei analysis.conf

```
[Project]
name=ProjektXY
bundleSourceFolder=/src/main/java
bundleRoot=com/
bundlePaths=de.rcenvironment.*
bundlePathsExclude=\de.rcenvironment.core.gui.workflow,
       \de.rcenvironment.core.component.integration, \


[Neo4j]
uri=bolt://neo4j.de:1234
user=neo4j
password=************
```

### 5.1.3 Persistierung

Um die Evolution einer Softwarearchitektur dauerhaft auf einer Neo4j-Graphdatenbank abzulegen, wird die Neo4j Object Graph Mapping Library (OGM) und die Spring Data Neo4j Library eingesetzt.

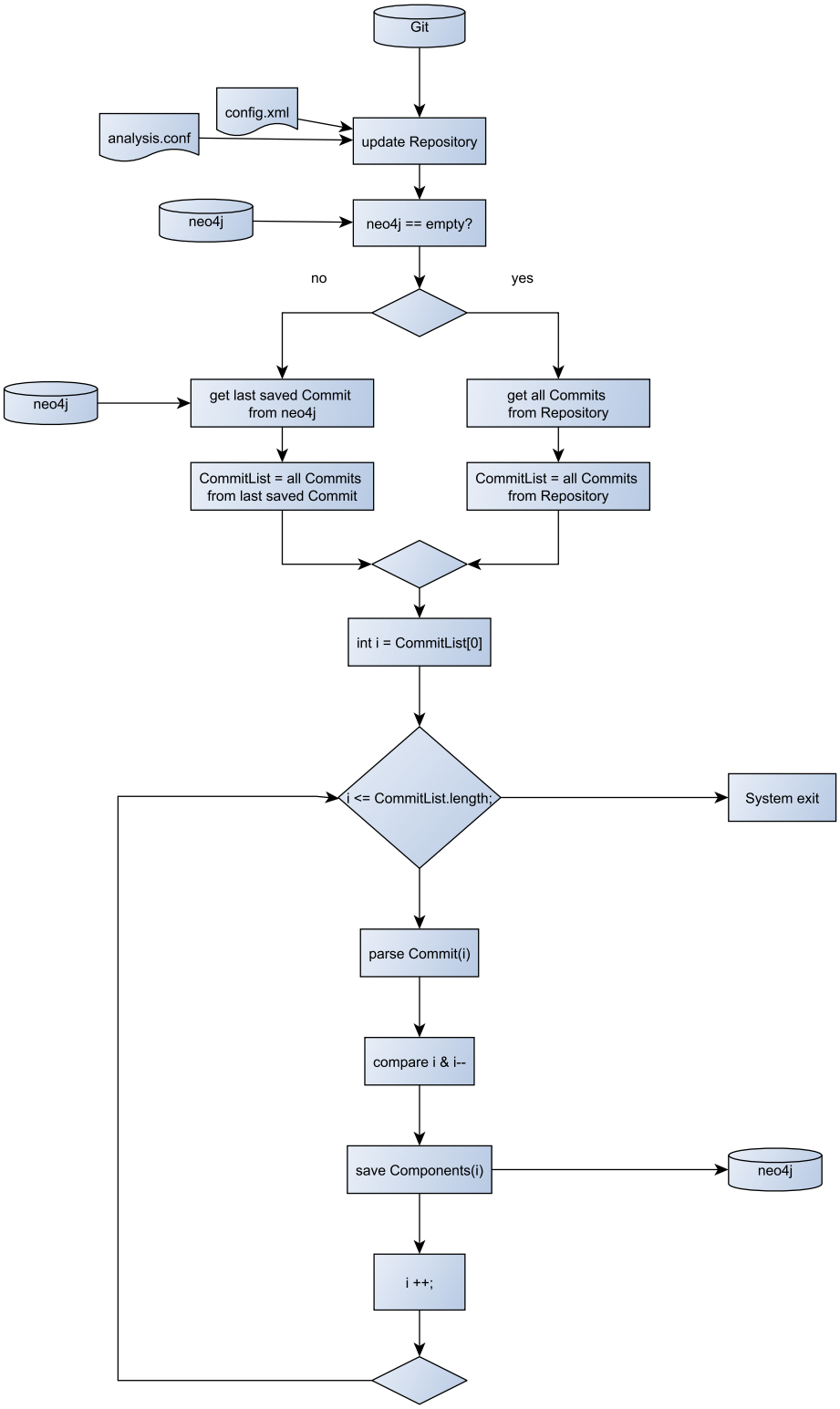



Abbildung 5.1: Workflow

### Neo4j Object Graph Mapping Library

Die Neo4j Object Graph Mapping Library (OGM) ist eine reine Java-Bibliothek von Neo4j entwickelt. Sie bietet drei Treiber als Möglichkeiten an, um eine Verbindung zur Datenbank herzustellen, Embedded, HTTP oder über das binäre Protokoll Bolt. In der Arbeit kommt das Bolt-Protokoll für die Verbindung zum Neo4j-Server zum Einsatz. Mithilfe der Session-Factory können Sessions geöffnet und wieder beendet werden. Während einer Session werden Interaktionen zwischen den definierten Entitäten und der Neo4j-Graphdatenbank gesteuert. Um eine Session zu öffnen, wird die URL der Neo4j-Datenbank benötigt. Daher muss diese zu Beginn in der analysis-Datei hinterlegt werden, wie in Kapitel 5.1.2 ausführlich erklärt wurde. Alle Änderungen, welche während einer Session an Entitäten oder Beziehungen vorgenommen werden, sind zu protokollieren.
Der Object Graph Mapper ordnet mit Anmerkungen versehene POJOs Knoten, Beziehungen und Eigenschaften zu. Die Zuordnung zwischen Objekt und Graph findet in dieser Arbeit in einer Klasse Namens OSGiApplicationModel statt.
Mit @NodeEntity annotierte POJOs werden als Knoten im Graph dargestellt. Wenn dem Knoten keine Beschriftung über die Eigenschaft *label* zugewiesen wurde, wird standardmäßig der Klassenname verwendet. Ein Feld in einer Entität, welches auf eine andere Entität verweist, definiert eine Beziehung. Mit Hilfe der Annotation *@Relationship* wird der Typ und die Richtung der Beziehung in der Entity-Klasse angegeben. In dem Projekt wurden die Richtungen ausschließlich als Outgoing definiert, um so das Modell von den großen Komponenten hin zu den kleineren zu gestalten.

### Spring Data Neo4j

Die Spring Data Neo4j Library basiert auf der Neo4j-OGM Library, siehe Abbildung 5.2. Die Kernfunktionaliät der Bibliothek kann jedoch standalone verwendet werden, ohne dass die Ioc-Dienste des Spring Containers aufgerufen werden müssen. Spring Data Neo4j bietet erweiterte Funktionen, um annotierte Entity-Klassen der Graphdatenbank zuzuordnen, welche bei der Umsetzung zur Speicherung der architekturrelevanten Änderungen einen Vorteil schaffen, siehe Kapitel 5.2. In dieser Arbeit wurden unter anderem folgende Entity-Klassen definiert :

- BundleImpl
- ClassImpl
- CommitImpl
- CompilationUnitImpl
- InterfaceImpl
- PackageImpl
- PackageFragmentImpl
- ServiceImpl
- ServiceComponentImpl
- VersionImpl

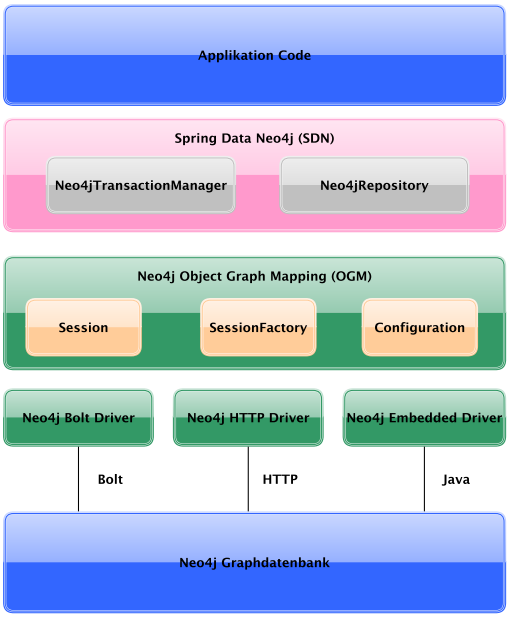



Abbildung 5.2: Aufbau der Neo4j-OGM Library und Spring Data Neo4j Library

## 5.2 Architekturrelevante Softwareänderungen

Lehman traf in den *Laws of Program Evolution* die Aussage, dass ein Softwaresystem mit der Zeit an Größe, Funktionalität und Komplexität zunimmt. Dies beeinflusst im Laufe der Zeit auch die Architektur des Systems. Doch nicht jede in einem Repository gespeicherte Änderung stellt eine Relevanz bei der Betrachtung der Architektur dar - nicht in jedem Commit befinden sich architekturrelevante Änderungen. Deshalb soll im Folgenden erläutert werden, wie architekturrelevante Änderungen definiert, identifiziert und in einer Graphdatenbank abgespeichert werden können.
Von jedem Commit wurde ein Architekturmodell des jeweiligen Zeitpunktes erstellt, siehe Kapitel 5.1.1. Teil dieser Arbeit ist es jedoch nicht die Architektur eines jeden Commits zu betrachten, sondern ausschließlich die zeitliche Veränderung der Architektur. Daher sollen auf Grundlage, des in der Analyse, erstellten Models die wesentlichen Änderungen identifiziert werden. Dazu gehören Änderungen auf Komponentenebene und Änderungen, die Abhängigkeiten und Beziehungen der Komponenten betreffen. Konkret sind das folgende Änderungen:

- Bundles, Packages oder Klassen/Interfaces wurden hinzugefügt
- Bundles, Packages oder Klassen/Interfaces wurden gelöscht
- Services wurden bereitgestellt

- Services wurden hinzugefügt
- Es wird auf ein Service referenziert

### 5.2.1 Identifizierung architekturrelevanter Änderungen

Im folgenden Kapitel soll erläutert werden, anhand welcher Vorgehensweise architekturrelevante Änderungen vor dem Speichervorgang identifiziert und die Menge der Daten auf die wesentlichen Informationen reduziert werden können.
Anhand des Workflows, siehe Abbildung 5.1, ist zu erkennen, dass zunächst von jedem analysierten Commit ein Architekturmodell erstellt wird. Dieses Modell besteht aus verschachtelten Listen, in denen die einzelnen Komponenten abgespeichert sind. Da der Speichervorgang pro Modell sehr aufwändig ist (siehe Kapitel 6.2), ist es sinnvoll das Modell vor dem Speichervorgang auf die wichtigsten Komponenten zu reduzieren und den Vergleich nicht erst danach, auf Grundlage der Datenbank, durchzuführen. Um Gleichheit und Veränderungen in der Architektur zu identifizieren, werden die Modelle zwei zeitlich benachbarter Commits miteinander verglichen. Der Einfachheit halber werden die Commits im weiteren Kapitel C1 und C2 genannt.
Der Vergleich der zwei Modelle beginnt auf höchster Komponentenebene - den Bundles - und wird absteigend, bis zu den Klassen und Interfaces, durchgeführt. Das bedeutet, dass im ersten Schritt die Liste der Bundles aus C1 und C2 miteinander verglichen werden. Dies führt zu einer Laufzeitkomplexität von $O(n^m)$.

