Universität Leipzig
Wirtschaftswissenschaftliche Fakultät
Institut für Wirtschaftsinformatik
Betreuender Hochschullehrer: Prof. Dr. Ulrich Eisenecker
Betreuender Assistent: Dr. Richard Müller

Thema

Konzeption und prototypische Implementierung eines web-basierten Dashboards zur Softwarevisualisierung

Masterarbeit zur Erlangung des akademischen Grades
Master of Science – Wirtschaftsinformatik

vorgelegt von: Mewes, Tino

Leipzig, den 11.10.2018

Abstract

Der Schwerpunkt dieser Arbeit liegt auf der Konzeption sowie prototypischen Implementierung eines web-basierten Dashboards zur Softwarevisualisierung. Ziel der Arbeit ist es, ein Dashboard zu entwickeln, welches Informationen eines Softwareprojekts dynamisch aus einer Graphdatenbank visualisiert und Projekteitern aufgabenbezogen zur Entscheidungsunterstützung darstellt. Derzeit existiert keine Softwarelösung, die diesen Anforderungen vollumfänglich gerecht wird. Es existieren jedoch bereits Bibliotheken und Softwaresysteme, welche Teilaspekte zu einer möglichen Gesamtlösung beitragen können. Diese können bei der prototypischen Implementierung von Nutzen sein und müssen daher beachtet werden. Um die Ziele der Arbeit zu erreichen, werden verschiedene Forschungsmethoden angewandt. Es wird eine Literaturrecherche durchgeführt, mit dem Ziel, typische Aufgaben von Projektleitern im Bereich Software Engineering zu identifizieren. Um die vom Dashboard zu unterstützenden Aufgaben ableiten zu können, werden außerdem verschiedene existierende Dashboard-Werkzeuge analysiert. Mithilfe der gewonnenen Ergebnisse wird das Dashboard konzipiert und prototypisch implementiert. Durch eine Fallstudie anhand von Open-Source-Projekten wird das Dashboard abschließend evaluiert.

Schlagwörter: Dashboard, Softwarevisualisierung, jQAssistant, Neo4j, React, D3.js

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Abkürzungsverzeichnis

API
Application Programming Interface [50]

CI
Continuous Integration [16]

CSS
Cascading Style Sheet [6, 30]

D3
Data Driven Documents [6]

DOM
Document Object Model [6]

HTML
Hypertext Markup Language [6, 25]

HTTP
Hypertext Transfer Protocol [5]

ISO
International Organization for Standardization [9]

JSON
JavaScript Object Notation [27]

LOC
Lines of Code [61]

MVC
Model View Controller [26–28]

MVVM
Model View ViewModel [26–28]

NPM
Node Package Manager [31, 60, 61]

SVG
Scalable Vector Graphics [6]

URL
Uniform Resource Locator [21, 35]

1 Einleitung

Dieses Kapitel motiviert das Thema der Arbeit und formuliert die entsprechende Problemstellung. Darüber hinaus beschreibt es die angestrebten Ziele der Arbeit. Abgeschlossen wird das Kapitel mit der Beschreibung des methodischen Vorgehens.

1.1 Motivation und Problemstellung

Softwaretechnik (engl. *software engineering*) hat die zielorientierte Bereitstellung und systematische Verwendung von Prinzipien, Methoden und Werkzeugen für die arbeitsteilige, ingenieurmäßige Entwicklung und Anwendung von umfangreichen Softwaresystemen zum Inhalt [Balzert 2009, S. 17]. Abbildung 1.1 zeigt die Problematik, dass während der Entwicklung und Wartung von Software viele Stakeholder viele Artefakte erzeugen und damit ein hohes Maß an Komplexität entsteht.

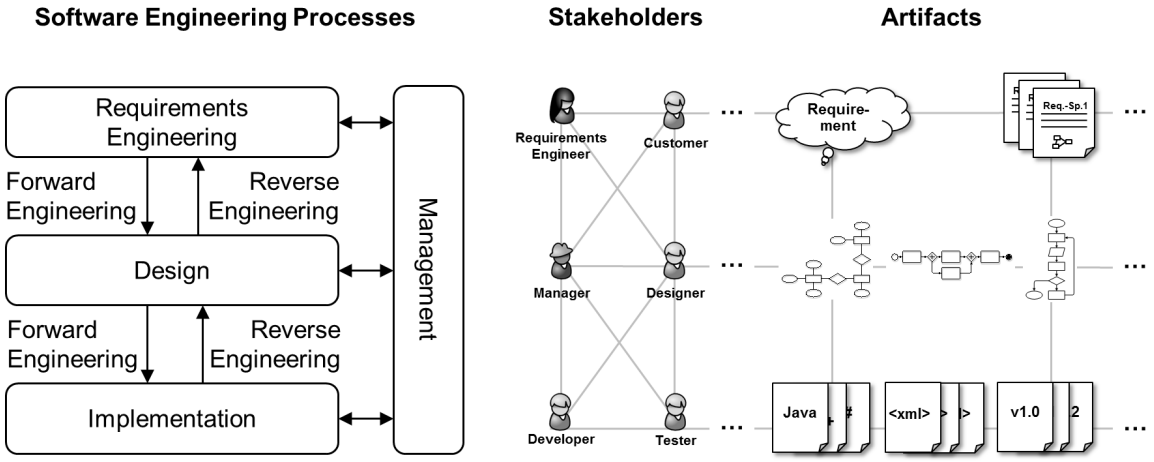

Abbildung 1.1: Software-Engineering-Prozesse mit Stakeholdern und Artefakten [Müller 2015, S. 1]

Im Rahmen des Software-Engineering-Managements werden Entwicklungs- und Wartungsprozesse geplant, koordiniert, gemessen, aufgezeichnet, überwacht sowie entsprechende Berichte erzeugt. Eine Möglichkeit zur Unterstützung des Software-Engineering-Managements bietet die Softwarevisualisierung beziehungsweise ein Dashboard zur Softwarevisualisierung. Daher wird ein Dashboard zur Softwarevisualisierung entwickelt, welches zukünftig von Projektleitern eingesetzt werden kann, um Aufgaben wie die Erhebung relevanter Metriken oder das Ressourcenmanagement zu unterstützen.
Dazu werden die Informationen von Softwareartefakten zunächst mithilfe des Werkzeugs *jQ-Assistant* extrahiert. Dabei werden Artefakte des Softwaresystems wie beispielsweise dessen

Bytecode, Testergebnisse und Analyseberichte mittels *jQAssistant* gescannt. Die Informationen dieser Artefakte werden extrahiert und in einer *Neo4j*-Graphdatenbank gespeichert. Auf die gespeicherten Daten kann mithilfe der Abfragesprache *Cypher* zugegriffen werden. Was derzeit allerdings noch fehlt, ist eine sinnvolle Darstellung beziehungsweise Visualisierung der Ergebnisse der Abfragen. Diese Lücke soll durch die Entwicklung eines web-basierten Dashboards, welches gezielt Aufgaben von Projektleitern unterstützt, geschlossen werden.
Um einen möglichst unkomplizierten Zugang zum Dashboard zu ermöglichen, wird es als Webanwendung (engl. *web application*, kurz *Web App*) entwickelt. Das Dashboard kann somit direkt über den Browser aufgerufen werden und funktioniert auf jedem internetfähigen Gerät, das diesen Browser unterstützt.
Im Rahmen der Entwicklung des Dashboards stehen folgende Qualitätskriterien im Mittelpunkt.

- **Erweiterbarkeit:** Das Dashboard soll einfach um weitere Komponenten erweiterbar sein.
- **Installierbarkeit:** Das Dashboard soll einfach und schnell zu installieren sein.
- **Gebrauchstauglichkeit:** Das Dashboard soll intuitiv benutzbar sein.
- **Wartbarkeit:** Die einzelnen Bestandteile beziehungsweise Bibliotheken des Dashboards sollen einfach aktualisierbar sein.

1.2 Zielstellung

Ziel dieser Arbeit ist die Konzeption und prototypische Implementierung eines web-basierten analytischen Dashboards zur Softwarevisualisierung. Das Dashboard dient Projektleitern zur Entscheidungsunterstützung. Es soll Informationen eines Softwareprojekts dynamisch aus einer Graphdatenbank abbilden. Das Dashboard soll erweiterbar sein und die Informationen aufgabenbezogen für Projektleiter darstellen. Das System soll interaktiv arbeiten und eine intuitive Bedienung durch eine Webschnittstelle realisieren. Der Mehrbenutzerbetrieb ist im Rahmen dieser Arbeit nicht vorgesehen.
Nach abgeschlossener Evaluation soll der Prototyp als Grundlage für die Weiterentwicklung des Systems dienen. Zu diesem Zweck sollte das System möglichst erweiterbar konzipiert werden. So sollten sich zum Beispiel neue Visualisierungen unkompliziert hinzufügen lassen.

1.3 Methodisches Vorgehen

Aufbauend auf dem Methodenspektrum der Wirtschaftsinformatik nach Wilde und Hess [2007] wird in dieser Arbeit die Methode des Prototyping als wissenschaftliche Vorgehensweise verwendet. Dazu wird eine Vorabversion eines Anwendungssystems entwickelt und

evaluiert. Sowohl die Entwicklung als auch die Evaluation können dabei neue Erkenntnisse generieren.
Die Gliederung der Inhalte erfolgt nach *arc42*, einer frei verfügbaren Dokumentationsvorlage für Softwarearchitekturen, nach der im Rahmen dieser Arbeit ein Prototyp entwickelt wird. Durch die Verwendung dieser Vorlage können maßgebliche Entwurfsentscheidungen der zu konzipierenden Infrastruktur nachvollzogen werden.
Für fast jedes Softwareprojekt muss mehrfach überlegt werden, was bezüglich der Softwarearchitektur bedacht, entschieden und dokumentiert werden muss. Dieser Aufwand kann durch *arc42* verringert werden. Die Vorlage gibt dazu eine standardisierte Gliederung für Softwarearchitekturbeschreibungen vor.
Zunächst werden im Rahmen der Einführung die Aufgabenstellung, Qualitätsziele, Anforderungen sowie die Stakeholder schriftlich festgehalten. Anschließend erfolgt die Kontextabgrenzung, also die Festlegung des Umfelds, in dem das System arbeitet. Im Rahmen der fachlichen Kontextabgrenzung wird dazu eine Literaturrecherche zu typischen Aufgaben der Stakeholder des zu konzipierenden Dashboards durchgeführt. Zur Identifikation von Use-Cases sowie zur Ableitung eines Benutzungskonzepts werden anschließend existierende Lösungen aus der Praxis analysiert. Danach werden die Randbedingungen für die zu konzipierende Softwarearchitektur festgelegt. Die Risiken und mögliche technische Schulden werden im anschließenden Schritt erläutert.
Zentrale, prägende und wichtige Entscheidungen werden anschließend im Schritt „Entwurfsentscheidungen“ getroffen. Dazu werden die fünf aktuell populärsten web-basierten Frameworks *React*, *Vue.js*, *Angular*, *Backbone.js* und *Ember.js* im Zuge einer Internetrecherche miteinander verglichen, um das geeignetste Framework für die Implementierung des Dashboards zu finden. Auf den bisherigen Punkten aufbauend wird eine Lösungsstrategie entwickelt und die fundamentalen Lösungsansätze werden näher beschrieben.
Es folgt die prototypische Implementierung des web-basierten Dashboards auf der Grundlage der konzipierten Architektur. Anschließend wird der entwickelte Prototyp anhand der Open-Source-Software *Spring Petclinic*[1] sowie *jUnit*[2] evaluiert, um zu zeigen, dass die Implementierung auch für andere Projekte korrekt funktioniert. Abgeschlossen wird die Arbeit durch ein Fazit sowie einen Ausblick auf mögliche Erweiterungen der entwickelten Lösung.
Im Glossar werden außerdem fachliche Begriffe erläutert, die im Rahmen der Systementwicklung und des -einsatzes eine wesentliche Rolle spielen.