Um das Vorgehen zur Identifikation architekturrelevanter Änderungen verständlich zu erläutern, wird die Vorgehensweise in folgende zwei Teile gegliedert und mithilfe von Entscheidungstabellen verdeutlicht:

- Keine architekturrelevanten Änderungen
- Identifizierung von Architekturänderung

#### Keine architekturrelevanten Änderungen

Wenn die Listen der einzelnen Komponenten von C1 und C2 gleich geblieben sind, kann auf dem ersten Blick davon ausgegangen werden, dass keine Architekturänderungen vorliegen, wie in der Entscheidungsmatrix 5.1 zu sehen. Es sind auf Bundle-,
Package-, Package Fragment-, Klassen-/Interface- und Serviceebene keine Komponenten hinzugefügt oder gelöscht worden, auch die Beziehungen zwischen den Komponenten haben sich nicht verändert.

Allerdings ist ein wichtiger Aspekt, der im Bezug auf die Identifizierung architekturrelevanter Änderungen beachtet werden muss, Refactorings. Durch die stetige Zunahme

| | |
|---|---|
| Bundles gleich | 1 |
| Packages gleich | 1 |
| Package Fragmente gleich | 1 |
| Klassen/Interfaces gleich | 1 |
| Services gleich | 1 |
| Architektur gleich | x |
| Architekturänderungen | |

Tabelle 5.1: Entscheidungsmatrix, wenn keine Architekturänderungen vorliegen

der Komplexität müssen Refactorings eingesetzt werden, um die interne Qualität zu verbessern, siehe Kapitel 2.3. Doch nur eine begrenzte Anzahl an Refactorings sind bei der Betrachtung der Evolution einer Softwarearchitektur von Interesse, wie zum Beispiel:

- Extract/Inline Class
- Extract Packages
- Replace Type Code with Subclasses
- Replace Data Value with Object

Renames dagegen können leicht als architekturrelevante Änderung identifiziert werden, wobei sie in diesem Kontext keine Relevanz darstellen. Das Löschen einer Komponente und das Hinzufügen einer scheinbar „neuen“ Komponente unter anderem Namen lassen leicht die Schlussfolgerung zu, dass es sich hierbei um eine Architekturänderung handelt. Doch da sich die Funktionalität der Komponente und deren Beziehungen und Abhängigkeiten nicht verändert haben, sollen Renames nicht als architekturrelevante Änderung aufgenommen werden. Sie bieten bei der Betrachtung der Evolution der Architektur keinen Mehrwert.
Um Renames fälschlicherweise nicht als Architekturänderung zu identifizieren, werden die darüber und darunter liegenden Komponentenebenen zusätzlich betrachtet. Wenn die Anzahl der jeweiligen Komponentenebene zwischen C1 und C2 gleich geblieben ist, sich lediglich nur eine Komponente verändert hat, doch die darunter und darüber liegenden Komponentenebenen ebenfalls unverändert geblieben sind, haben sich die Beziehungen und die Funktionalität der Komponente nicht verändert. Daher liegt keine Architekturänderung vor, siehe Entscheidungsmatrix 5.2.

| | | | | |
|---|---|---|---|---|
| Bundles gleich | 0 | 1 | 1 | 1 |
| Packages gleich | 1 | 0 | 1 | 1 |
| Package Fragmente gleich | 1 | 1 | 0 | 1 |
| Klassen/Interfaces gleich | 1 | 1 | 1 | 0 |
| Services gleich | 1 | 1 | 1 | 1 |
| Architektur gleich | x | x | x | x |
| Architekturänderungen | | | | |

Tabelle 5.2: Entscheidungstabelle, wenn keine Architekturänderungen vorliegen trotzdem sich eine Komponente scheinbar verändert hat

| | | | | |
|---|---|---|---|---|
| Bundles gleich | 0 | 1 | 1 | 1 |
| Packages gleich | 1 | 0 | 1 | 1 |
| Package Fragmente gleich | 1 | 1 | 0 | 1 |
| Klassen/Interfaces gleich | 1 | 1 | 1 | 0 |
| Services gleich | 1 | 1 | 1 | 1 |
| Architektur gleich | | | | |
| Architekturänderungen | x | x | x | x |

Tabelle 5.3: Entscheidungstabelle, wenn eine Komponenten gelöscht oder hinzugefügt wurden

#### Identifizierung von Architekturänderung

Eine Architekturänderung liegt auf jeden Fall vor, wenn eine Komponente gelöscht oder hinzugefügt wurde, siehe Entscheidungsmatrix 5.3. Sich somit die Anzahl der Komponenten verändert hat.

Es liegt ebenfalls eine Architekturänderung vor, wenn sich auf mehr wie einer Komponentenbene die Liste der Komponenten geändert hat, siehe Entscheidungsmatrix 5.4 und 5.5. In diesem Fall ändern sich die Beziehungen und die Funktionalitäten der jeweiligen Komponenten.

Auch Renames können in diesem Fall ausgeschlossen werden. Liegt zum Beispiel eine Änderung auf Package Fragmentebene vor, aber die Anzahl der Package Fragmente und die Bundles, Klassen und Interfaces sind gleich geblieben, kann davon ausgegangen werden, dass es sich hierbei um ein Rename handelt. Wenn allerdings auch eine Änderung auf Klassen-/Interfacebene oder auf Bundlebene identifiziert wurde, handelt es sich um keinen Rename, da sich die Beziehungen und die Funktionalität des Package Fragments verändert hat und somit eine Architekturänderung vorliegt.

| Bundles gleich | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|
| Packages gleich | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| Package Fragmente gleich | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| Klassen/Interfaces gleich | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| Services gleich | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| Architektur gleich | | | | | | | | |
| Architekturänderungen | x | x | x | x | x | x | x | x |

Tabelle 5.4: Entscheidungstabelle, wenn Architekturänderungen vorliegen

| Bundles gleich | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|
| Packages gleich | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| Package Fragmente gleich | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| Klassen/Interfaces gleich | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| Services gleich | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| Architektur gleich | | | | | | | | | | |
| Architekturänderungen | x | x | x | x | x | x | x | x | x | x |

Tabelle 5.5: Entscheidungstabelle, wenn Architekturänderungen vorliegen

### 5.2.2 Speicherung architekturrelevanter Änderungen

Nachdem die Architekturänderungen zwischen dem Commit C1 und C2 identifiziert wurden, sollen diese ohne Informationsverlust in der Graphdatenbank abgespeichert werden. Da der Inhalt der Graphdatenbank auf die wesentlichen Daten reduziert werden soll, um den Speichervorgang und die Visualisierung effizient zu gestalten, bleibt zunächst die Schlussfolgerung ausschließlich die identifizierten Architekturänderungen zu speichern. In der Abbildung 5.3 wurde dieses Vorgehen beispielhaft dargestellt. Unter dem Node C1, welcher den ersten analysierten Commit darstellt, wurde die gesamte Architektur gespeichert. Unter dem nachfolgenden Node C2 ausschließlich die identifizierten Komponenten, die sich von Commit C1 zu Commit C2 verändert haben. In der Abbildung ist zu erkennen, dass sich Bundle A verändert hat und ein Package Fragment D mit einer Compilation Unit und einem Interface hinzugefügt wurden.
Doch ist mit diesem Vorgehen keine Aussage möglich, was mit Bundle B und Bundle C von C1 zu C2 passiert ist beziehungsweise ob Package Fragment A in Bundle A von C1 zu C2 gelöscht wurde oder einfach unverändert geblieben ist. Das heißt, eine Unterscheidung zwischen gelöschtem Element und unverändertem Verhalten ist somit nicht möglich. Es kommt dadurch zu einem Informationsverlust.
Ein anderes Vorgehen wurde entwickelt, um diesen Informationsverlust zu umgehen und die Graphdatenbank trotzdem auf die wesentlichen Daten zu reduzieren, siehe Abbildung 5.4.
Wenn keine Architekturänderung von C1 zu C2 identifiziert wurde, wird eine Beziehung von dem Commit-Node C2 zu dem unveränderten Bundle hergestellt. In der Abbildung wird dieses Vorgehen durch den rot gestrichelten Pfeil von C2 zu dem Bundle B hervorgehoben. Somit ist leicht zu erkennen, dass Bundle B unverändert im Commit C2 enthalten ist, Bundle C jedoch von C1 zu C2 gelöscht wurde.
Wenn allerdings eine Architekturänderung von C1 zu C2 vorliegt, wird unter dem Node C2 das gesamte Bundle mit allen dazugehörigen Komponenten gespeichert. Denn würde nicht das gesamte Bundle mit den veränderten Komponenten gespeichert werden, würde dies zu einer falschen Schlussfolgerung führen, wie in Abbildung 5.5 dargestellt. Das Vorgehen in Abbildung 5.5 würde dazu führen, dass das Package Fragemt im Bundle sowohl in C1 als auch in C2 vorliegt und somit keine Änderung identifiziert wurde. Daher muss immer das gesamte Bundle mit allen Komponenten abgespeichert werden, wenn eine Architekturänderung vorliegt. Wenn keine Änderung vorliegt, wird eine Art Querverweis zum unveränderten Bundle hergestellt.
Mithilfe dieses Vorgehens, siehe Abbildung 5.4, lässt sich auf der einen Seite zu jedem beliebigen Zeitpunkt die komplette Architektur reproduzieren und auf der anderen Seite wird die Datenmenge auf die wesentlichen Informationen beschränkt.

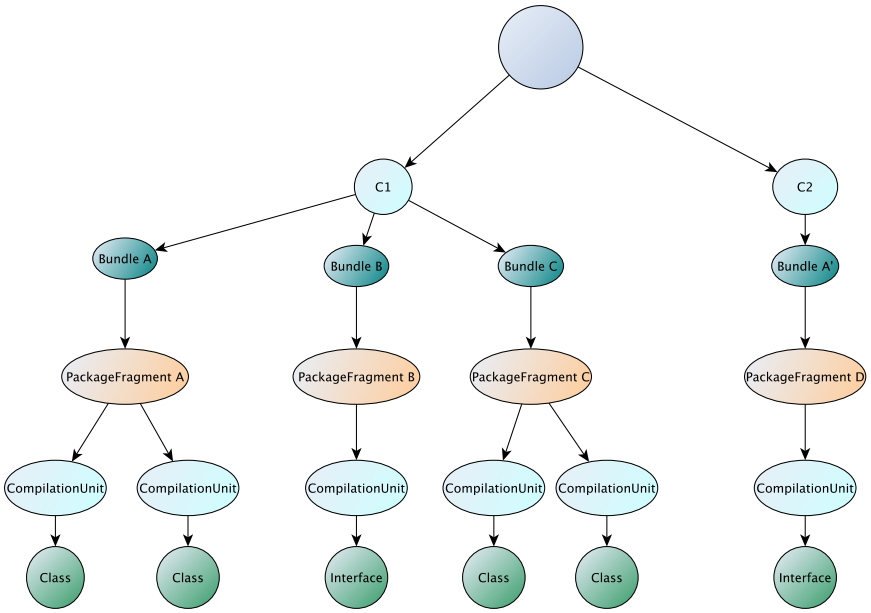



Abbildung 5.3: Beispielhafter Datenbankausschnitt mit Informationsverlust

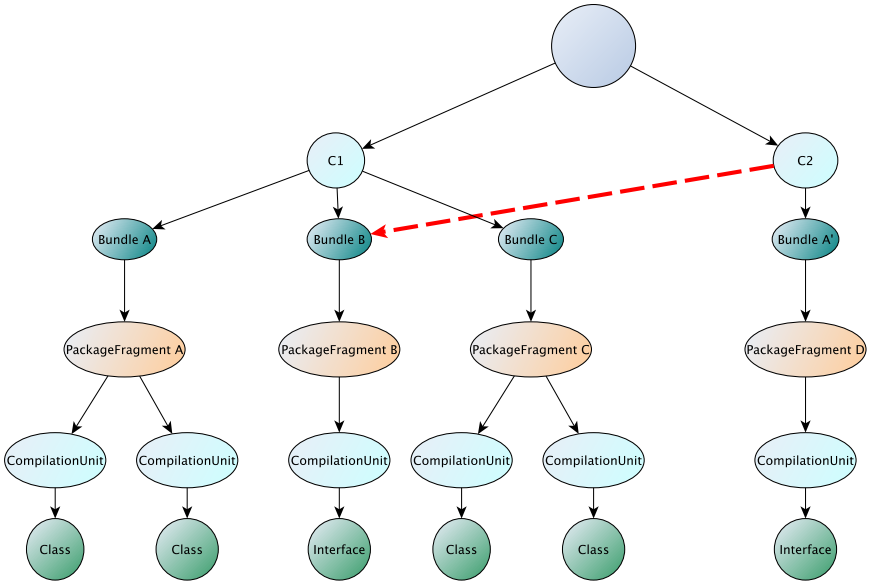



Abbildung 5.4: Beispielhafter Datenbankausschnitt ohne Informationsverlust

### Technische Umsetzung

Jeder Knoten und jede Beziehung müssen in einer Graphdatenbank eine eindeutige ID besitzen. Neo4j bietet die Möglichkeit mit Hilfe der Annotation @Index jedes belie-

bige Feld in einer Entity-Klasse als Indize zu deklarieren. Durch den Parameter unique=true kann diesem Feld zusätzlich eine Einschränkungen hinzugefügt werden. Die Einschränkung wird bei der Speicherung der architekturrelevanten Änderungen zunutze gemacht. Bei allen Entity-Klassen, siehe Kapitel 5.1.3, wurde die Property, welche den Namen enthält, mit dieser Einschränkung versehen. Wenn keine Architekturänderung in einem Bundle vorliegt werden auch keine neuen Knoten erzeugt. Es wird automatisch eine Beziehung zu dem bereits bestehenden Knoten erstellt. Stimmt beispielsweise das komplette Bundle zwischen C1 und C2 überein, werden keine neuen Knoten erzeugt, ausschließlich eine Beziehung zwischen dem Commit C2 und dem unveränderten Bundle wird hergestellt.
Wurde allerdings eine Architekturänderung identifiziert wird allen unveränderten Komponenten in diesem Bundle eine Versionsnummer, welche sich an dem Commit orientiert, zu dem Namen hinzugefügt. Somit wird die Einschränkung unique=true in diesem Fall umgangen und das komplette Bundle mit allen dazugehörigen Komponenten wird als architekturrelevante Änderung abgespeichert. So findet keine Vermischung zwischen veränderten und unveränderten Komponenten eines Bundles in den Commits C1 und C2 statt, siehe Kapitel 5.2.2.
Durch dieses Vorgehen werden nur neue Knoten erzeugt, wenn eine Architekturänderung vorliegt, wodurch die Graphdatenbank auf die wesentlichen Komponenten beschränkt wird.

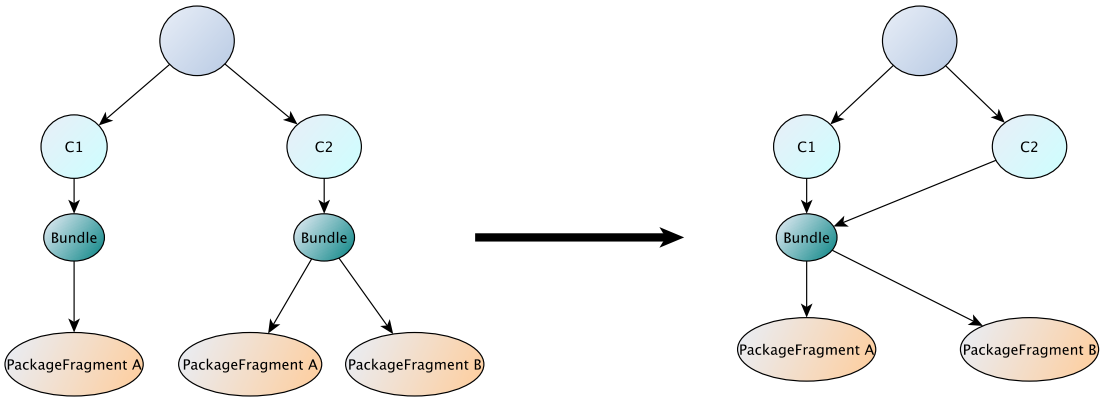



Abbildung 5.5: Beispielhafter Datenbankausschnitt architekturrelevanter Änderungen

### 5.2.3 Mögliche Erweiterungen

Die Analyse findet mithilfe eines iterativen Vorgehens statt, wenn sich bereits Daten in der Graphdatenbank befinden, beginnt der Analyseprozess nicht bei dem ersten Commit, sondern nach dem zuletzt in der Graphdatenbank gespeicherten Commit. Um architekturrelevante Änderungen zwischen dem zuletzt in der Graphdatenbank gespeicher-

ten Commit und dem analysierten Commit danach zu identifizieren, muss eine weitere Funktion implementiert werden. Die Architekturinformationen aus der Graphdatenbank müssen von dem jeweiligen Commit extrahiert werden, bevor ein Vergleich der zwei Models erfolgen kann.
Des Weiteren wurde bei dem zuvor beschriebenen Vorgehen nicht die Anwendung von Renames auf Klassen und Interfaces berücksichtigt. Da es in dieser Arbeit vordergründig um die Komponenten und deren Beziehungen geht, werden Renames auf Klassen und Interfaces nicht weiter beachtet. Die Funktionalität der einzelnen Klassen und Interfaces spielt bei der Betrachtung der gesamten Architektur eine untergeordnete Rolle. Daher wird in dieser Arbeit ausschließlich die Anzahl und die Beziehungen der Klassen und Interfaces berücksichtigt. Neben den Klassen und Interfaces deckt der bisher implementierte Algorithmus nicht alle, im Abschnitt 5.2.1 beschriebenen, Fälle ab.

## 5.3 Visualisierung

Wenn das menschliche Gehirn große Datenmengen erfassen muss, sind Visualisierungen extrem leistungsstark. Daher soll im folgenden Kapitel die Umsetzung des im Kapitel 4.3 beschriebenen Konzepts zur Visualisierung der gesamten Evolution einer Softwarearchitektur vorgestellt werden.

### 5.3.1 Plattform

Für die Umsetzung der Visualisierung bieten sich zwei grundlegende Ansätze an: die Entwicklung einer eigenständigen Anwendung und die einer webbasierten Anwendung im Browser. Beide Ansätze haben Vor- und Nachteile, welche in den folgenden Tabellen 5.6 und 5.7 aufgelistet werden.

| **Vorteile** | **Nachteile** |
|---|---|
| - Performance | - Implementierungsaufwand |
| - freie Wahl der Programmiersprache | - Verfügbarkeit |

Tabelle 5.6: Vor- und Nachteile einer eigenständig entwickelten Anwendung

| **Vorteile** | **Nachteile** |
|---|---|
| - Plattformunabhängigkeit | - Einschränkung auf Webtechnologien und Programmiersprachen |
| - Verfügbarkeit | - Performance |

Tabelle 5.7: Vor- und Nachteile einer webbbasierten Anwendung im Browser

Trotz der gleichen Anzahl an Vor- und Nachteilen fiel die Entscheidung für die eigenständig entwickelte webbasierte Anwendung im Browser aus. Gründe für diese Wahl waren vorrangig die Plattformunabhängigkeit und die hohe Verfügbarkeit. Dies führt zu einer geringen Einstiegshürde, denn der Benutzer hat dadurch die Möglichkeit jederzeit von überall aus auf die Visualisierung zuzugreifen. Des Weiteren kann die Analyse im Hintergrund auf dem Server ausgeführt werden, wodurch die Visualisierung immer den aktuellsten Stand präsentieren kann.

### 5.3.2 Herausforderung

Im folgenden Kapitel soll die Herausforderung bei der Umsetzung einer Visualisierung zur Evolution von Softwarearchitekturen erläutert werden.

#### Erste Ansätze

Die verbreitetsten Bibliotheken zur Visualisierung von Daten sind D3.js, OpenGL und Roassal [40]. Die bekannteste unter diesen ist D3.js.
D3.js steht für Data-Driven Documents. Dabei handelt es es sich um eine Javascript Bibliothek, mit der Daten in Form von HTML und SVG manipuliert werden können. D3.js beinhaltete drei Hauptfunktionen:

1. Laden der Daten in den Browser
2. Binden der Daten an Elemente des Dokuments
3. Transformation der Elemente
4. Übergang der Elemente in verschiedene Zustände aufgrund von Benutzereingaben

Dies ermöglicht die Umsetzung vieler Visualisierungsaspekte. Durch die Verwendung diverser Layout-Algorithmen, die D3.js zur Verfügung stellt, ist der Aufwand der Umsetzung überschaubar. Der Force-Layout sorgt zum Beispiel dafür, dass Knoten in der Darstellung nicht überlappen und Kanten sich nicht kreuzen. Damit lassen sich Graphen einfach darstellen. Auch die Größe der Knoten kann individuell angepasst werden, wodurch zum Beispiel die Art der Komponenten und Metriken veranschaulicht werden können. Durch die farbliche Hervorhebung von Kanten kann die Richtung und die Art der Beziehungen zwischen diesen visuell dargestellt werden, siehe Abbildung 5.6. Folgende grafische Variablen lassen sich mit D3.js leicht umsetzen, um die unterschiedlichen Komponenten, Beziehungen und Metriken dazustellen:

- Größe der Knoten
- Farbe der Knoten
- Dicke der Kanten
- Farbe der Kanten

Abbildung 5.6: Darstellung eines Graphen mit D3.js

Neben den Layout-Algorithmen bietet die Bibliothek auch eine große Anzahl an Interaktionsmöglichkeiten. Knoten können mit einem Mausklick selektiert werden, um nähere Informationen über ein bestimmtes Element zu erhalten. Zudem kann mithilfe eines Suchfeldes nach bestimmten Informationen gefiltert werden. Dies ermöglicht es nur bestimmte Komponenten oder ausschließlich einen Teil des Graphen darzustellen. Somit kann der Inhalt der Darstellung auf bestimmte Merkmale reduziert werden, um dem Betrachter eine bessere Aufnahme der Informationen zu gestatten.

Doch bei all diesen Möglichkeiten, bietet die mächtige Bibliothek D3.js keinen Visualisierungsansatz zur Darstellung der zeitlichen Veränderung der Architektur anhand eines Graphen in einer interaktiven Videosequenz.