[1] `https://github.com/spring-projects/spring-petclinic`, Zugriff am: 16.01.2018
[2] `https://github.com/junit-team/junit4`, Zugriff am: 16.01.2018

2 Grundlagen

In diesem Kapitel werden die Grundlagen des zu konzipierenden Dashboards betrachtet. Da das Dashboard für die Visualisierung von Softwareprojekten eingesetzt werden soll, wird einleitend die Softwarevisualisierung thematisiert und deren Kategorisierung näher erläutert. Anschließend wird das Werkzeug *jQAssistant* beschrieben, welches die Informationen eines Softwaresystems zunächst extrahiert. Im Zuge dessen werden die extrahierten Informationen in einer *Neo4j*-Graphdatenbank gespeichert. Auf *Neo4j* wird im nachfolgenden Abschnitt eingegangen. Da es aktuell an web-basierten Lösungen zur Visualisierung der Ergebnisse der Graphdatenbank-Abfragen fehlt, werden anschließend Webanwendungen und deren Vorteile betrachtet. Im nächsten Abschnitt wird die *JavaScript*-Bibliothek *D3.js* beschrieben, die die Visualisierung von Datensätzen im Browser ermöglicht. Abschließend werden Dashboards sowie deren Kategorisierungsmöglichkeiten näher betrachtet.

2.1 Softwarevisualisierung

Die Softwarevisualisierung beschäftigt sich mit der statischen oder animierten Darstellung von Informationen über Softwaresysteme in 2D oder 3D [Diehl 2002]. Nach Diehl [2007] können die Informationen dabei die Struktur, das Verhalten oder die Historie des Systems beschreiben.
Die Struktur bezieht sich auf die statischen Teile und Beziehungen des Systems, das heißt alle Informationen, die ohne die Ausführung von Software gewonnen werden können.
Das Verhalten bezieht sich auf Informationen, die durch die Ausführung der Software mit realen oder simulierten Daten gewonnen werden können. Die Ausführung kann als eine Sequenz von Systemzuständen gesehen werden, wobei ein Systemzustand sowohl den aktuellen Quellcode als auch die Daten des Systems enthält. Abhängig von der Programmiersprache kann die Ausführung auf einer höheren Abstraktionsebene als Funktionen, die andere Funktionen aufrufen, oder als Objekte, die mit anderen Objekten kommunizieren, betrachtet werden.
Die Historie bezieht sich auf Informationen, die im Rahmen des Entwicklungsprozesses von Software entstehen und betont insbesondere die Tatsache, dass Quellcode im Laufe der Zeit verändert wird, um die Funktionalität des Systems zu erweitern oder um Fehler zu beseitigen.
Die Ziele der Softwarevisualisierung beinhalten das Verstehen von Softwaresystemen und Algorithmen sowie die Analyse von Softwaresystemen zur Entdeckung von Anomalien, beispielsweise durch die Darstellung von sogenannten Hotspots, also Bereichen eines Software-

systems, in denen ein hoher Anteil von ausgeführten Befehlen auftritt oder die meiste Zeit während der Systemausführung verbracht wird.

2.2 jQAssistant

Softwaresysteme wachsen oft so schnell, dass es ab einer gewissen Größe erhebliche Aufwände verursacht, Informationen über entstandene Strukturen zu erhalten. *jQAssistant* ist ein Plugin für das Build-Management-Werkzeug *Apache Maven*, welches die Artefakte einer *Java*-Anwendung analysiert. Dazu wird der Bytecode des Softwaresystems mittels *jQAssistant* gescannt, die Informationen extrahiert und in einer *Neo4j*-Graphdatenbank gespeichert. Diese Informationen können mithilfe von *jQAssistant*-Konzeptregeln[3] mit weiteren Daten, wie beispielsweise Beschriftungen, Eigenschaften oder Beziehungen, angereichert werden, um den Prozess des Verfassens von Abfragen der *Neo4j*-Graphdatenbank zu vereinfachen. Durch weitere Plugins kann der Graph nochmals, beispielsweise um Informationen aus der Git-Historie, erweitert werden.

2.3 Neo4j

Eine typische Möglichkeit der Darstellung von Software sind Graphen [Diehl 2014]. Ein Graph besteht aus Knoten und Kanten, den Verbindungen zwischen den Knoten. Sowohl Knoten als auch Kanten können Eigenschaften besitzen. Um stark vernetzte Informationen in Form von Graphen abzubilden, können sogenannte Graphdatendanken verwendet werden. Die populärste Graphdatenbank ist *Neo4j*[4]. In *Neo4j* wird analog zu Graphen jede Information entweder als Kante, als Knoten oder als Eigenschaft gespeichert. Im Zuge der Speicherung der Informationen über ein Softwaresystem mittels *Neo4j* werden unter anderem Dateien, Klassen, Pakete und Methoden der Software als Knoten angelegt. Kanten werden durch Schlüsselwörter wie CONTAINS, DEPENDS_ON, INVOKES, DECLARES, IMPLEMENTS und RETURNS abgebildet. Die Abfrage des Graphen erfolgt mithilfe der Abfragesprache *Cypher*. Mittels *Cypher* lassen sich über diese Strukturen sehr flexible und performante Abfragen formulieren.

2.4 Webanwendungen und web-basierte Frameworks

Eine Webanwendung stellt Funktionen sowie dynamische Inhalte über das Internetprotokoll *Hypertext Transfer Protocol (HTTP)* zur Verfügung. Dazu werden auf einem Server Dokumente und Benutzungsoberflächen wie Bedienelemente oder Eingabemasken erzeugt und an

[3] `http://buschmais.github.io/jqassistant/doc/1.4.0/#_concepts`, Zugriff am: 23.09.2018
[4] `https://db-engines.com/de/ranking/graph+dbms`, Zugriff am: 30.12.2017

entsprechende Clientprogramme beziehungsweise Browser ausgeliefert[5]. Webanwendungen werden gewöhnlich auf der Grundlage von web-basierten Frameworks entwickelt.
Web-basierte Frameworks oder kurz *Webframeworks* sind ein fester Bestandteil einer Webanwendung, die in der Regel das Lesen der Daten von einem Server sowie das Rendern, also das Darstellen der angeforderten Inhalte, in einem Browser übernehmen.
Webframeworks sind darauf ausgelegt, sehr schnell lauffähige Webanwendungen zu erstellen. Dazu bieten aktuelle *Webframeworks* meist Template-Mechanismen für die automatische Generierung von Dokumenten, welche aus *Hypertext Markup Language (HTML)*, *Cascading Style Sheets (CSS)* und *JavaScript* bestehen. Zudem ermöglichen *Webframeworks* durch die Verwendung von Architekturmustern eine saubere Trennung von Präsentation und Logik.
Durch *Webframeworks* können sich wiederholende Tätigkeiten vereinfacht und die Wiederverwendung von Quellcode sowie die Dokumentation der Softwareentwicklung gefördert werden. Durch Konzepte wie *Don't repeat yourself* oder *Konvention vor Konfiguration* bieten Webframeworks einfache, klare und mit wenig Aufwand wartbare Strukturen.
Es existiert eine Vielzahl solcher Frameworks zur Unterstützung der Erstellung von Webanwendungen. Die fünf populärsten web-basierten Frameworks sind aktuell *React*, *Vue.js*, *Angular*, *Backbone.js* und *Ember.js*[6]. Im Rahmen dieser Arbeit werden diese noch genauer betrachtet und miteinander verglichen, um die am besten geeignete Grundlage für die Implementierung eines Prototyps zu finden.

2.5 D3.js

Große Datenmengen in einem angemessenen Format zu präsentieren ist eine Herausforderung. Doch durch die ständige Weiterentwicklung können immer größere und bessere Visualisierungen erstellt werden. Mittlerweile gibt es einige *JavaScript*-Bibliotheken, die sich für solche Zwecke bestens eignen. *D3.js* ist eine dieser *JavaScript*-Bibliotheken, die die Bearbeitung und Visualisierung von Datensätzen im Browser ermöglichen. Dabei handelt es sich um eine der populärsten Bibliotheken zur Datenvisualisierung. Die Abkürzung *D3* steht für *Data Driven Documents*, also datengetriebene Dokumente. Aktuell nutzen circa 240.000 Webseiten *D3.js*[7]. Damit können sehr leicht Grafiken im Dateiformat *Scalable Vector Graphics (SVG)* erstellt und die Struktur des *Document Object Model (DOM)* durch Datensätze manipuliert werden. Dafür nutzt *D3.js* die aktuellen *HTML*-, *CSS*- und *SVG*-Web-Standards.

[5] `https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Grundschutz/Download/IT-GS-Bausteine/Webanwendungen/Baustein_Webanwendungen-B5_21.pdf?__blob=publicationFile`, Zugriff am: 16.01.2018
[6] `https://bestof.js.org/tags/framework`, Zugriff am: 01.12.2017
[7] `https://www.wappalyzer.com/applications/d3`, Zugriff am: 03.01.2018

2.6 Dashboards

Ein Dashboard ist eine visuelle Anzeige der wichtigsten Informationen, die benötigt werden, um ein oder mehrere Ziele zu erreichen. Die Informationen eines Dashboards werden auf einem einzigen Bildschirm angeordnet, so dass diese auf einen Blick überwacht werden können [Few 2004]. Sie können Menschen dabei helfen, Trends, Muster und Anomalien visuell zu erkennen, zu begründen, was sie sehen und ihnen zu effektiven Entscheidungen verhelfen [Brath 2004]. Das Dashboard in Abbildung 2.1 zeigt eine Sammlung von Verkaufsmaßnahmen zur Analyse der Produktleistung nach Kategorie.

Abbildung 2.1: Darstellung von Verkaufsmaßnahmen zur Analyse der Produktleistung nach Kategorie in Form eines Dashboards [Few 2006, S. 17]

In der Regel werden die Informationen in einem Dashboard als Kombination aus Text und Grafik, jedoch mit Schwerpunkt auf Grafiken, dargestellt. Ein Dashboard dient zur Überwachung von Informationen auf einen Blick und sollte daher auf einen einzigen Computerbildschirm passen.

Es existieren verschiedene Möglichkeiten der Kategorisierung von Dashboards. Nach Few [2006] lassen diese sich am besten nach ihrer Rolle unterscheiden, also der Art der Geschäftsaktivität, die unterstützt wird. Dabei werden Dashboards in die drei Rollen strategisch, analytisch und operativ aufgeschlüsselt. Diese Methode bezieht sich auf Unterschiede im visuellen Design des Dashboards.

Dashboards strategischer Natur können beispielsweise Manager auf den einzelnen Ebenen in einer Organisation unterstützen. Dashboards dieser Art konzentrieren sich auf Prognosen, um den Weg in die Zukunft zu beleuchten. Extrem einfache Anzeigemechanismen funktionieren am besten für diese Art von Dashboards. Angesichts des Ziels einer langfristigen strategischen Ausrichtung und nicht der unmittelbaren Reaktion auf rasante Änderungen benötigen

diese Dashboards keine Echtzeitdaten. Vielmehr profitieren sie von statischen Momentaufnahmen, die monatlich, wöchentlich oder täglich gemacht werden.
Dashboards für analytische Zwecke erfordern einen anderen Entwurfsansatz. In diesen Fällen erfordert die Information oft einen größeren Kontext, zum Beispiel umfangreiche Vergleiche oder eine umfangreichere Historie. Wie strategische Dashboards profitieren auch analytische Dashboards von statischen Daten, die sich nicht ständig von einem Moment zum nächsten ändern. Analytische Dashboards sollten Interaktionen mit den Daten unterstützen, um nicht nur zu sehen, was vor sich geht, sondern auch, um die Ursachen zu untersuchen. In einem analytischen Dashboard genügt es beispielsweise nicht zu sehen, dass die Verkäufe eines Onlineshops sinken. Es sollte zusätzlich auf bestimmte Muster und Merkmale aufmerksam machen, um herauszufinden, was die Abnahme verursacht und wie sie korrigiert werden könnte.
Abschließend werden die operativen Dashboards betrachtet. Diese zeichnen sich durch ihre dynamische und unmittelbare Natur aus. Bei der Überwachung von Vorgängen werden zu jedem Zeitpunkt Aufmerksamkeit und Reaktion benötigt. Wenn die Zugriffszahlen der Organisationswebseite plötzlich auf die Hälfte der normalen Werte sinken, müssen die zuständigen Mitarbeiter sofort benachrichtigt werden. Wie bei strategischen Dashboards müssen die Anzeigemedien in operativen Dashboards sehr einfach sein. In einem Notfall müssen die entsprechenden Reaktionen klar und einfach sein, um keine Fehler zu verursachen. Im Gegensatz zu strategischen Dashboards müssen operative Dashboards die Möglichkeit bieten, sofort auf das Problem aufmerksam zu machen, wenn ein Vorgang außerhalb der akzeptablen Leistungsschwelle liegt.