### Integration historischer Informationen

Im Gegensatz zu vielen bisherigen Veröffentlichungen zum Thema Softwarevisualisierung, soll in dieser Arbeit die gesamte Evolution einer Softwarearchitektur visualisiert werden und nicht ein bestimmter Zeitpunkt. Die zeitliche Veränderung der Architektur

soll in einer Art Zeitraffer dargestellt werden.
Während der interaktiven Videosequenz wird durchgehend die Entwicklung der gesamten Architektur im Zeitraffer dargestellt. Die Architektur wird als Graph visualisiert, der automatisch beim Übergang von einem Zeitpunkt zum nächsten ausschließlich an den Elementen, welche Architekturänderungen vorweisen, angepasst wird, während der Rest des Graphs unverändert bleibt. Die durchgehend gleichbleibende Darstellung des Graphen verhilft dem Betrachter den Überblick zu behalten und den Fokus auf die Architekturänderungen zu legen. Doch genau darin besteht die größte Herausforderung, den Übergang von eine Zeitpunkt zum nächsten Zeitpunkt umzusetzen.
Um die Architektur zu einem Zeitpunkt darzustellen müssen die entsprechenden Daten im Browser gerendert werden. Der Graph richtet sich entsprechend den Daten aus, sodass Knoten nicht überlappt dargestellt werden und Beziehungen sich so wenig wie möglich kreuzen. Doch wenn die Architektur zum nächsten Zeitpunkt dargestellt werden soll, erhält der Browser neue Daten, die neu gerendert werden. Dies bedeutet, dass sich der Graph von einem Zeitpunkt zum nächsten jedes mal vollkommen anders anordnet. Wie zum Beispiel in der Abbildung 5.7 bildlich dargestellt, wurde ein neuer Knoten E hinzugefügt und der Graph zeigt eine deutliche Veränderung in seiner Anordnung. Für den Betrachter würde in diesem Fall der Wiedererkennungseffekt und die Orientierung innerhalb der Architekturdarstellung verloren gegangen. Insbesondere bei Graphen mit mehr als 200 Knoten erhält der Benutzer keinen Überblick mehr, wenn sich der Graph bei jeder Veränderung neu anordnet. Das hätte zur Folge, dass der Betrachter sich bei jeder Darstellung eines Zeitpunktes neu orientieren müsste und wesentliche Informationen zur Architekturänderungen nicht mehr auf den ersten Blick ersichtlich wären.
Es ist deshalb essentiell, dass die unveränderten Elemente des Graphen von einem Zeitpunkt zum nächsten ihre Position beibehalten. Wie in Abbildung 5.8 zu sehen ist, soll sich der Graph nicht vollkommen neu anordnen, wie in Abbildung 5.7, sondern lediglich an die neuen Daten anpassen. Daher wurde versucht ausschließlich die Differenz zwischen den beiden Zeitpunkten an der Browser zu übergeben und nicht die komplette Architektur. Doch auch dieser Ansatz konnte das Problem nicht beheben.
Ein weiterer Lösungsansatz ist jedem Knoten eine Position zu übergeben, die so lange unverändert bleibt, wie keine Architekturänderung vorliegt. Doch wenn neue Knoten zum Graphen hinzukommen, würde sich der Graph beim Zuweisen von Positionen nicht mehr automatisch ausrichten. Alle Positionen müssten manuell angepasst werden, um entsprechend Platz für neue Knoten zu schaffen. Der Graph verändert kontinuierlich seine Form und Größe im Laufe der Zeit, Knoten entstehen und verschwinden, neue Verbindungen werden hergestellt und wieder gebrochen. Das manuelle anpassen und ausrichten der Knoten und Verbindungen, für einen reibungslosen Übergang von einem Zeitpunkt zum nächsten, ist im Rahmen dieser Arbeit nicht umsetzbar. Daher wurde zur Umsetzung der Visualisierung das Toolkit KeyLines verwendet.

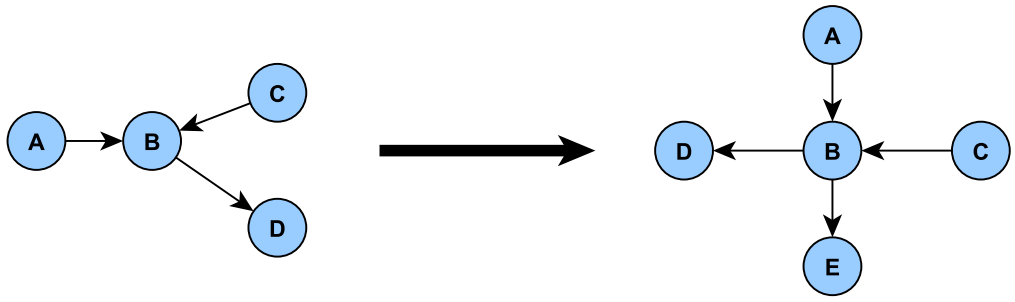



Abbildung 5.7: Ausgewählte Darstellungsart für Bundles

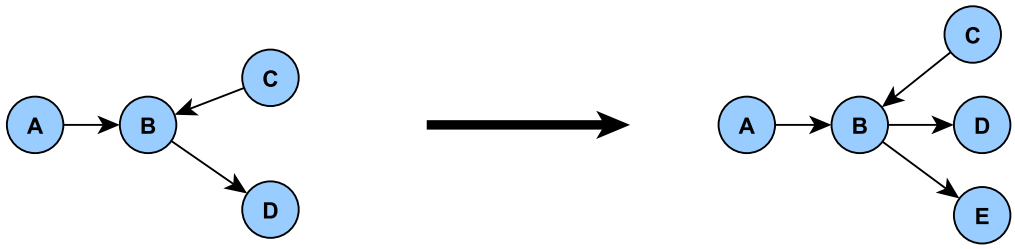



Abbildung 5.8: Ausgewählte Darstellungsart für Bundles

### 5.3.3 KeyLines

KeyLines ist das erste Visualisierungs-Toolkit für dynamische Graphen. Mit Hilfe der JavaScript-SDK können individuelle Visualisieurungsanwendungen für Graphen erstellt werden.

#### Schnittstelle zwischen Neo4j und Visualisierung

Die Visualisierung läuft vollständig im Webbrowser ab und die Architektur der Anwendung ist an den Model-View-Controller-Ansatz angelehnt. Das bedeutet, zu Beginn wird eine Verbindung zwischen der Graphdatenbank Neo4j und der Anwendung hergestellt, sowie eine Neo4j-Instanz eingerichtet. URL, Benutzername und Passwort der Graphdatenbank müssen angegeben werden, um später mit der REST-Schnittstelle interagieren zu können, siehe Abbildung 5.9.

Daten werden von Neo4j mit der Funktion „sendQuery“ abgefragt, welche eine AJAX-Cypher-Query abschickt, siehe Kapitel 2.7.1. Anschließend folgt mit der Callback-Funktion die Antwort. Da die Graphdatenbank ein Array zurückgibt, in dem jedes Element selbst ein Array ist, muss dieses in ein JSON-Format geparst werden.
In der Beispieldatei 5.2 ist die Funktion „makeNode“ abgebildet, mit Hilfe dieser Funktion werden alle Informationen eines Knotens in das JSON-Format geparst, das Äquivalent

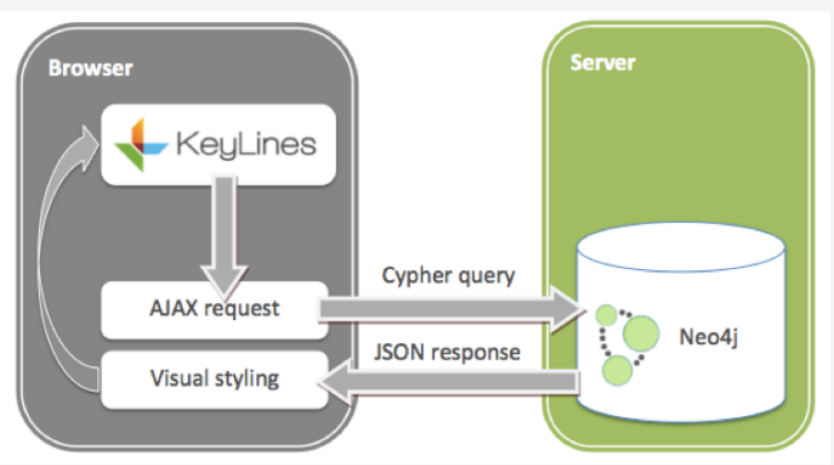



Abbildung 5.9: Darstellung der Architektur [8]

dazu ist die Funktion „makeLinks“. Nachdem die Daten peparst und in die Anwendung geladen wurden, wird der Graph in der Visualisierung dargestellt.

Listing 5.2: Funktion makeNode

```
var makeNode = function (neoItem) {
    var label = neoItem.label;
    var isBundle = label === "BundleImpl";

    return {
            type: 'node', // tells this object is a node
        c: isBundle ? "#1f77b4" : "#F54444", // Colour of the node
            e: isBundle ? 2 : 1, // Size of the node
            id: neoItem.id,
            dt: [neoItem.properties.timestamp], // Info for the timebar
            t: neoItem.id

    };
  };
```

### Detailebenen

Da die Darstellung der Architektur gerade bei großen und langlebigen Softwareprojekten zu einem Zeitpunkt mehrere tausend Knoten und Kanten enthalten kann, ist es von

Vorteil die Visualisierung in mehrere Detailebenen zu unterteilen. Die Darstellung der gesamten Architektur innerhalb eines Bildes, hätte für den Betrachter keinen Mehrwert. Die große Anzahl an Knoten und Kanten ist nicht übersichtlich auf einem Blick darstellbar. Der Betrachter kann somit keine Informationen mehr aus der Visualisierung ziehen, siehe Abbildung 5.10.

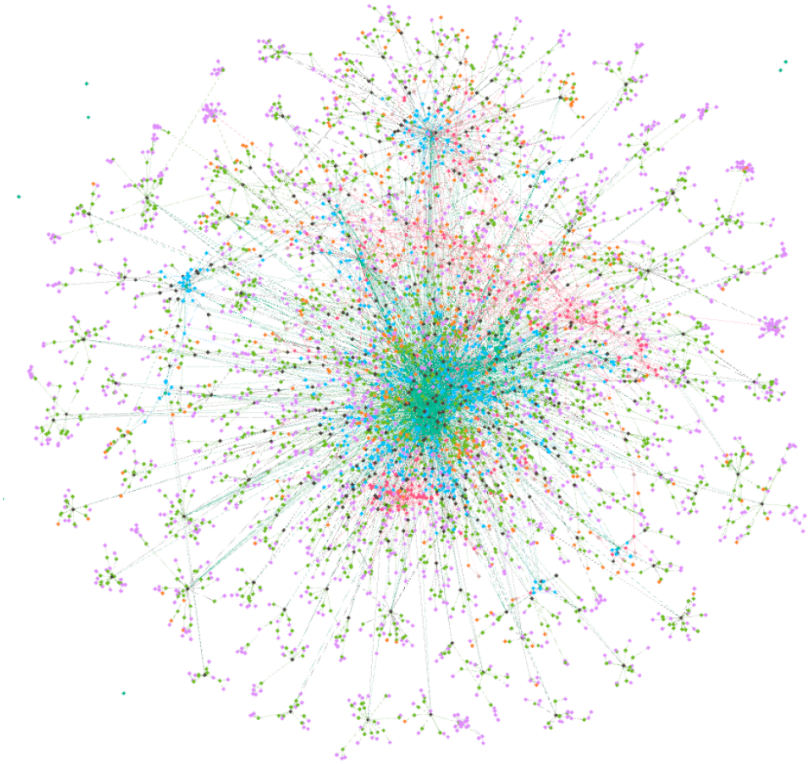


Abbildung 5.10: Darstellung der gesamten Architektur

Daher wurde die Visualisierung in folgende drei Sichten unterteilt:

1. Bundle-Graph
2. Service-Graph
3. Package- und Klassen-Graph

Die zentrale Ansicht der Visualisierung ist der Bundle-Graph, der einen ersten Überblick über alle Bundles darstellt, welche in einem Zeitraffer visualisiert werden, siehe Abbildung 5.11.
Somit erhält der Betrachter eine erste Übersicht über die Architektur auf höchster Komponentenebene und kann sich den eigenen Interessen entsprechend durch selektieren eines bestimmten Bundles dessen Aufbau anzeigen lassen. Besonders für Entwickler ist diese

Funktion von Interesse, da diese nicht immer eine Übersicht über die gesamte Architektur erhalten wollen. Entwickler sind in großen und komplexen Systemen für eine bestimmte Menge an Modulen verantwortlich die sie ausschließlich visualisieren wollen. Durch die beschriebene Funktion ist es ihnen möglich sich eine konzentrierte Ansicht der für sie relevanten Komponenten sowie deren Aufbau zu generieren.
Darüber hinaus hat der Betrachter die Möglichkeit zwischen den beiden weiteren Sichten zu wechseln, dem Service-Graph und dem Package- und Klassen-Graph, welche ebenfalls als Zeitraffer dargestellt werden, siehe in Abbildung 5.12 das Beispiel eines solchen Service-Graphen.

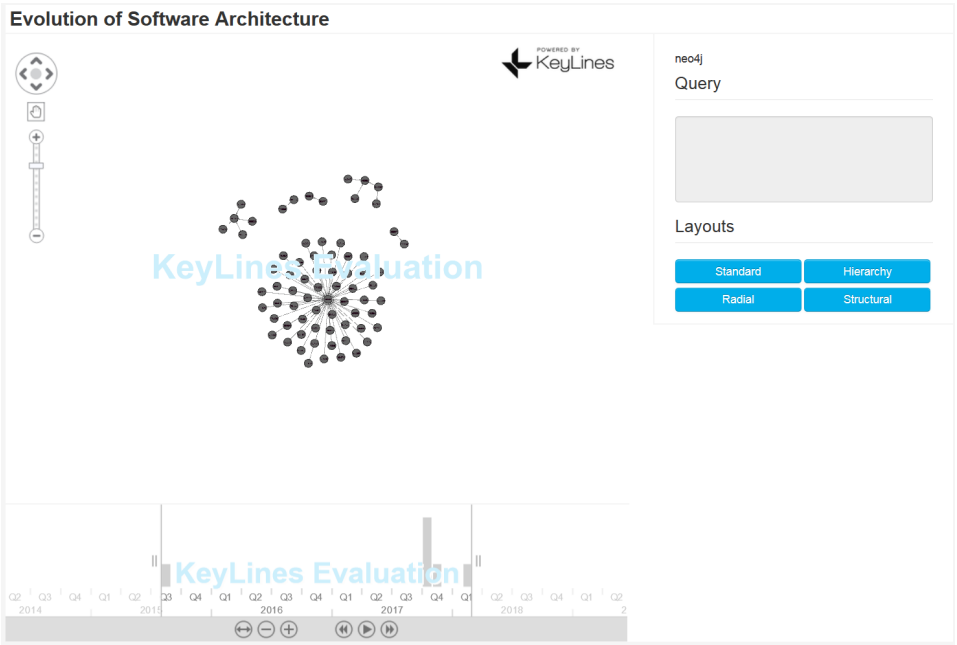



Abbildung 5.11: Darstellung des Bundle-Graphen

Im Vordergrund der Visualisierung steht, dass der Betrachter sich vollkommen eigenständig die für ihn relevanten Sichten zusammenstellen kann. Mithilfe der interaktiven Visualisierung können auch eigene Detailebenen gewählt werden. Auf der einen Seite durch Mausklick auf eine Komponente, wie oben beschrieben, auf der anderen Seite durch die Query-Funktion, siehe Abbildung 5.13. Dadurch hat der Betrachter die Möglichkeit durch eigene Queries die Darstellung der Architektur vollkommen individuell auf seine Bedürfnisse anzupassen.

### Interaktionsmöglichkeiten

Da in dieser Arbeit der Multi-View-Ansatz verfolgt wird, ist es wichtig eine intuitive Navigation zwischen den verschiedenen Sichten zu ermöglichen.
Die Knoten ordnen sich durch das Auto-Layouting automatisch an, können jedoch auch

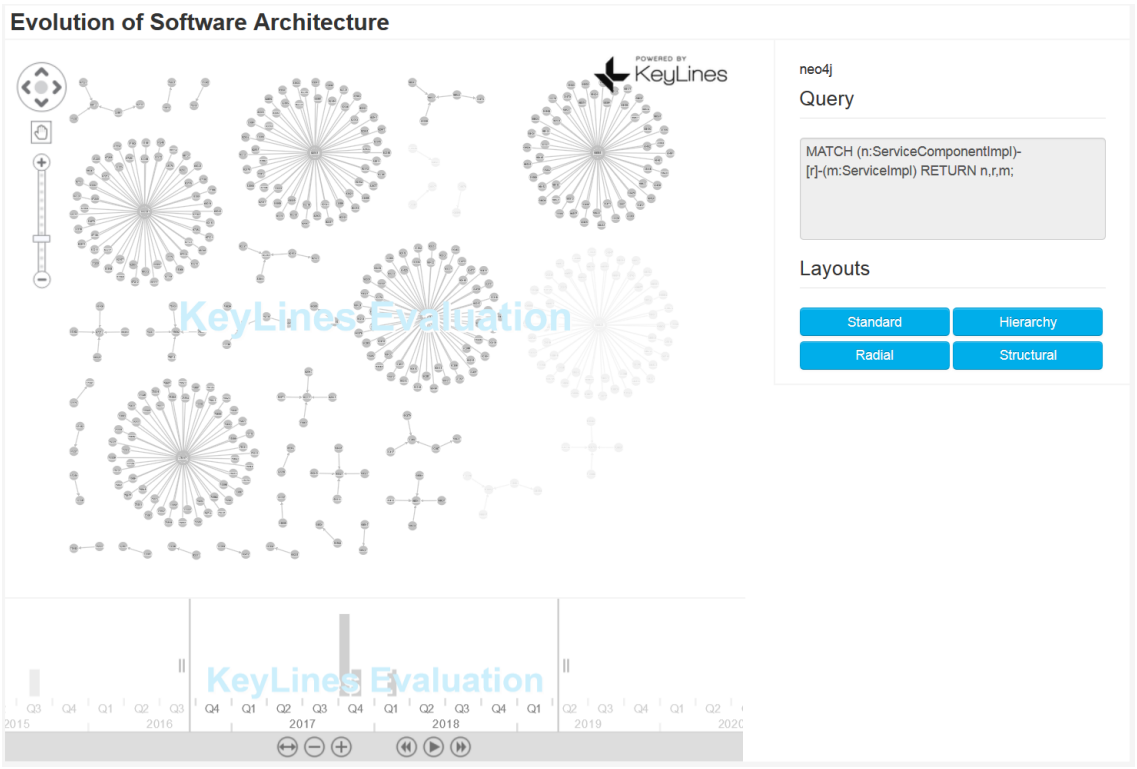



Abbildung 5.12: Darstellung des Service-Graphen

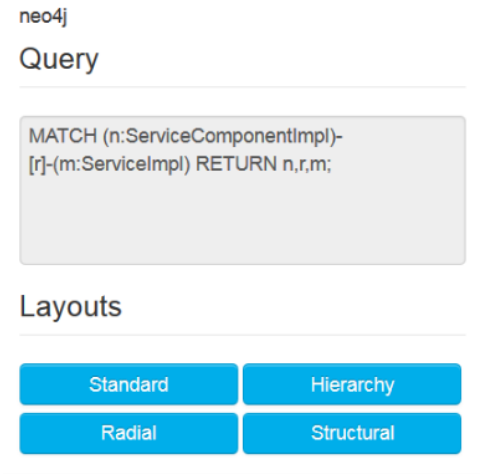



Abbildung 5.13: Darstellung der Query-Funktion

mit der Maus vom Benutzer verschoben werden. Des Weiteren können durch die Zoomleiste, siehe Abbildung 5.14, Ausschnitte des Graphen vergrößert, verkleinert oder verschoben werden. Somit kann der Betrachter in einem Standbild das Layout dahingehend

manipulieren, um bestimmte Aspekte besser betrachten zu können.