3 Konzeption

Dieses Kapitel beschreibt das Konzept eines web-basierten Dashboards zur Softwarevisualisierung, welches den in Abschnitt 1.1 besprochenen Anforderungen gerecht wird. Hierzu werden nach der Festlegung der Anforderungen zunächst im Rahmen einer Literaturrecherche die typischen Aufgaben von Projektleitern identifiziert. Darauffolgend werden existierende Dashboard-Werkzeuge analysiert. Das Ergebnis der Literaturrecherche dient dabei als Checkliste für die Überprüfung, welche typischen Aufgaben von Projektleitern jeweils vom untersuchten Dashboard-Werkzeug unterstützt werden. Aus dieser Ergebnismenge werden die vom zu konzipierenden Dashboard unterstützten Aufgaben abgeleitet. Anschließend werden ein Use-Case-Diagramm der zu unterstützenden Aufgaben sowie die Mockups der Benutzungsschnittstelle des Dashboards erstellt und beschrieben. Außerdem werden die Randbedingungen für die Implementierung des Dashboards sowie die Risiken und eventuelle technische Schulden schriftlich festgehalten. Anschließend werden die Entwurfsentscheidungen getroffen und eine Lösungsstrategie entwickelt.

3.1 Mission Statement

Es soll ein web-basiertes System erstellt werden, das Informationen eines Softwareprojekts dynamisch aus einer Graphdatenbank in Form eines erweiterbaren Dashboards für Projektleiter aufgabenbezogen visualisiert.

3.2 Architekturziele

In diesem Abschnitt werden die Architekturziele des Dashboards thematisiert. Dazu werden zunächst die konkreten Qualitätskriterien festgelegt. Im nachfolgenden Abschnitt werden außerdem typischen Aufgaben von Projektleitern im Rahmen einer Literaturrecherche identifiziert und anschließend existierende Dashboard-Werkzeuge analysiert, um weitere Ziele ableiten zu können.
Im Rahmen der Entwicklung des Dashboards stehen folgende Qualitätskriterien aus der Norm International Organization for Standardization (ISO) 25010[8] im Mittelpunkt.

- **Erweiterbarkeit:** Das Dashboard soll einfach um weitere Komponenten erweiterbar sein.
- **Installierbarkeit:** Das Dashboard soll einfach und schnell zu installieren sein.

[8] `https://www.beuth.de/de/norm/iso-iec-25010/140332975`, Zugriff am: 16.01.2018

- **Gebrauchstauglichkeit:** Das Dashboard soll intuitiv benutzbar sein.
- **Wartbarkeit:** Das Dashboard und einzelne Bestandteile/Bibliotheken sollen einfach aktualisierbar sein.

3.3 Kontextabgrenzung

In diesem Abschnitt wird das Umfeld des Dashboards beschrieben. Dabei wird festgelegt, für welche Benutzer das Dashboard konzipiert wird.

3.3.1 Fachlicher Kontext

Im Rahmen der fachlichen Kontextabgrenzung wird eine Literaturrecherche zu typischen Aufgaben von Projektleitern durchgeführt. Anschließend werden ein Use-Case-Diagramm der zu unterstützenden Aufgaben sowie einige Mockups der Benutzungsschnittstelle entworfen.

3.3.1.1 Literaturrecherche zu typischen Aufgaben von Projektleitern

Da die Informationen im Dashboard aufgabenbezogen visualisiert werden sollen, stellt sich zunächst die Frage, welche allgemeingültigen Aufgaben Projektleitern obliegen. Um diese Frage zu beantworten, wird eine Literaturrecherche durchgeführt. Diese Literaturrecherche hat das Ziel, typische Aufgaben von Projektleitern im Bereich Software Engineering zu identifizieren. Die dazu verwendeten Quellen sind nachfolgend aufgelistet.

- Project Management Body of Knowledge [Project Management Institute 2017]
- Software Engineering Body of Knowledge [Bourque 2014]
- An Examination of Software Engineering Work Practices [Singer 2010]
- Project management: a systems approach to planning, scheduling, and controlling [Kerzner 2017]
- A task oriented view of software visualization [Maletic 2002]

Die Werke Project Management Institute [2017] und Bourque [2014] bieten eine gute Zusammenfassung etablierten Wissens und bilden die Grundlage der Literaturrecherche. Ergänzt wird diese Grundlage durch die wissenschaftlichen Artikel Singer [2010], Kerzner [2017] sowie Maletic [2002], um die Aufgaben, mit denen sich ein Projektleiter typischerweise beschäftigt, möglichst umfassend identifizieren zu können. Die Ergebnisse der Literaturrecherche dienen später als Checkliste, um die Aufgabenunterstützung von existierenden Dashboard-Werkzeugen zu untersuchen.
Im Kontext dieser Arbeit wird unter einem Projektleiter ein technischer Leiter (engl. *technical lead*) verstanden. Ein technischer Leiter ist ein Entwickler, der für die Leitung eines Entwicklungsteams verantwortlich ist [Kua 2014]. Abbildung 3.1 verdeutlicht den Standpunkt eines technischen Leiters innerhalb eines Entwicklerteams.

Abbildung 3.1: Standpunkt des technischen Leiters innerhalb eines Entwicklerteams [Kua 2014]

Ein technischer Leiter vereint somit die Funktionen eines Projektleiters sowie eines Entwicklers. Daher wird die genannte Literatur zunächst sowohl nach typischen Aufgaben von Entwicklern als auch von Projektleitern durchsucht. Der Begriff Aufgabe ist im Kontext des Software-Engineering-Managements als die kleinste betrachtete Arbeitseinheit definiert [Angermeier 2005]. Tabelle 3.1 fasst die identifizierten Aufgaben zusammen.

Stakeholder	Aufgabe	Quelle
Entwickler	Programmieren	[Bourque 2014, S. 3-1 ff.] [Singer 2010] [Maletic 2002]
	Fehlerbehebung	[Bourque 2014, S. 3-1 ff.] [Maletic 2002] [Singer 2010]
	Testen	[Bourque 2014, S. 3-1] [Maletic 2002] [Singer 2010]
	Dokumentation der Softwareentwicklung	[Singer 2010]
	Performanceanalyse	[Bourque 2014, S. 3-6]
	Wiederverwendung vorhandener Software	[Bourque 2014, S. 3-3 ff.]
	Verwendung von Schnittstellen	[Bourque 2014, S. 3-8]
	Systemerweiterung	[Singer 2010]
	Planung von Entwicklungs- und Wartungsprozessen	[Bourque 2014, S. 5-9]
	Problemanalyse	[Bourque 2014, S. 5-7]
	Systemanalyse/-verständnis	[Bourque 2014, S. 5-5]
	Überprüfung von Änderungen	[Bourque 2014, S. 5-8]
	Clone Detection	[Bourque 2014, S. 5-4]
Projektleiter	Prozessplanung	[Maletic 2002]
	Schätzung von Aufwand, Zeit und Kosten	[Bourque 2014, S. 7-6]
	Zuweisung von Mitteln	[Bourque 2014, S. 7-6]
	Risikomanagement	[Bourque 2014, S. 7-6]
	Qualitätsmanagement	[Bourque 2014, S. 7-6]
	Implementierung von Messvorgängen	[Bourque 2014, S. 7-7]
	Prozessüberwachung	[Bourque 2014, S. 7-7]
	Prozesskontrolle	[Bourque 2014, S. 7-8]
	Berichterstattung	[Bourque 2014, S. 7-8]
	Festlegung der Erfüllung der Anforderungen	[Bourque 2014, S. 7-8]
	Performanceüberprüfung und -evaluation	[Bourque 2014, S. 7-9]
	Festlegung des Projektabschlusses	[Bourque 2014, S. 7-9]
	Stilllegungsaktivitäten	[Bourque 2014, S. 7-9]

Tabelle 3.1: Typische Aufgaben von technischen Leitern

Da der Fokus des Dashboards auf der Unterstützung von Projektleitern in Form von technischen Leitern liegt, werden die Aufgaben, die sich auf das aktive Programmieren beziehen, nicht weiter betrachtet, da diese Aufgaben eher die Entwickler innerhalb des Entwickler-

teams und nicht die Projektleiter betreffen. Auch die Softwareanschaffung wird nicht näher betrachtet, da diese Aufgabe nicht durch Dashboard-Werkzeuge unterstützt wird.
Um den Umfang von Tabelle 3.1 zu reduzieren und diese als Checkliste für die Analyse existierender Dashboard-Werkzeuge nutzbar zu machen, werden die identifizierten Aufgaben nun in Wissensbereiche (engl. *Knowledge Areas*), also Oberbegriffe zugehöriger Aufgaben, zusammengefasst. In Project Management Institute [2017, S. 41 f.] sind die zehn Wissensbereiche Integrations-, Zeitplan-, Kosten-, Qualitäts-, Ressourcen-, Kommunikations-, Risiko-, Umfangs-, Beschaffungs- sowie Stakeholdermanagement definiert. Ein Projektleiter im Bereich der Softwareentwicklung und -wartung benötigt zudem einen Überblick des zu entwickelnden Systems. Daher wird zusätzlich der Wissensbereich des Architekturüberblicks eingeführt. In Tabelle 3.2 wird jeder identifizierten Aufgabe einer dieser Wissensbereiche zugeordnet.

Wissensbereich	**Aufgabe**
Qualitätsmanagement	Fehlerbehebung
	Testen
	Dokumentation der Softwareentwicklung
	Performanceanalyse
	Wiederverwendung vorhandener Software
	Verwendung von Schnittstellen
	Problemanalyse
	Überprüfung von Änderungen
	Clone Detection
	Festlegung der Erfüllung der Anforderungen
	Berichterstattung
	Implementierung von Messvorgängen
	Performanceüberprüfung und -evaluation
	Festlegung des Projektabschlusses
	Stilllegungsaktivitäten
Architekturüberblick	Planung von Entwicklungs- und Wartungsprozessen
	Systemanalyse/-verständnis
	Prozessplanung
Ressourcenmanagement	Schätzung von Aufwand, Zeit und Kosten
	Zuweisung von Mitteln
	Planung von Entwicklungs- und Wartungsprozessen
Risikomanagement	Prozessüberwachung
	Prozesskontrolle
	Planung von Entwicklungs- und Wartungsprozessen

Tabelle 3.2: Aufgaben der Stakeholder zur Verwendung als Checkliste für Tools

Die Wissensbereiche, welchen keine der identifizierten Aufgaben zugeordnet wurden, sind in diesem Fall nicht relevant und werden nicht weiter betrachtet. Somit konnten folgende Wissensbereiche identifiziert werden, mit denen sich ein Projektleiter typischerweise beschäftigt:

- Architekturüberblick
- Ressourcenmanagement
- Risikomanagement
- Qualitätsmanagement

Diese Liste von Wissensbereichen wird in Abschnitt 3.3.1.2 als Checkliste verwendet, um zu überprüfen, welche Wissensbereiche von den jeweiligen Dashboard-Werkzeugen unterstützt werden.

3.3.1.2 Analyse existierender Dashboard-Werkzeuge

Um die vom Dashboard zu unterstützenden Aufgaben ableiten zu können, werden in diesem Abschnitt verbreitete existierende Dashboard-Werkzeuge für die Softwareanalyse in einem Browser hinsichtlich folgender Kriterien analysiert:

- Unterstützte Programmiersprachen
- Integration
- Unterstützte Aufgaben
- Besonderheiten

Auf Basis der kostenfreien Versionen der ausgewählten Dashboard-Werkzeuge werden die genannten Kriterien abgearbeitet.
Teamscale[9] ist ein Werkzeug zur Qualitätsanalyse von Quellcode. Dazu können Dashboards mit integrierten Widgets erstellt werden, welche die Qualität des Quellcodes durch verschiedene Visualisierungen beleuchten. Die Ergebnisse der zu analysierenden Kriterien für *Teamscale* fasst Tabelle 3.3 zusammen.

[9] `https://www.cqse.eu/en/products/teamscale/landing`, Zugriff am: 07.05.2018

	Teamscale
Unterstützte Programmiersprachen	*C#*, *Groovy*, *Java*, *JavaScript*, *PHP*, *Python*, *Rust*, *TypeScript*, *Visual Basic .NET*, *Xtend*
Integration	Integration in *Eclipse*, *Microsoft Visual Studio*, *IntelliJ*, *PhpStorm*, *PyCharm* und *WebStorm*
	Versionsverwaltung: *Subversion*, *Git*, Dateisystem
Unterstützte Aufgaben	Qualitätsmanagement: Wiederverwendung vorhandener Software, Clone Detection, Berichterstattung
	Architekturüberblick: Planung von Entwicklungs- und Wartungsprozessen
	Ressourcenmanagement: Unterstützung bei der Zuweisung von Mitteln
Besonderheiten	Konfiguration von individuellen Dashboards
	Nutzung von Widgets wie Kreisdiagrammen, Treemaps sowie Metrik-Tabellen

Tabelle 3.3: Analyse des Dashboard-Werkzeugs *Teamscale*

CodeScene[10] ist ein Dashboard-Werkzeug für die Identifikation von Mustern in der Historie von Quellcode. Dies gibt die Möglichkeit, Prognosen zu erstellen und den Quellcode zu finden, der schwer zu entwickeln und fehleranfällig ist. Die Ergebnisse der zu analysierenden Kriterien für *CodeScene* fasst Tabelle 3.4 zusammen.

	CodeScene
Unterstützte Programmiersprachen	*C*, *C++*, *C#*, *Visual Basic*, *Java*, *Clojure*, *Kotlin*, *Groovy*, *Ruby*, *Python*, *Erlang*, *Scala*, *JavaScript*, *TypeScript*, *Objective-C 2.0*, *Go*, *PHP*, *Apex*, *Swift*
Integration	Einbindung von *Git* zur Analyse von Quellcode
Unterstützte Aufgaben	Qualitätsmanagement: Problemanalyse, Berichterstattung
	Architekturüberblick: Systemanalyse/-verständnis durch diverse Metriken
	Risikomanagement: Erfassung von Hotspots
Besonderheiten	Fokus auf Quellcodeanalyse
	Darstellung von Hotspots mittels geschachtelter Kreise

Tabelle 3.4: Analyse des Dashboard-Werkzeugs *CodeScene*

[10] `https://codescene.io`, Zugriff am: 07.05.2018

SonarQube[11] ist ein Dashboard-Werkzeug für die statische Quellcodeanalyse der technischen Qualität von Quellcode. *SonarQube* analysiert den Quellcode hinsichtlich verschiedener Qualitätsbereiche und stellt die Ergebnisse über eine Webseite dar. Die Ergebnisse der zu analysierenden Kriterien für *SonarQube* fasst Tabelle 3.5 zusammen.

	SonarQube
Unterstützte Programmiersprachen	*C*, *C++*, *JavaScript*, *C#*, *Java*, *COBOL*, *PHP*, *ABAP*, *VB.NET*, *VB6*, *Python*, *RPG*, *Flex*, *Objective-C*, *Swift*
Integration	Mittels der Build-Systeme *Apache Maven*, *Gradle*, *Apache Ant*, *Microsoft Build* sowie *Makefile*
	Mittels der folgenden Systeme für Continuous Integration (CI): *Jenkins*, *Travis CI*, *Bamboo*, *AppVeyor*, sowie *Team City*
Unterstützte Aufgaben	Qualitätsmanagement: Testen, Wiederverwendung vorhandener Software, Clone Detection, Problemanalyse, Optionen zur Problemlösung dokumentieren, Berichterstattung
	Ressourcenmanagement: Unterstützung bei der Zuweisung von Mitteln
	Risikomanagement: Prozessüberwachung
Besonderheiten	Einbindung zahlreicher Metriken
	Darstellung der Testabdeckung mit direkter Anbindung an den Quellcode

Tabelle 3.5: Analyse des Dashboard-Werkzeugs *SonarQube*

Structurizr[12] ist ein Werkzeug, mit dem die Architektur von Softwareprojekten auf Basis von *Java*-Quellcode dokumentiert werden kann. Die Ergebnisse der zu analysierenden Kriterien für *Structurizr* fasst Tabelle 3.6 zusammen.

[11] `https://www.sonarqube.org`, Zugriff am: 07.05.2018
[12] `https://structurizr.com`, Zugriff am: 07.05.2018

	Structurizr
Unterstützte Programmiersprachen	*Java*, *C#*
Integration	Erstellte Diagramme können in *Atlassian Confluence* integriert werden.
Unterstützte Aufgaben	Architekturüberblick: Systemanalyse/-verständnis durch die Darstellung der Systemstruktur
Besonderheiten	Prozesse können per Programmcode abgebildet werden.
	Der Quellcode für die Erstellung der Diagramme muss händisch geschrieben werden.
	Dieses Tool unterstützt die Analyse von vorhandenem Quellcode nicht.

Tabelle 3.6: Analyse des Dashboard-Werkzeugs *Structurizr*

NDepend[13] ist ein statisches Analysewerkzeug für Quellcode, welcher in .NET verwaltet wird. Das Werkzeug unterstützt eine große Anzahl von Quellcodemetriken, die es ermöglichen, Abhängigkeiten mithilfe von Graphen zu visualisieren. *NDepend* kann zudem die Validierung von Architektur- und Qualitätsregeln durchführen. Die Regeln können während der Integration automatisch überprüft werden. Die Ergebnisse der zu analysierenden Kriterien für *NDepend* fasst Tabelle 3.7 zusammen.

	NDepend
Unterstützte Programmiersprachen	*C#*
Integration	*Microstoft Visual Studio*, *TeamCity*, *SonarQube*, *Jenkins Reflector*
Unterstützte Aufgaben	Qualitätsmanagement: Testen durch direkte Anzeige der Testabdeckung sowie Berichterstattung durch Dashboard-Integration in Microsoft Visual Studio Team Services
	Architekturüberblick: Validierung von Architektur- und Qualitätsregeln
Besonderheiten	Visualisierung von Quellcodemetriken
	Trend Monitoring

Tabelle 3.7: Analyse des Dashboard-Werkzeugs *NDepend*

[13] `http://www.ndepend.com`, Zugriff am: 07.05.2018

Structure101[14] ist eine Architekturentwicklungsumgebung, die es Teams ermöglicht, die Dateien ihrer Quellcodebasis in einer modularen Hierarchie mit geringer und kontrollierter Kopplung zu organisieren [Klocwork Inc. 2014]. Die Ergebnisse der zu analysierenden Kriterien für *Structure101* fasst Tabelle 3.8 zusammen.

	Structure101
Unterstützte Programmiersprachen	*Java*, *C#*
Integration	*Jenkins*, *Apache Maven*, *Eclipse*, *SonarQube*, *IntelliJ*, *Microsoft Visual Studio*
Unterstützte Aufgaben	Architekturüberblick: Systemanalyse/-verständnis durch Abhängigkeitsgraph
Besonderheiten	Darstellung von Abhängigkeiten mittels Abhängigkeitsgraph

Tabelle 3.8: Analyse des Dashboard-Werkzeugs *Structure101*

Um die Analyse existierender Dashboard-Werkzeuge abzuschließen, wurden in Tabelle 3.9 die Ergebnisse der Analyse zusammengefasst.

	Teamscale	**CodeScene**	**SonarQube**	**Structurizr**	**NDepend**	**Structure101**
Architekturüberblick	✓	✓	✗	✓	✓	✓
Ressourcenmanagement	✓	✓	✓	✗	✗	✗
Risikomanagement	✗	✓	✓	✗	✗	✗
Qualitätsmanagement	✓	✗	✓	✗	✓	✗

Tabelle 3.9: Zusammenfassung der Analyse existierender Dashboard-Werkzeuge

Im Tabellenkopf sind die Dashboard-Werkzeuge aufgelistet, wohingegen die vier Wissensbereiche in der ersten Spalte aufgelistet sind. Wird ein Wissensbereich durch Komponenten oder Visualisierungen unterstützt, ist dieser mit einem Häkchen gekennzeichnet, anderenfalls mit einem Kreuz.
Aus dieser Ergebnistabelle kann abgelesen werden, dass die vier Bereiche teilweise von den verschiedenen Werkzeugen abgedeckt werden. Es existiert allerdings noch kein Werkzeug, das alle vier Bereiche unterstützt.

[14] `https://structure101.com`, Zugriff am: 07.05.2018

Daraus kann wiederum abgeleitet werden, dass die vier Bereiche nicht nur in der Theorie, sondern auch in der Praxis relevant sind. Da es, wie erwähnt, noch keine Lösung gibt, die alle Bereiche abdeckt, soll diese Lücke durch das Dashboard geschlossen werden. Das Dashboard soll somit alle vier identifizierten Wissensbereiche unterstützen.

3.3.1.3 Use-Case-Diagramm der zu unterstützenden Aufgaben

Das Dashboard soll Projektleitern bei der Entscheidungsunterstützung helfen. Abbildung 3.2 zeigt die zu unterstützenden Aufgaben des Dashboards. Konkret soll das Dashboard Projektleiter bei der Überwachung der Architektur, der Ressourcen, des Risikos sowie der Qualität von Softwareprojekten unterstützen.