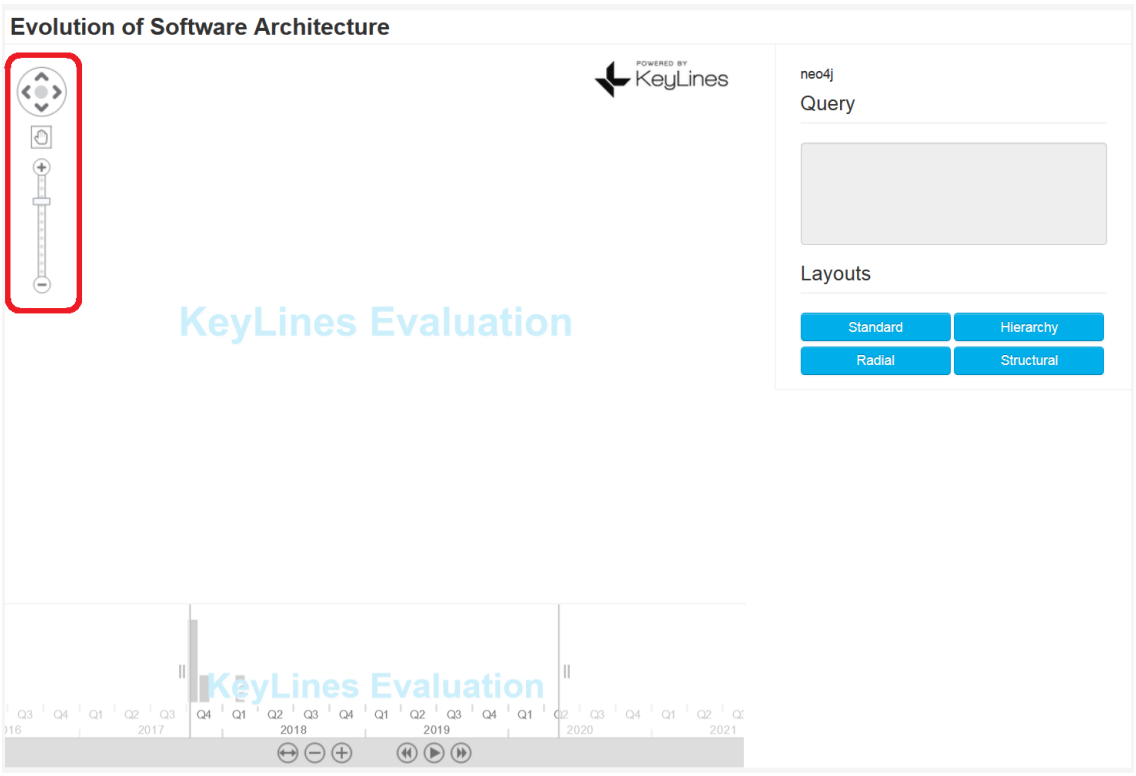



Abbildung 5.14: Darstellung der Zoomleiste in der Visualisierung

Um den Verlauf des Graphen in einen zeitlichen Kontext zu setzen und in einem Zeitraffer abspielen zu lassen, bietet KeyLines die Komponente Timebar, siehe Abbildung 5.15. Die Zeitleiste am unteren Rand ist vollständig in die Visualisierung integriert. Neo4j hat standardmäßig kein Datumsformat als Eigenschaft eines Knotens. Daher wurde jedem Knoten zusätzlich die Eigenschaft „timestamp“ hinzugefügt, in der das Datum des jeweiligen Commits als UNIX-Zeitstempel gespeichert wird. Die Zuordnung des Zeitstempels zum Knoten oder zu der Beziehung ist für die Umsetzung der Zeitleiste nicht von Interesse. In dieser Arbeit wurde die Property timestamp jeder Node-Entity-Klasse, siehe Kapitel 5.1.3, hinzugefügt. Zusätzlich bietet die Eigenschaft timestamp die Möglichkeit Informationen nach dem Datum zu filtern.
Wie anhand der Abbildung 5.15 zu sehen ist, bietet die Zeitleiste noch zusätzliche Steuerelemente. Somit ist nicht nur das Abspielen des Zeitraffers möglich. Somit ist nicht nur ein Abspielen des Zeitraffers möglich, auch anhalten, zurück- und vorspulen.
Zusammengefasst lässt sich mithilfe mehrerer Interaktionsmöglichkeiten innerhalb KeyLines der zeitliche Verlauf der gesamten Architektur so manipulieren, dass dem Betrachter Informationen zugänglich gemacht werden, welche ansonsten verborgen bleiben

würden.


Abbildung 5.15: Darstellung der Zeitleiste

### 5.3.4 Mögliche Erweiterungen

Wie zuvor schon erwähnt, beschränkt sich die Umsetzung der Visualisierung hauptsächlich auf die Implementierung des Zeitraffers. Daher sind Erweiterungen wie sie in dem Konzept, siehe Kapitel 4.3.1, beschrieben wurden möglich.
Des Weiteren ist es sinnvoll dem Betrachter die Möglichkeit zu geben aus einer Liste von vorgegebene Metriken auszuwählen und sich diese in der Visualisierung hervorheben zu lassen oder vollkommen losgelöst vom Rest der Visualisierung in einem extra Fenster anzeigen zu lassen.
Zusätzlich könnte die Visualisierung dahingehend erweitert werden, dass durch das Selektieren einer Komponente deren Spezifikation oder Source Code in einem separaten Fenster angezeigt wird. Inwiefern dieses Feature für Entwickler von Interesse ist, müsst jedoch erst noch geprüft werden.

# 6 Evaluation - Proof of Concept

In diesem Kapitel soll die Evaluation des Analyseteils der Arbeit und der Visualisierung vorgestellt werden. Als Anwendungsbeispiel soll das Remote Component Environment dienen, eine Software des Deutschen Zentrum für Luft- und Raumfahrt (DLR).
Eine Gegenüberstellung des Analyseverfahrens ohne Filterung architekturrelevanter Änderungen und mit der Filterung von Architekturänderungen soll aufzeigen, welchen Vorteil eine Selektierung der Daten bietet.
Im zweiten Teil sollen ausgewählte User Stories mit der Visualisierung umgesetzt werden, was als Grundlage für die Evaluierung dienen soll.

## 6.1 Remote Component Environment

*Remote Component Environment* (kurz RCE) ist eine Open-Source-Software, mit der komplexe Berechnungs- und Simulationsworkflows erstellt werden können [41].
Ein Workflow besteht dabei aus mehreren miteinander verbundenen Komponenten, siehe Abbildung 6.1. Dabei kann es sich um ein Simulationstool, ein Werkzeug für den Datenzugriff oder ein benutzerdefiniertes Skript handeln. Die Verbindung der Komponenten definiert den Datenfluss zwischen ihnen. Die Software stellt vordefinierte Komponenten, welche häufig benötigte Funktionen anbieten, zur Verfügung. Der Benutzer kann jedoch auch eigene Komponenten entwickeln und somit eigene Tools in ein Workflow integrieren. Bei RCE handelt es sich zudem um eine verteilte Anwendung, weshalb es möglich ist, die Komponenten lokal oder auf Remote-Instanzen von RCE auszuführen [13].
RCE ist ein mächtiges Werkzeug, welches in vielen Bereichen, wie zum Beispiel bei der Entwicklung von Flugzeugen oder bei der Optimierung des Wärmemanagements von Raumgleitern, Verwendung findet. Das Projekt wird seit 10 Jahren kontinuierlich (weiter-)entwickelt und gewartet, Während der Entstehung dieser Arbeit sind fünf Vollzeitmitarbeiter für das Projekt zuständig. Die Software besteht aus über 2000 Quellcode-Dateien und Modulen. Das Repository der aktuellsten Version 8.2.0 umfasst 5588 Dateien in 3292 Ordnern.
RCE ist eine in Java implementierte Software, welche auf dem OSGi-Framework basiert. Zudem nutzt RCE die Eclipse Rich Client Platform, welche zur Entwicklung von Desktop Anwendungen angewendet wird [16].
Bei RCE handelt es sich um ein äußerst umfangreiches und langlebiges Softwaresystem, weshalb die Analyse und Visualisierung der Evolution der Architektur des Systems von außerordentlichem Interesse ist.

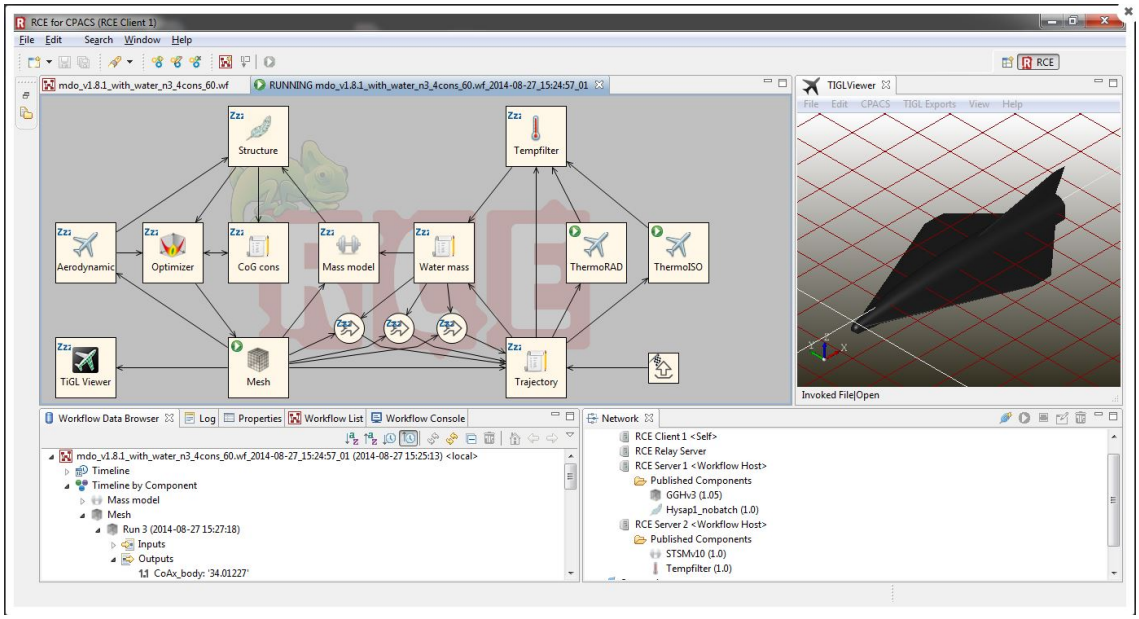



Abbildung 6.1: Screenshot einer RCE-Instanz (Workflow zur Optimierung des Thermal Management Systems eines SpaceLiners) [12]

## 6.2 Modellerstellung

Vor der eigentlichen Visualisierung muss die gesamte Architekturhistorie von RCE extrahiert und die wesentlichen Informationen in der Graphdatenbank gespeichert werden. Dieser Vorgang wurde auf einem Computer mit einem Intel i7-Prozessor aus dem Jahr 2009 mit 2,80GHz Grundtaktfrequenz ausgeführt.
Dabei war zu beobachten, dass das Speichern der gesamten Architektur für jeden Commit aus dem Repository sehr zeitaufwendig ist. Ungefähr 90% der Zeit wird für die Speicherung der Architektur in der Graphdatenbank benötigt, wobei die Analyse nur circa 2% beanspruchte, siehe Tabelle 6.1. Die restliche Zeit wird für das Herunterladen des Repository benötigt.

Um diesen Vorgang effektiver und zeitsparender zu gestalten, wurde in dieser Arbeit nicht die gesamte Architektur sondern ausschließlich die Änderungen der Softwarearchitektur gespeichert, siehe Kapitel 5.2. Wie anhand der Tabelle 6.2 zu sehen ist, führt dieses Vorgehen tatsächlich zu einer Zeitersparnis, wenn ausschließlich die Differenzen von einem Commit zum nächsten gespeichert werden. Nur beim ersten Commit ist keine Zeiteinsparung zu erkennen, da an dieser Stelle intitial die gesamte Architektur abgespeichert wird und noch keine Architekturänderungen, wie sie in Kapitel 5.2 definiert wurden, vorliegen.

Es lässt sich festhalten, dass die Beschränkung der Daten auf die wesentlichen Infor-

| Anzahl der Commits | Anzahl der Bundles | Gesamt in min | Analyse in % | Speicherung in % |
|---|---|---|---|---|
| 1 | 213 | 38:25 | 2,63 | 86,84 |
| 2 | 426 | 75:38 | 1,33 | 94,67 |
| 3 | 639 | 103:21 | 1,94 | 92,23 |
| 5 | 1.065 | 172:43 | 1,74 | 95,35 |

Tabelle 6.1: Zeiten für die Speicherung der gesamten Architektur für jeden Commit

| Anzahl der Commits | Anzahl der Bundles | Gesamt in min | Analyse in % | Speicherung in % |
|---|---|---|---|---|
| 1 | 213 | 40:15 | 2,51 | 87,57 |
| 2 | 426 | 48:02 | 2,33 | 86,79 |
| 3 | 639 | 61:23 | 2,72 | 85,64 |

Tabelle 6.2: Zeiten für die Speicherung der Architekturänderungen

mationen von Nutzen ist. Wie die Zahlen bestätigen, bringt das Identifizieren von Architekturänderungen und ausschließlich deren Speicherung eine erhebliche Zeitersparnis.