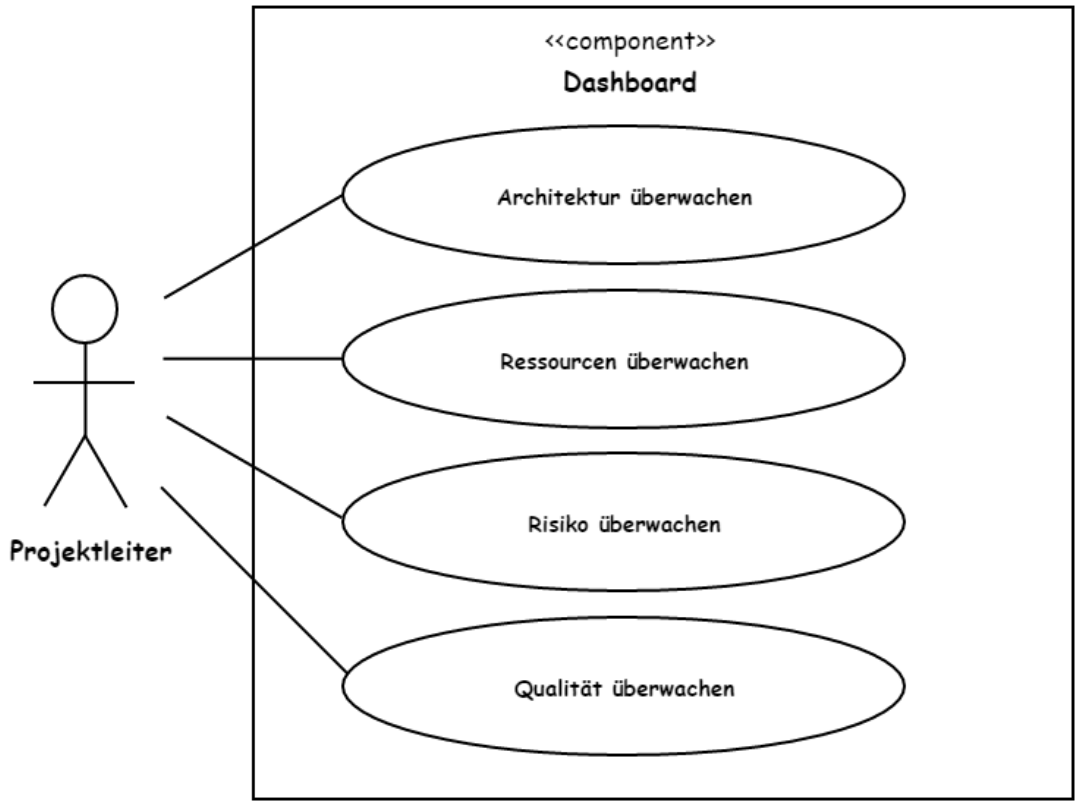

Abbildung 3.2: Use-Case-Diagramm der zu unterstützenden Aufgaben des Dashboards

3.3.1.4 Mockups der Benutzungsschnittstelle

In diesem Abschnitt werden erstellte Mockups des Dashboards thematisiert, um die Benutzungsoberfläche zu skizzieren. Auf jeder Seite des Dashboards befinden sich die Kopfzeile, die Sidebar sowie der Hauptinhaltsbereich. Die Kopfzeile enthält Links zu allgemeinen Unterseiten, wie zum Beispiel den Einstellungsmöglichkeiten des Dashboards. Die Sidebar enthält die Navigation zu allen sich im Dashboard befindenden Visualisierungen und der Hauptinhaltsbereich beinhaltet die jeweils ausgewählte Visualisierung.

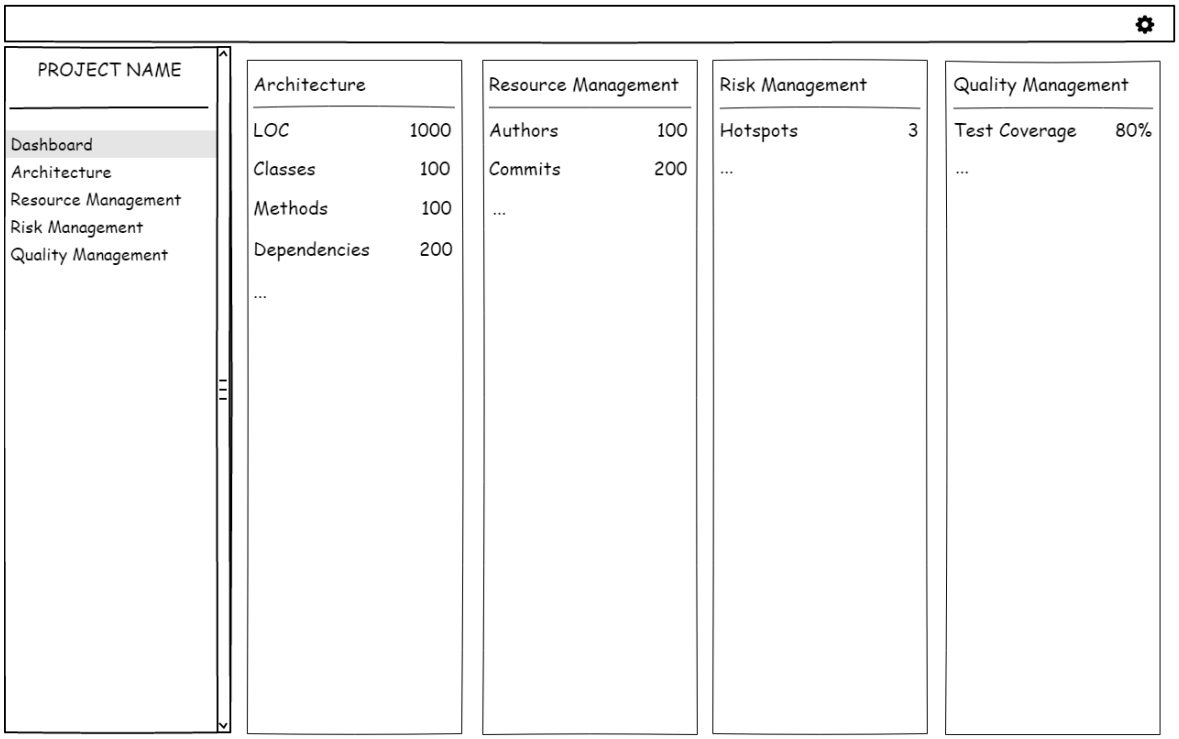

Abbildung 3.3: Mockup der Startseite des Dashboards

Abbildung 3.3 zeigt die Startseite des Dashboards. Auf dieser befindet sich eine tabellarische Übersicht aller relevanten Metriken. Beim Klick auf die jeweilige Metrik soll auf die zugehörige Visualisierung weitergeleitet werden.

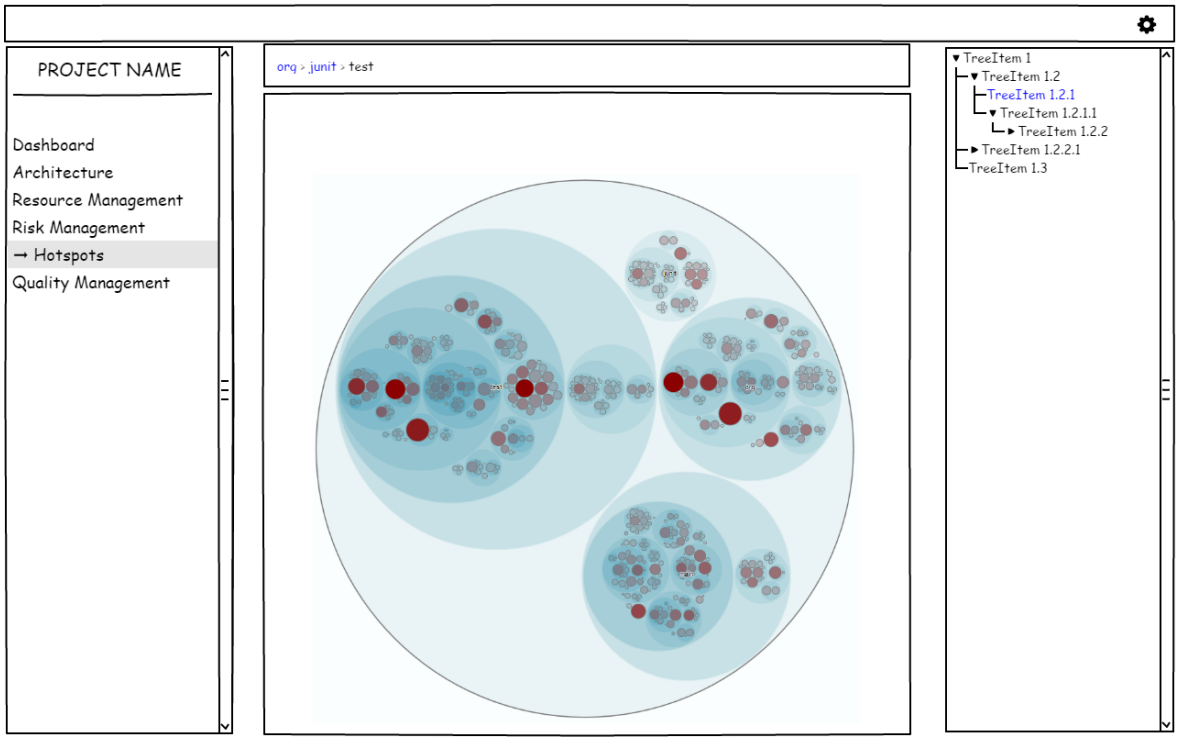

Abbildung 3.4: Mockup für die Darstellung der Hotspots innerhalb des Dashboards

Eine dieser Visualisierungen zeigt Abbildung 3.4. In diesem Beispiel werden die Hotspots, also Ressourcen beziehungsweise Pakete, die sich durch ihre Komplexität in Form einer Vielzahl von Quellcodezeilen und häufige Veränderungen auszeichnen, in der Kategorie des Risikomanagements mittels geschachtelter Kreise (engl. *circle packing*) dargestellt. Oberhalb der Visualisierung befindet sich die Anzeige der Breadcrumbs, also der jeweiligen Pfade der Ressourcen. Beim Klick auf einen Teilpfad der Breadcrumbs soll der Fokus auf das angeklickte Paket gesetzt werden und das Kreisdiagramm den entsprechenden Kreis im Browser des Nutzers zentrieren und vergrößern. Neben dem Kreisdiagramm befindet sich der Paketexplorer. Darin sollen die Ressourcen des Projekts in Form eines Baumdiagramms abgebildet werden. Beim Klick auf eine Ressource soll die Anzeige der aktuellen Breadcrumbs angepasst sowie analog zum Klick auf einen Teil der Breadcrumbs die ausgewählte Ressource in den Fokus gesetzt werden.

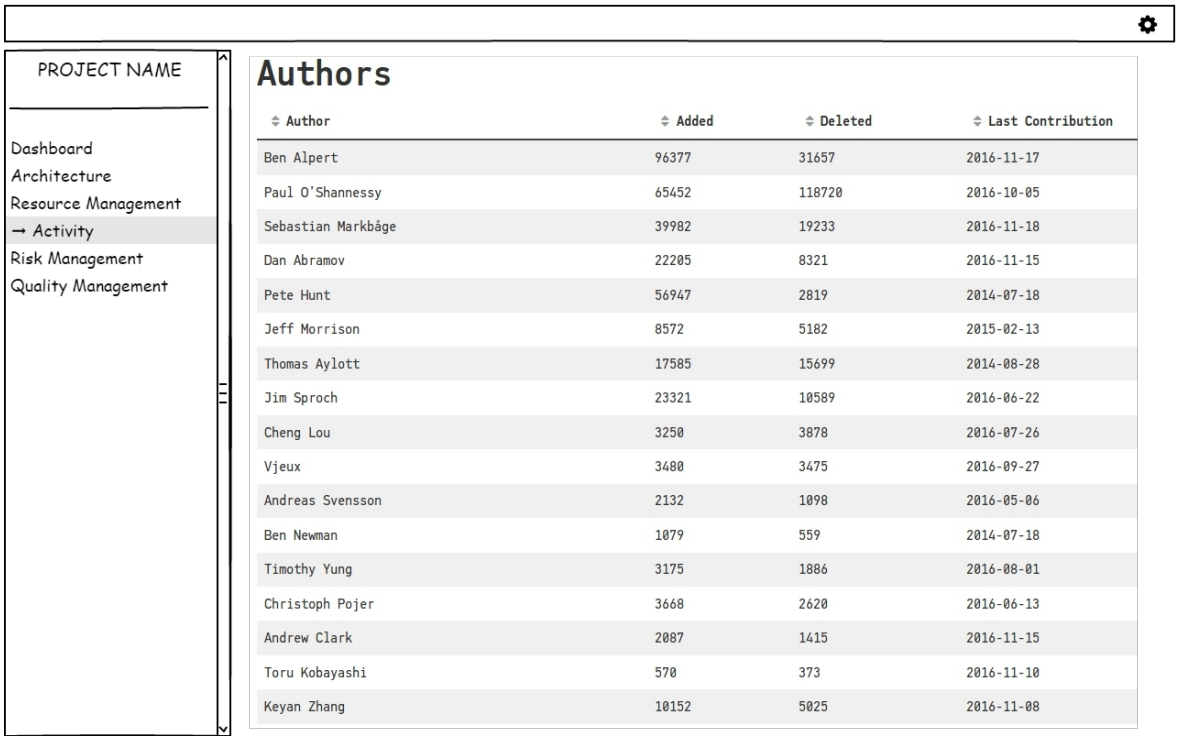

Abbildung 3.5: Mockup für die Darstellung der Autoren des Softwareprojekts innerhalb des Dashboards

In der Kategorie des Ressourcenmanagements sollen die Aktivitäten rund um das Softwareprojekt dargestellt werden. Abbildung 3.5 zeigt die am Softwareprojekt beteiligten Autoren in tabellarischer Form. Es sollen dabei die Namen der Autoren sowie die Anzahl der zu dem Softwareprojekt hinzugefügten beziehungsweise bearbeiteten Dateien sowie Informationen über die Bearbeitungszeitpunkte der einzelnen Autoren dargestellt werden.
Abschließend zeigt Abbildung 3.6 die Einstellungsmöglichkeiten des Dashboards. Zunächst sollen die Verbindungsdaten zur *Neo4j*-Graphdatenbank in Form der Uniform Resource Locator (URL), die angibt, wo sich die Datenbank befindet, sowie Benutzername und Passwort für den Zugriff auf die Datenbank angegeben werden können. Außerdem sollen projekt-

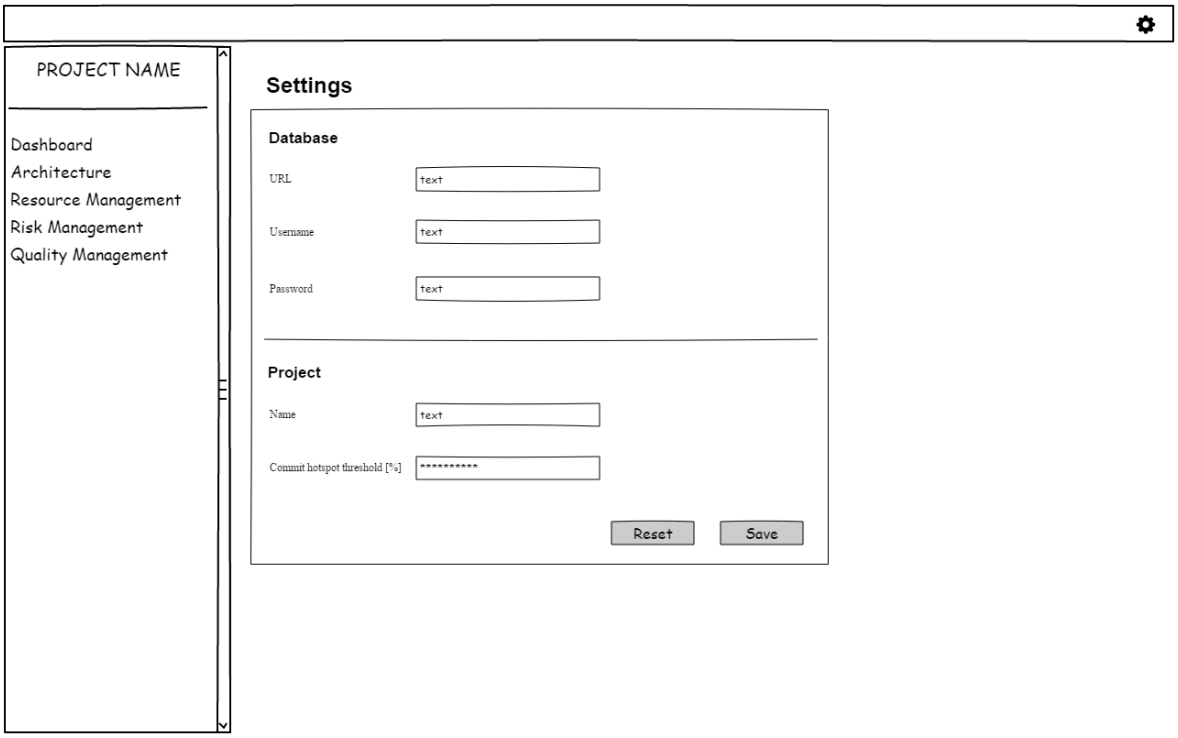

Abbildung 3.6: Mockup für die Darstellung der Einstellungsmöglichkeiten des Dashboards

spezifische Informationen wie der Projektname und der Schwellenwert für die Zählung von Commit-Hotspots, auf die in Kapitel 4 noch näher eingegangen wird, eingestellt werden können.

3.3.2 Technischer Kontext

Im Rahmen des technischen Kontexts werden nun die einzelnen Komponenten des Dashboards erläutert und in einen Zusammenhang gesetzt.
Abbildung 3.7 zeigt den Soll-Zustand der Architektur. Im Zentrum des Dashboards befindet sich die Visualisierungskomponente. Diese nutzt die *Neo4j*-Graphdatenbank, die sich wiederum auf einem Graphdatenbank-Server befindet. Die Visualisierungskomponente kommuniziert über die Schnittstelle zur Datenbank, welche die Datenbankabfragen sowie die Aufbereitung der erhaltenen Daten beinhaltet, mit der *Neo4j*-Graphdatenbank. Die aufbereiteten Abfrageergebnisse werden mithilfe von *D3.js* dargestellt. Durch diesen Aufbau soll die Komplexität der Visualisierungskomponenten verringert und die Logik von der Anzeige getrennt werden.

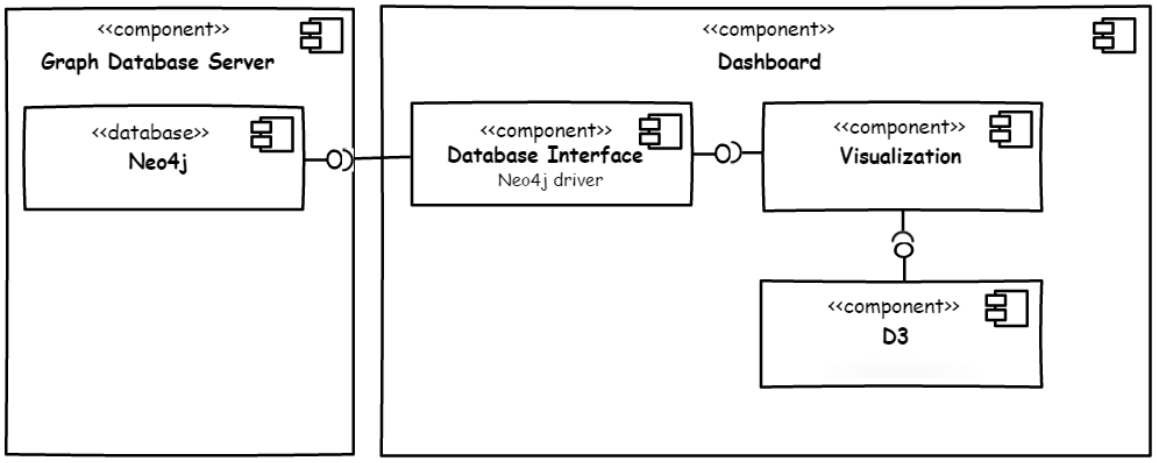

Abbildung 3.7: Komponentendiagramm des Dashboards

3.4 Randbedingungen

Dieser Abschnitt umfasst die technischen sowie die organisatorischen Randbedingungen, die im Rahmen dieser Arbeit festgelegt wurden.

3.4.1 Technische Randbedingungen

Nachfolgend sind alle technischen Randbedingungen bezüglich der Umsetzung des Dashboards aufgelistet.

- Als Datenquelle für die Visualisierungen im Dashboard dient eine durch *jQAssistant* erstellte *Neo4j*-Graphdatenbank, deren Daten durch die in Anhang A.1 zu findenden *jQAssistant*-Konzeptregeln angereichert werden.
- Die zu visualisierenden Daten werden dynamisch per *Cypher*-Abfrage aus der *Neo4j*-Graphdatenbank geholt.
- Die Visualisierungskomponenten werden mit *D3.js* realisiert.
- Für die Versionsverwaltung wird ein öffentliches *Git-Repository* bei *GitHub* genutzt.
- Für die Erstellung der Mockups wird *Pencil*[15] eingesetzt.
- Das Dashboard sollte keine eigenen Daten in der *Neo4j*-Graphdatenbank speichern.
- Für die Umsetzung des Dashboards sollte eine *Node.js*-unterstützende integrierte Entwicklungsumgebung eingesetzt werden.

[15] `https://pencil.evolus.vn/`, Zugriff am: 27.05.2018

3.4.2 Organisatorische Randbedingungen

Die für diese Arbeit festgelegten organisatorischen Randbedingungen können der nachfolgenden Auflistung entnommen werden.

- Die Abfragen sowie die jeweilige Art der Visualisierung werden von dem Betreuer bereitgestellt.
- Die während dieser Arbeit zu verwendenden Sprachen wurden wie folgt festgelegt.
  - Masterarbeit sowie daraus resultierende *arc42*-Dokumente: Deutsch
  - Quellcode, Dokumentation des Quellcodes, Commit-Nachrichten: Englisch

3.5 Risiken und technische Schulden

Als Risiken für einen eventuellen Projektmisserfolg beziehungsweise eventuelle zukünftige Probleme im Zusammenhang mit der Implementierung des Dashboards sind die externen Abhängigkeiten der einzelnen *Node.js*-Module zu nennen. Diese Risiken liegen allerdings in der Natur von *Node.js*, da die Nutzung bereits vorhandener Pakete zur Reduzierung des Aufwands bei der Umsetzung von Projekten ein zentraler Bestandteil von *Node.js* ist.
Als technische Schuld ist zunächst die unvollständige Testabdeckung des Dashboards zu erwähnen. Durch die fehlende Möglichkeit der Vortäuschung oder Simulation einer *Neo4j*-Graphdatenbank während der Ausführung der einzelnen Testfälle können die Teile des Quellcodes, in denen sich beispielsweise die *Cypher*-Abfragen befinden, sowie die direkt anschließende Datenaufbereitung nicht automatisch getestet werden. Des Weiteren konnten auf Zeit basierende beziehungsweise zeitabhängige Visualisierungen aufgrund der dazu fehlenden Datenbasis nicht vollständig umgesetzt werden. Jedoch werden in Abschnitt 4.2.4 erste Ansätze dazu erläutert und umgesetzt.

3.6 Entwurfsentscheidungen

In diesem Abschnitt erfolgt die Auswahl des Treibers zur Anbindung der *Neo4j*-Graphdatenbank. Anschließend werden die fünf populärsten existierenden *Webframeworks* miteinander verglichen, um die geeignetste Grundlage zur Implementierung des Dashboards zu finden.

3.6.1 Auswahl des Neo4j-Treibers

Um die Verbindung des Dashboards zu einer *Neo4j*-Graphdatenbank herzustellen, wird ein Treiber in Form einer *JavaScript*-Bibliothek benötigt, der die Abfrage der *Neo4j*-Graphdatenbank in der Abfragesprache *Cypher* mittels *JavaScript* ermöglicht. Da mehrere solcher *JavaScript*-Bibliotheken zur Anbindung von *Neo4j* existieren, muss zunächst einer dieser

Treiber ausgewählt werden, welcher in dem Dashboard verwendet wird. Seitens der offiziellen Webseite von *Neo4j* wird eine Auflistung der verschiedenen Treiber angeboten[16]. So gibt es neben dem offiziell von *Neo4j* unterstützten Treiber *Neo4j Javascript Driver* noch weitere Anbindungsmöglichkeiten, welche von der *Neo4j*-Nutzergemeinde bereitgestellt wurden. Namentlich handelt es sich dabei um *node-neo4j*, *cypher-stream*, *Seraph*, *simple-neo4j*, *Meteor.js – Package ccorcos:neo4j*, *Ostrio – Meteor.js – Driver*, *Neo4j Dart* sowie *node-neo4j*. Das Dashboard soll jedoch den offiziell von *Neo4j* unterstützten Treiber *Neo4j Javascript Driver* verwenden. Dies hat den Vorteil, dass die Verwendung dieses Treibers keine zusätzlichen Abhängigkeiten der Software Dritter nach sich zieht, was für das Qualitätsziel der Wartbarkeit spricht. Außerdem begünstigt die offizielle Unterstützung von *Neo4j* ebenfalls die Wartbarkeit des Dashboards, da die Wartung des Treibers direkt durch *Neo4j* erfolgt.
Der *Neo4j Javascript Driver* wird unter den Lizenzbedingungen der *Apache License 2.0* vertrieben und kann damit problemlos für die prototypische Implementierung des Dashboards verwendet werden.