## 6.3 Evaluation der Visualisierung

Im folgenden Kapitel wird die Visualisierung am Beispiel von RCE und für jede der folgenden User Stories evaluiert:

1. Der Betrachter erhält eine Übersicht über die zeitliche Veränderung der Architektur.

2. Neue Entwickler erhalten einen Überblick über die gesamte Architektur.

### Übersicht über die zeitliche Veränderung

In diesem Anwendungsfall wird davon ausgegangen, dass sich die Architektur eines Softwareprojektes im Laufe der Zeit an verändernde Anforderungen anpassen muss. Des Weiteren wird davon ausgegangen, dass Entwickler während der Projektphase bewusst oder unbewusst falsche oder suboptimale Entscheidungen treffen, die nicht mehr der, zu Beginn des Projektes festgelegten, Strukturvorgaben entsprechen. Weshalb im Laufe eines Entwicklungsprojektes die Ist-Architektur von der Soll-Architektur abweicht und somit keine Übereinstimmung mehr zwischen diesen beiden vorhanden ist. Oftmals

liegt es an den technischen Schulden die über die Zeit hinweg entstanden sind. Daher soll die Visualisierung dazu beitragen, dass Entwickler oder Softwarearchitekten eine Übersicht über die zeitliche Veränderung einer Architektur erhalten, um somit die bisher angehäuften technischen Schulden zu identifizieren und der weiteren Erosion der Architektur gezielt entgegenwirken zu können, siehe Abbildung 6.2.

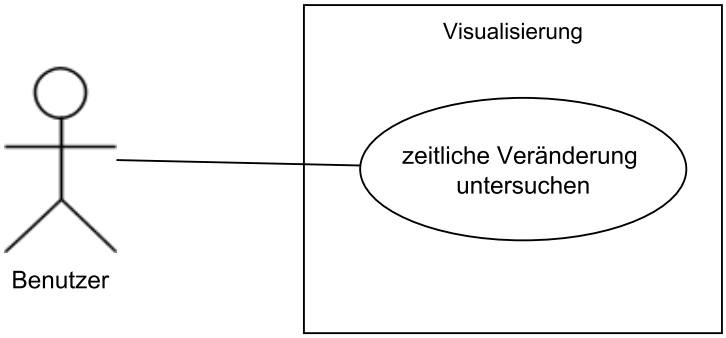



Abbildung 6.2: Use Case - Übersicht über die zeitliche Veränderung erhalten

Die Visualisierung der zeitlichen Veränderung der Architektur als Zeitraffer ermöglicht dem Betrachter eine große Menge an Daten visuell aufzunehmen und dadurch mit einem geringen Aufwand einen Überblick vom Entwicklungsverlauf einer Softwarearchitektur zu erhalten. Um konkret einzelne Änderungen in der Darstellung zu identifizieren, werden hinzugefügte Komponenten automatisch in den Graphen eingegliedert und die im bisherigen Entwicklungsverlauf gelöschten Komponenten transparenter abgebildet. Diese Darstellung bietet den Vorteil, dass auf einem Blick für den Benutzer erkennbar ist welche Komponenten von einem Zeitpunkt zum anderen Zeitpunkt hinzugekommen oder nicht mehr vorhanden sind. Wenn der Knoten dagegen vollkommen aus der Visualisierung verschwinden würde, wäre dem Betrachter bei der großen Anzahl an Knoten nicht bewusst, dass die entsprechenden Komponenten aus der Architektur gelöscht wurden.
Um die Interaktion zwischen Nutzer und Visualisierung nicht auf eine Betrachtung zu beschränken, wurden zusätzliche Interaktionsmöglichkeiten integriert, welche es gestatten aktiv in die abgebildete graphische Oberfläche einzugreifen. Dabei wurde eine spezielle Timebar zu Kontrolle der zeitlichen Dimension direkt in die Visualisierung integriert, um so durch eine intuitive Bedienung auf ausgewählte Stellen der gesamten Historie zugreifen zu können.

### Überblick über die gesamte Architektur

Wenn ein neuer Entwickler nachträglich in ein laufendes Softwareprojekt einsteigt, benötigt dieser einen ersten Überblick über die Software. Da das menschliche Gehirn nicht tausende Zeilen von Quellcode überblicken kann, soll die Visualisierung einen ersten Überblick

verschaffen, siehe Abbildung 6.3.

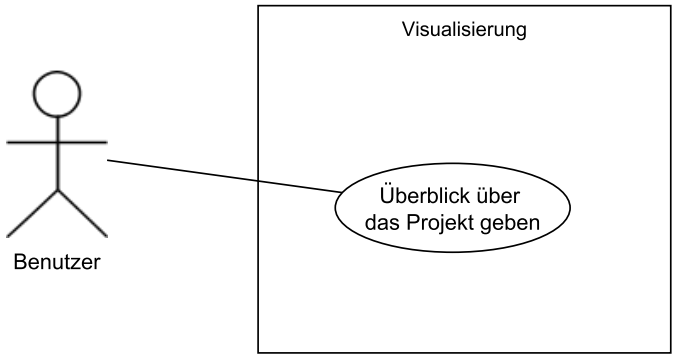



Abbildung 6.3: Use Case - Überblick über die gesamte Architektur bekommen

Im ersten Schritt ist es sinnvoll ein Gefühl für die Größe des Projektes zu vermitteln. Der Bundle-Graph welcher unmittelbar nach dem Start der Visualisierung dargestellt wird, vermittelt schnell einen ersten Eindruck von der Anzahl der Komponenten. Um einen ersten Einstieg in die Software zu erhalten, können die zentralen Module als Orientierung dienen. Durch das Einblenden der gerichteten Beziehungen lässt sich feststellen, wie viele eingehende Abhängigkeiten ein Bundle besitzt. Die zentralen Bundle, welche häufig benutzt werden, sind in dem Anwendungsbeispiel von RCE Core Communication, Core Component und Core Workflow. Diese Feststellung könnte durch das darstellen der Größe der einzelnen Komponenten noch weiter untermauert werden. Daher ist es sinnvoll die Größe der Knoten der Größe der Komponenten in der Visualisierung anzupassen.
Darüber hinaus werden durch das Layout die Bundles mit den meisten Abhängigkeiten eher in der Mitte der Visualisierung platziert. Da es jedoch Ausnahmen gibt, sollte dieser Annahme nicht zu viel Gewicht beigemessen werden.
Da neben den Bundles die Services ein weiteres zentrales Konzept von OSGi darstellen, ist eine Betrachtung dieser ebenfalls für den Einstieg relevant. Wie im Konzept, siehe Kapitel 4.3.1, bereits vorgestellt, sollen die Services über die Farbe den entsprechenden Bundles zugeordnet werden. Allerdings verursacht die Große Anzahl an Bundles zu viele verschiedene Farben, um auf den ersten Blick eine Unterscheidung treffen zu können. Daher ist eine Orientierung innerhalb des Service-Graph schwer. Der Service-Graph kann dem Betrachter insofern als Unterstützung dienen, indem zuvor nur eine begrenzte Anzahl an Bundles selektiert wird.
Doch neben den einzelnen Sichten ist auch als Überblick für neue Entwickler die zeitliche Komponente von Interesse. Denn es ist von großer Wichtigkeit die gesamte Evolution zu betrachten und nicht nur einen aktuellen Schnappschuss, um in der Lage zu sein, komplexe Systeme zu verstehen.

Als Ergebnis der Evaluation lässt sich festhalten, dass die Visualisierung der zeitlichen Veränderung einer Architektur als Zeitraffer von Nutzen sein kann. Die hier dargestellte Visualisierung soll ausschließlich als Prototyp dienen und ist in einigen Aspekten noch ausbaufähig. Doch die Darstellung der gesamten Evolution hat eine viel größere Aussagekraft über ein Softwareprojekt wie die Visualisierung einer einzelnen Momentaufnahme.

# 7 Fazit - To be continued...

Im folgenden Kapitel werden die Ergebnisse dieser Arbeit zusammengefasst und anschließend ein Ausblick auf mögliche Verbesserungen und auf diese Arbeit aufbauende Erweiterungen gegeben.

## 7.1 Zusammenfassung

Die Visualisierung der zeitlichen Veränderung von Architekturen kann dazu beitragen ein Verständnis des Softwareprojektes zu vermitteln und bietet durch die Darstellung des bisherigen Entwicklungsprozesses die Möglichkeit, Schlussfolgerungen aus diesem ziehen zu können. Jedoch gibt es bisher für die Darstellung der Evolution einer Softwarearchitektur noch keine praxistauglichen Visualisierungsanwendungen. Daher wurde in dieser Arbeit ein Visualisierungsansatz für die zeitliche Veränderung einer OSGi-Software konzipiert und beispielhaft implementiert.
Eine Java-Anwendung lieferte die dafür benötigten Daten, indem die Architekturinformationen über den gesamten Entwicklungszeitraum aus einem Git-Repository extrahiert und in ein Modell transformiert wurden. Um sowohl den Speichervorgang des Models in eine Graphdatenbank als auch die Visualisierung so effizient wie möglich zu gestalten, wurde das Java-Programm um weitere Funktionen erweitert. Das Modell wurde auf die wesentlichen Komponenten beschränkt, somit der Speichervorgang wesentlich Zeitsparender gestaltet und die Datenmenge der Graphdatenbank verringert.
Aufbauend auf dem Stand der Technik im Bereich der Softwarevisualisierung wurden Sichten konzipiert. Als visuelle Metapher wurden Graphen eingesetzt, um das Modell grafisch abzubilden. Zur Darstellung des zeitlichen Verlaufs der Architektur werden die einzelnen Models in einem Zeitraffer abgespielt. Interaktionsmethoden, wie die Selektion oder Navigation, bieten dem Benutzer die Möglichkeit spezifischere Informationen über einzelne Komponenten und Beziehungen der Software zu erhalten.
Sowohl die Beschränkung der Modelle auf die wesentlichen Komponenten als auch die Visualisierung wurden an dem Beispiel der OSGi-Anwendung RCE vom Deutschen Zentrum für Luft- und Raumfahrt evaluiert. Trotz der Umsetzung bietet diese Arbeit weiterhin Potential für Verbesserungen und Erweiterungen.

## 7.2 Ausblick

An dieser Stelle soll ein Ausblick gegeben werden, welche als Basis für eine Weiterentwicklung des Systems dienen kann.
Der Schwerpunkt dieser Arbeit liegt auf der Visualisierung der zeitlichen Veränderung von Softwarearchitekturen. Diese sollte unter anderem eine unterstützende Funktion einnehmen um Schwachstellen und Metriken ausfindig zu machen. Bisher können Metriken durch eigene Query-Abfragen visualisiert werden. Darüber hinaus wäre es besonders für unerfahrene Benutzer von Vorteil, wenn diese die Möglichkeiten bekommen aus einer Liste mit vorgegebenen Metriken zu wählen und die ausgewählte Metrik in der Visualisierung hervorzuheben oder in einem separaten Fenster anzeigen zu lassen.
Doch nicht nur Metriken wie Kohäsion, Kopplung oder die Größe können für den Betrachter von Interesse sein. Auch der Name des Entwicklers, der die entsprechende Änderung committet hat, kann in die Visualisierung integriert werden. Da die Information in der Datenbank bereits vorhanden ist, muss im nächsten Schritt eine geschickte visuelle Metapher gefunden werden, mit der die Information zusätzlich in die Visualisierung integriert werden kann. Damit könnte eine Aussage über die Aufgabenverteilung getroffen werden, ob die Verantwortlichkeiten im Entwicklungsteam gut verteilt sind oder vorrangig bei einer kleinen Menge an Teammitgliedern liegen.
Durch die Visualisierung der Historie als Zeitraffer kann beobachtet werden, wie die Architektur kontinuierlich wächst und an Komplexität zunimmt. Doch darüber hinaus kann diese Information erweitert werden, indem die einzelnen Releases in der Datenbank gespeichert und in der Timebar farblich hervorgehoben werden. Das kann neben dem Wachstum Aufschluss über die Produktivität oder Stagnation des Projektes geben.
Wie anhand der möglichen Erweiterungen schon angedeutet, ist das Erkennen einer Softwarearchitekturänderung nur ein Indikator. Für die weitere Ursachenforschung müssen weitere Quellen und Informationen herangezogen und in der Datenbank und Visualisierung integriert werden.