3.6.2 Vergleich existierender Webframeworks

Um das am besten geeignete web-basierte Framework als Grundlage für die Umsetzung des Dashboards zu finden, werden nun die fünf populärsten web-basierten Frameworks näher betrachtet. Zur Bestimmung dieser wurde die Webseite *https://bestof.js.org* nach den populärsten web-basierten Frameworks zur Erstellung von Benutzungsoberflächen gefiltert[17]. Das Resultat dieser Filterung zeigt Abbildung 3.8.
Bei allen zu untersuchenden Frameworks handelt es sich um Open-Source-Software. Es wird jeweils die aktuelle Version der Frameworks betrachtet. Somit wird *Angular* beispielsweise nur in Version 6.1.2 untersucht, obwohl in Abbildung 3.8 auch *Angular 1* als eines der populärsten UI-Frameworks genannt wird.
Prinzipiell ist die Umsetzung des Dashboards mit jedem der untersuchten Frameworks möglich, jedoch unterscheidet sich die Dauer der Umsetzung anhand der bereits vorhandenen Möglichkeiten des jeweiligen Frameworks, wie beispielsweise die Wiederverwendung von Komponenten. Da die Komponenten des Dashboards mithilfe von *D3.js* realisiert werden sollen, stellt die eventuell bereits vorhandene Unterstützung von Komponenten in Verbindung mit *D3.js* ebenfalls ein Auswahlkriterium dar.

3.6.2.1 Angular

Das erste betrachtete web-basierte Framework ist *Angular*[18]. Es wird von *Google* sowie einer Entwicklergemeinde, bestehend aus weiteren Unternehmen und Einzelpersonen, entwickelt und als Open-Source-Software publiziert. Mit *Angular* können sogenannte Single-Page-Webanwendungen umgesetzt werden, bei denen lediglich ein einziges *HTML*-Doku-

[16] `https://neo4j.com/developer/javascript/`, Zugriff am: 03.08.2018
[17] `https://bestof.js.org/tags/framework`, Zugriff am: 01.12.2017
[18] `https://angular.io/`, Zugriff am: 05.08.2018

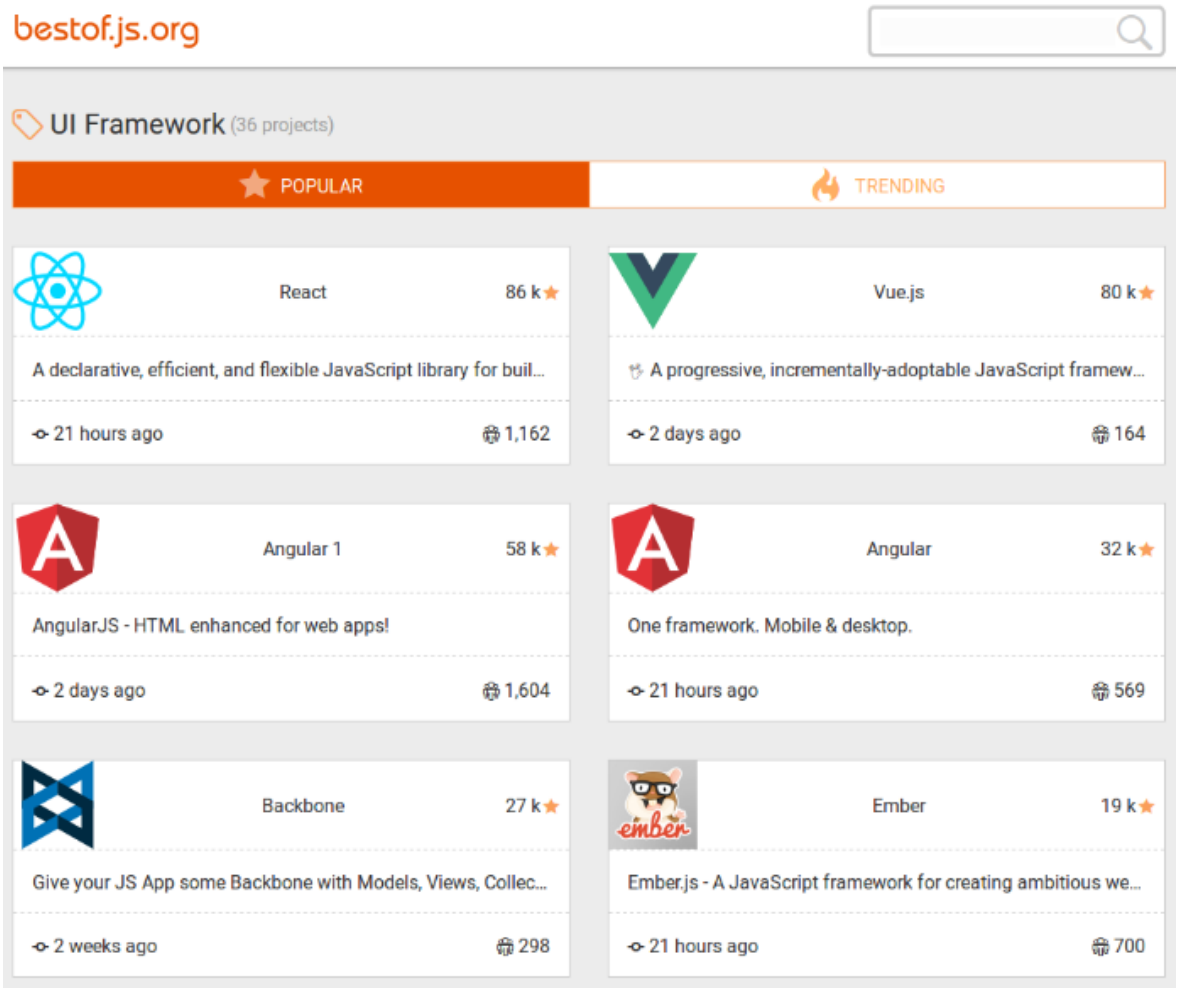

Abbildung 3.8: Die fünf populärsten web-basierten Frameworks zum Stand der Erstellung der Arbeit

ment zum Einsatz kommt und neue oder veränderte Daten dynamisch nachgeladen werden. Bei der Verwendung von *Angular* können die Entwicklungsmuster *Model View Controller (MVC)* und *Model View ViewModel (MVVM)* verfolgt werden[19].
Angular wird unter anderem von den Unternehmen *VMWare*, *Nokia Health*, *Bose* sowie *NBA* produktiv eingesetzt[20].
Zwar sind bereits erste Ansätze zur Verwendung wiederverwendbarer *D3.js*-Komponenten in Verbindung mit *Angular* vorhanden[21], diese sind jedoch leider eher ein konzeptioneller Beweis als eine in der Praxis verwendbare Grundlage. Zudem ist *Angular* im Vergleich zu den anderen betrachteten web-basierten Frameworks sehr komplex und schwergewichtig, weshalb *Angular* bei der Auswahl des Frameworks für das Dashboard ausscheidet.

[19] `https://namitamalik.github.io/MVC-and-MVVM-with-AngularJS/`, Zugriff am: 05.08.2018
[20] `https://www.madewithangular.com/`, Zugriff am: 07.08.2018
[21] `http://busypeoples.github.io/post/reusable-chart-components-with-angular-d3-js/`, Zugriff am: 05.08.2018

3.6.2.2 Backbone.js

Das nächste betrachtete Framework ist *Backbone.js*[22]. Dabei handelt es sich um eine *JavaScript*-Bibliothek mit integrierter Schnittstelle zu dem kompakten Datenaustauschformat *JavaScript Object Notation (JSON)*. *Backbone.js* basiert auf dem *MVC*-Entwicklungsmuster und ist zudem für seine geringe Größe bekannt. *Backbone.js* wird ebenfalls zur Programmierung von Single-Page-Webanwendungen verwendet.
Backbone.js wird unter anderem von den Unternehmen *Airbnb*, *Pinterest*, *Wordpress* sowie *Bitbucket* produktiv eingesetzt.
Analog zu *Angular* existieren für *Backbone.js* bereits erste Ansätze zur Verwendung wiederverwendbarer *D3.js*-Komponenten[23], diese sind jedoch nicht im produktiven Einsatz verwendbar und müssten zunächst vollständig implementiert werden. Daher wird *Backbone.js* ebenfalls nicht für die Implementierung des Dashboards verwendet.

3.6.2.3 Ember.js

Wie die bereits betrachteten web-basierten Frameworks ist auch *Ember.js*[24] ein *JavaScript*-Framework zur Erstellung von Single-Page-Webanwendungen. *Ember.js* basiert auf dem *MVVM*-Entwicklungsmuster und folgt den Prinzipien *Don't repeat yourself* und *Konvention vor Konfiguration*.
Ember.js wird unter anderem von den Unternehmen *Microsoft*, *Netflix*, *NASA*, *Kickstarter* sowie *LinkedIn* produktiv eingesetzt.
Leider existieren derzeit noch keine Komponenten mit Blick auf *D3.js*. Da für die Umsetzung des Dashboards mittels *Ember.js* demnach zunächst die Anbindung an *D3.js* realisiert werden müsste, scheidet *Ember.js* ebenfalls aus.

3.6.2.4 Vue.js

Auch bei *Vue.js* handelt es sich um ein web-basiertes Framework zur Umsetzung von Single-Page-Webanwendungen, wobei allerdings auch Multipage-Webseiten für einzelne Abschnitte verwendet werden können. *Vue.js* ist ein progressives Framework zur Erstellung von Benutzungsoberflächen. Im Vergleich zu anderen web-basierten Frameworks ist *Vue.js* inkrementell adaptierbar, was eine erhöhte Flexibilität bei der Strukturierung der Anwendung ermöglicht. Wie *Ember.js* basiert auch *Vue.js* auf dem *MVVM*-Entwicklungsmuster[25].
Vue.js wird unter anderem von den Unternehmen *Font Awesome*, *Codeship* sowie *Cachet* produktiv eingesetzt[26].

---

22 `https://backbonejs.org/`, Zugriff am: 05.08.2018
23 `https://medium.com/@sxywu/marrying-backbone-js-and-d3-js-a-follow-up-b6a62a9731e1`, Zugriff am: 07.08.2018
24 `https://guides.emberjs.com/release/`, Zugriff am: 05.08.2018
25 `https://vuejs.org/v2/guide/`, Zugriff am: 05.08.2018
26 `https://madewithvuejs.com/`, Zugriff am: 07.08.2018

Im Vergleich zu den bisher betrachteten web-basierten Frameworks ist *Vue.js* noch sehr jung. Es existieren zwar bereits Ansätze zu der Anbindung von *D3.js*, allerdings wurden diese bisher nur sehr rudimentär zu verwendbaren Komponenten umgesetzt[27]. Durch diese derzeit fehlende Praxistauglichkeit wird *Vue.js* ebenfalls nicht zur Umsetzung des Dashboards verwendet.

3.6.2.5 React

React[28] ist eine deklarative, effiziente und flexible *JavaScript*-Bibliothek zur Erstellung von Benutzungsoberflächen. Diese Benutzungsoberflächen bestehen aus kleinen und isolierten Programmcodeteilen und werden als Komponenten bezeichnet.
Anders als bei den bisher betrachteten web-basierten Frameworks, welche entweder auf dem *MVC*- oder dem *MVVM*-Entwicklungsmuster basieren, bietet das Unternehmen *Facebook* für *React* eine eigene Architektur mit dem Namen *Flux*.
Da *React*-Komponenten lediglich den Teil des *Views* im klassischen *MVC*-Entwicklungsmuster abbilden, stellt sich die Frage nach der Verwaltung der restlichen Bestandteile für die Auslieferung dynamischer Inhalte. Den Aufbau der *Flux*-Architektur verdeutlicht Abbildung 3.9.