# Literaturverzeichnis


[1] The OSGi Alliance. OSGi service platform core specification, release 6, June 2014.

[2] Andrew Caudwell. Gource: a software version control visualization tool, 20.02.2018. `http://gource.io/`.

[3] Ken Arnold and James Gosling. *The Java programming language*. The Java series. Addison-Wesley, Boston, 3rd ed. edition, 2000.

[4] Fabian Beck and Stephan Diehl. On the congruence of modularity and code coupling. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 354–364, New York, NY, USA, 2011. ACM.

[5] Jacques Bertin. *Graphische Semiologie (Sémiologie graphique, dt.). Diagramme, Netze, Karten*. 2. franz. aufl. v. georg jensch [u. a.] edition.

[6] Christian Bird, Tim Menzies, and Thomas Zimmermann, editors. *The art and science of analyzing software data*. Morgan Kaufmann/Elsevier, Amsterdam and Boston, 2015.

[7] Brooks. No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.

[8] Cambridge Intelligence. Technology - cambridge intelligence. `https://cambridge-intelligence.com/keylines/technology/`.

[9] Pierre Caserta and O. Zendra. Visualization of the static aspects of software: A survey. *IEEE transactions on visualization and computer graphics*, 17(7):913–933, 2011.

[10] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[11] Marco D'Ambros and Michele Lanza. Bugcrawler: Visualizing evolving software systems. In René Krikhaar, editor, *11th European Conference on Software Maintenance and Reengineering, 2007: CSMR '07 ; 21 - 23 March 2007, Amsterdam, the Netherlands*, pages 333–334, Los Alamitos, Calif., 2007. IEEE Computer Society.

[12] Deutsches Zentrum für Luft- und Raumfahrt e.V. Rce, 02.02.2018. `http://rcenvironment.de/`.

[13] Deutsches Zentrum für Luft- und Raumfahrt e.V. Rce user guide 8.2.0. 2017.

[14] Stephan Diehl. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer-Verlag Berlin Heidelberg, Berlin, Heidelberg, 2007.

[15] Doreen Seider. Open source framework rce: Integration, automation, collaboration. In *4th Symposium on Collaboration in Aircraft Design*, 2014.

[16] Ralf Ebert. Eclipse rcp - entwicklung von desktop-anwendungen mit der eclipse rich client platform.

[17] L. Feijs, R. Krikhaar, and R. van Ommering. A relational approach to support software architecture analysis. *Software: Practice and Experience*, 28(4):371–400, 1998.

[18] Norman E. Fenton and Martin Neil. Software metrics: Roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 357–370, New York, NY, USA, 2000. ACM.

[19] Norman E. Fenton and Shari Lawrence Pfleeger. *Software metrics: A rigorous and practical approach*. Intern. Thomson Computer Press, London, 2. ed. edition, 1997.

[20] Ulf Fildebrandt. *Software modular bauen: Architektur von langlebigen Softwaresystemen Grundlagen und Anwendung mit OSGi und Java*. dpunkt.verlag, s.l., 1. aufl. edition, 2012.

[21] Martin Fowler. *Refactoring: Improving the design of existing code*. The Addison-Wesley object technology series. Addison-Wesley, Reading Mass. u.a., 17. print edition, 2005.

[22] Cigdem Gencel and Onur Demirors. Functional size measurement revisited. *ACM Transactions on Software Engineering and Methodology*, 17(3):1–36, 2008.

[23] Yaser Ghanam and Sheelagh Carpendale. A survey paper on software architecture visualization. 2008.

[24] S. Hamza, S. Sadou, and R. Fleurquin. Measuring qualities for osgi component-based applications. In *2013 13th International Conference on Quality Software*, pages 25–34, 2013.

[25] C. Hofmeister, R. L. Nord, and D. Soni. Describing software architecture with uml. In Patrick Donohoe, editor, *Software Architecture: TC2 First Working IFIP Conference on Software Architecture (WICSA1) 22-24 February 1999, San Antonio,*

*Texas, USA*, IFIP - The International Federation for Information Processing, pages 145–159. Springer, Boston, MA, 1999.

[26] Danny Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE transactions on visualization and computer graphics*, 12(5):741–748, 2006.

[27] Danny Holten and Jarke J. van Wijk. Visual comparison of hierarchically organized data. *Computer Graphics Forum*, 27(3):759–766, 2008.

[28] Michael Hunger. *Neo4j 2.0: Eine Graphdatenbank für alle*, volume 51 of *Schnell + kompakt*. Entwickler.press, Frankfurt, Main, 2014.

[29] IEEE. Ieee 1471-2000 - ieee recommended practice for architectural description for software-intensive systems, 2000. `https://standards.ieee.org/standard/1471-2000.html`.

[30] Pourang Irani and Colin Ware. Diagrams based on structural object perception. In Stefano Levialdi, editor, *Proceedings of the working conference on Advanced visual interfaces*, pages 61–67, New York, NY, 2000. ACM.

[31] ISO/IEC/IEEE. Iso/iec/ieee 42010: Systems and software engineering – architecture description, 2011.

[32] Garima Jaiswal. Comparative analysis of relational and graph databases. *IOSR Journal of Engineering*, 03(08):25–27, 2013.

[33] Rainer Koschke. Software visualization in software maintenance, reverse engineering, and re-engineering: A research survey. *Journal of Software Maintenance and Evolution: Research and Practice*, 15(2):87–109, 2003.

[34] Michele Lanza and Stéphane Ducasse. Understanding software evolution using a combination of software visualization and software metrics. In *In Proceedings of LMO 2002 (Langages et Modèles à Objets*, pages 135–149. Lavoisier, 2002.

[35] M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[36] Carola Lilienthal. *Langlebige Software-Architekturen: Technische Schulden analysieren, begrenzen und abbauen*. dpunkt.verlag, Heidelberg, 2nd ed. edition, 2017.

[37] Jeff McAffer, Paul VanderLei, and Simon Archer. *OSGi and Equinox: Creating highly modular Java systems*. The eclipse series. Addison-Wesley Pearson Education, Upper Saddle River NJ u.a., 2010.

[38] Andrew McNair, Daniel M. German, and Jens Weber. *Visualizing Software Architecture Evolution Using Change-Sets*. 2007.

[39] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

[40] Leonel Merino, Mohammad Ghafari, and Oscar Nierstrasz. Towards actionable visualisation in software development. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation: 2-3 October 2016, Raleigh, North Carolina : proceedings*, pages 61–70, Piscataway, NJ, 2016. IEEE.

[41] Inc. Neo4j. Neo4j's graph query language: An introduction to cypher. `https://neo4j.com/developer/cypher-query-language/`.

[42] Inc. Neo4j. Neo4j cypher refcard 3.3, 27.12.2017. `https://neo4j.com/docs/cypher-refcard/current/`.

[43] T. Panas, R. Berrigan, and J. Grundy. A 3d metaphor for software production visualization. In *Proceedings, Seventh International Conference on Information Visualization: IV 2003, an international conference on computer visualization & graphics applications : 16-18 July, 2003, London, England*, pages 314–319, Los Alamitos, Calif, 2003. IEEE Computer Society.

[44] Thomas Panas, Thomas Epperly, Daniel J. Quinlan, Andreas Sæbjørnsen, and Richard Vuduc. Communicating software architecture using a unified single-view visualization, 2007.

[45] Thomas Panas, Thomas Epperly, Daniel Quinlan, Andreas Saebjornsen, and Richard Vuduc. Communicating software architecture using a unified single-view visualization. In *12th IEEE International Conference on Engineering Complex Computer Systems, 2007: ICECCS 2007 ; 11 - 14 July 2007, Auckland, New Zealand ; proceedings ; [including] selected papers from the UML & AADL 2007 Workshop*, pages 217–228, Los Alamitos, Calif., 2007. IEEE Computer Society.

[46] M. Petre. Mental imagery, visualisation tools and team work. *Second Program Visualization Workshop*, 2002.

[47] Marian Petre and Ed de Quincey. A gentle overview of software visualization. 2006.

[48] Philipp Hauer. Java classloader: Wesen, funktionsweise und anwendung. `https://www.philipphauer.de/study/se/classloader.php`.

[49] Roger S. Pressman. *Software engineering: A practitioner's approach*. McGraw-Hill Education, New York, NY, eighth edition edition, 2015.

[50] Jyothi Priya and HS, Mohammad Sharif KB Badri. Software architecture visualization.

[51] Ralf Reussner and Wilhelm Hasselbring, editors. *Handbuch der Software-Architektur*. dpunkt.-Verl., Heidelberg, 2., überarb. und erw. aufl. edition, 2009.

[52] George G. Robertson, Stuart K. Card, and Jack D. Mackinlay. Information visualization using 3d interactive animation. *Communications of the ACM*, 36(4):57–71, 1993.

[53] Ian Robinson. *Graph databases: New opportunities for connected data*. O'Reilly, Sebastopol, CA, second edition edition, 2015.

[54] C. Russo dos Santos, P. Gros, P. Abel, D. Loisel, N. Trichaud, and J. P. Paris. Metaphor-aware 3d navigation. In *Information Visualization (InfoVis 2000): 2000 IEEE Symposium*, pages 155–165, s.l. and Los Alamitos, Jan. 2000. I E E E Imprint.

[55] Johannes Siedersleben. *Moderne Softwarearchitektur: Umsichtig planen, robust bauen mit Quasar*. dpunkt-Verl., Heidelberg, 1. aufl., korr. nachdr edition, 2006.

[56] Andre L.C. Tavares and Marco Tulio Valente. A gentle introduction to osgi. *ACM SIGSOFT Software Engineering Notes*, 33(5):1, 2008.

[57] Alfredo R. Teyseyre and Marcelo R. Campo. An overview of 3d software visualization. *IEEE transactions on visualization and computer graphics*, 15(1):87–105, 2009.

[58] Juraj Vincur, Pavol Navrat, and Ivan Polasek. Vr city: Software analysis in virtual reality environment. In *2017 IEEE International Conference on Software Quality, Reliability and Security (companion volume): QRS-C 2017 : proceedings : 25-29 July 2017, Prague, Czech Republic*, pages 509–516, Piscataway, NJ, 2017. IEEE.

[59] Matthew Ward, Georges G. Grinstein, and Daniel Keim. *Interactive data visualization: Foundations, techniques, and applications*. Peters, Natick Mass., 2010.

[60] Gerd Wütherich, Nils Hartmann, Bernd Johannes Kolb, and Matthias Lübken. *Die OSGi Service Platform: Eine Einführung mit Eclipse Equinox*. dpunkt, s.l., 1. aufl. edition, 2015.