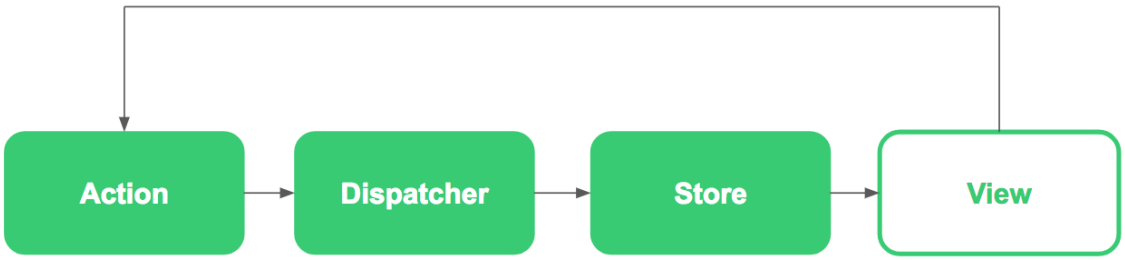

Abbildung 3.9: Die *Flux*-Architektur [Deutsch 2015]

Es existiert lediglich eine Datenquelle, wobei die Veränderungen dieser Datenquelle niemals direkt geschehen, sondern nur über sogenannte *Actions*.
Actions sind *JavaScript*-Objekte, welche aus den *Views* beziehungsweise den konkreten *React*-Komponenten aufgerufen werden. Durch die Angabe des *type*-Attributs können *Stores* erkennen, ob die *Action* für sie relevant ist.
Das *Action-JavaScript*-Objekt wird dann an den *Dispatcher* weitergeleitet, welcher es wiederum an die *Stores* verteilt und bei Bedarf die *React*-Komponente neu rendert.
In den *Stores* wird die Applikationslogik verwaltet. Sie werden als Datenquellen für *React*-Komponenten genutzt. Die *Stores* reagieren auf *Actions* aus dem *Dispatcher*. Wird eine *Action* erkannt, kann diese den *Store* verändern. Ein geänderter *Store* benachrichtigt daraufhin automatisch seine *React*-Komponenten und die Änderung wird anschließend für den Nutzer sichtbar.

[27] `https://corpglory.github.io/d3vue/`, Zugriff am: 08.08.2018
[28] `https://reactjs.org/`, Zugriff am: 05.08.2018

Die Anwendung der *Flux*-Architektur ermöglicht somit eine sehr gute Strukturierung des Quellcodes, insbesondere durch eine klare Modularisierung der Anwendung.
React wird unter anderem von den Unternehmen *Facebook*, *Instagram*, *ARTE*, *Atlassian* sowie *Twitter Mobile* produktiv eingesetzt[29].
Für *React* existieren bereits einige Visualisierungskomponenten in Verbindung mit *D3.js*, so zum Beispiel das Softwareprojekt *Nivo*[30]. *Nivo* bietet eine reichhaltige Sammlung von Datenvisualisierungskomponenten, welche auf *D3.js* und *React* aufbaut.
Durch die einfache Erstellung von wiederverwendbaren Komponenten wird zudem das Qualitätsziel der Erweiterbarkeit unterstützt. Daher fällt die Auswahl des web-basierten Frameworks zur Umsetzung des Dashboards auf *React*.

3.7 Lösungsstrategie

Die folgende Tabelle 3.10 stellt die Qualitätsziele des Dashboards aus Abschnitt 3.2 passenden Architekturansätzen gegenüber und erleichtert so einen Einstieg in die Lösung.

Qualitätsziel	**Umsetzung**
Erweiterbarkeit	*React* *arc42* *Travis CI* *Codecov*
Installierbarkeit	*Node.js* *Docker*
Gebrauchstauglichkeit	Literaturrecherche zu typischen Aufgaben von Projektleitern Analyse existierender Dashboard-Werkzeuge *Bootstrap*
Wartbarkeit	*Node.js* *arc42* *Codecov* *Prettier*

Tabelle 3.10: Gegenüberstellung von Qualitätszielen und passenden Architekturansätzen

[29] `https://github.com/facebook/react/wiki/Sites-Using-React`, Zugriff am: 07.08.2018
[30] `http://nivo.rocks/components`, Zugriff am: 08.08.2018

4 Implementierung

In diesem Kapitel wird die Implementierung eines Prototyps für das in dem vorangegangenen Kapitel konzipierte Dashboard beschrieben. Dazu werden zunächst geeignete Hilfsmittel, die den Implementierungsaufwand minimieren, betrachtet. Danach werden die einzelnen Bestandteile des Dashboards erarbeitet, bis jeder der identifizierten Wissensbereiche von dem Dashboard unterstützt wird. Anschließend wird die Erstellung neuer Visualisierungskomponenten an einem Beispiel näher beschrieben. Im darauffolgenden Abschnitt wird die Möglichkeit des Testens der Funktionalitäten des Dashboards thematisiert. Die für das Dashboard erstellten Tests werden im anschließenden Abschnitt weiter betrachtet, indem die Testabdeckung des Dashboard-Quellcodes gemessen und analysiert wird. Im letzten Abschnitt dieses Kapitels werden die Installier- und Wartbarkeit des Dashboards erhöht.

4.1 Implementierungskomponenten

Nachfolgend werden die Implementierungskomponenten des Dashboards beschrieben. Dabei handelt es sich um *CoreUI* und *Nivo*, welche die Grundsteine für die Implementierung des Dashboards bilden.

4.1.1 CoreUI

Da es sich bei *React* lediglich um ein Framework handelt, müsste bei alleiniger Verwendung von *React* zur Umsetzung des Dashboards die komplette Benutzungsschnittstelle selbst erstellt werden. Um diesem Mehraufwand entgegenzuwirken, wird *CoreUI*[31] für die Implementierung des Dashboards verwendet. *CoreUI* ist eine Open-Source-Software für die Erstellung von Benutzungsschnittstellen und basiert auf dem freien *CSS*-Framework *Bootstrap*. *CoreUI* dient somit als Vorlage (engl. *template*) für die Erstellung des Dashboards, was die Qualitätsziele der Gebrauchstauglichkeit sowie der Wartbarkeit unterstützt. Es existiert eine speziell für *React* erstellte Version von *CoreUI*, welche bereits fertige *React*-Komponenten zur Verfügung stellt. Mithilfe dieser *React*-Komponenten wird die in Abschnitt 3.3.1.4 beschriebene strukturelle Umsetzung von Kopfzeile, Sidebar sowie Hauptinhaltsbereich realisiert. Auch die Erstellung von Formularen und Aufzählungen für die Umsetzung der Startseite sowie der Einstellungsmöglichkeiten des Dashboards können mit den bereitgestellten Komponenten umgesetzt werden.

[31] `https://coreui.io/`, Zugriff am: 20.08.2018

Das Dashboard wurde zunächst in Version 1.0.6[32] von *CoreUI* umgesetzt. Im Verlauf der Anfertigung dieser Arbeit wurde Version 2.0.5[33] veröffentlicht.
Da Version 2.0.5 neue Features ermöglicht und die Qualitätsziele der Wartbarkeit sowie der Erweiterbarkeit unterstützt, wurde das Update des Dashboards im Rahmen der Arbeit umgesetzt.
Die neue Version bietet einige nutzbringende Veränderungen. So wird für *CoreUI* und damit auch für das Dashboard *create-react-app* verwendet. Damit werden Fehler und Warnungen nun direkt in der Konsole angezeigt, wenn das Dashboard ausgeführt wird. Die Qualität dieser Hinweise übertrifft die standardmäßige Ausgabe in den Entwicklertools deutlich. Es werden beispielsweise Hinweise angezeigt, wenn Quellcode eingebunden beziehungsweise definiert ist, aber nie verwendet wird. Mithilfe dieser Hinweise wurden alle Dateien des Dashboards nochmals refaktorisiert, um die Qualität des Quellcodes zu verbessern.
Im Rahmen dieser Refaktorisierung wurde auch der *neo4j-driver* so angepasst, dass statt der statischen *JavaScript*-Datei nun das entsprechende Paket des *Node Package Manager (NPM)* verwendet wird und der Treiber somit aktualisierbar ist. Außerdem wurden nicht mehr verwendete Packages in der *package.json* entfernt.
Die vorhandenen Tests des Dashboards konnten vollständig migriert werden. Zur Ausführung kann weiterhin `npm run test-dashboard` verwendet werden.

4.1.2 Nivo

Nachdem die Umsetzung der allgemeinen Struktur des Dashboards gewährleistet ist, wird in diesem Abschnitt die Erstellung der Visualisierungen im Hauptinhaltsbereich thematisiert. Bei der ausschließlichen Verwendung der derzeitigen Werkzeuge müssten alle Visualisierungen des Dashboards manuell mittels *D3.js* erstellt werden, was bereits bei der Erstellung eines Balkendiagramms eines hohen Aufwands bedarf[34]. Um diesem Mehraufwand entgegenzuwirken, wird *Nivo* für die Implementierung des Dashboards verwendet. *Nivo*[35] ist eine Zusammenfassung von *React*-Komponenten für die einfache Erstellung von Datenvisualisierungen, die auf *D3.js* aufbauen und als Open-Source-Software zur freien Verfügung stehen. Mithilfe von *Nivo* können neben vollständig deklarativen Balken-, Kreis- und Liniendiagrammen zahlreiche weitere Visualisierungen wie die Darstellung geschachtelter Kreise oder Kalender erstellt werden. Die Komponenten bieten dabei Interaktionsmöglichkeiten bei Klick oder dem Überfahren einzelner Datensätze mit dem Mauszeiger. Bei der Interaktion werden spezifische Informationen des ausgewählten Datensatzes mithilfe von Tooltips, also kleinen Pop-up-Fenstern, dargestellt. Leider gab es noch keine Möglichkeit der benutzerdefinierten Ausgabe von Tooltips. Daher wurde diese Möglichkeit zunächst lokal in *Nivo* ein-

[32] `https://github.com/coreui/coreui-free-react-admin-template/releases/tag/v1.0.6`, Zugriff am: 21.08.2018
[33] `https://github.com/coreui/coreui-free-react-admin-template/releases/tag/v2.0.5`, Zugriff am: 21.08.2018
[34] `https://bl.ocks.org/mbostock/3885304`, Zugriff am: 20.08.2018
[35] `http://nivo.rocks/components`, Zugriff am: 08.08.2018

gebaut und nach erfolgreichem Test mittels eines *Git-Pull-Request* auf *GitHub* eingereicht[36]. Dieser *Git-Pull-Request* wurde bereits erfolgreich in *Nivo* aufgenommen, sodass die Verwendung von benutzerdefinierten Tooltips nun möglich ist. Damit kann *Nivo* vollständig für die Erstellung der Visualisierungskomponenten des Dashboards verwendet werden.

4.2 Dashboard

In diesem Abschnitt werden die Bestandteile des Dashboards erarbeitet. Zunächst werden die einzelnen Komponenten der Startseite beschrieben. Danach wird näher auf die Einstellungsmöglichkeiten des Dashboards eingegangen. Zudem soll es möglich sein, benutzerdefinierte Anfragen an die *Neo4j*-Graphdatenbank zu stellen, was im anschließenden Abschnitt behandelt wird. Abschließend wird auf die konkreten Visualisierungskomponenten des Dashboards inklusive ihrer jeweiligen *Cypher*-Abfragen eingegangen.

4.2.1 Startseite

Um die bestmögliche Übersicht zu gewährleisten, sollen auf der Startseite des Dashboards alle relevanten Kennzahlen nach Wissensbereichen gruppiert dargestellt werden. Zusätzlich sollen pro Wissensbereich die konkreten Kennzahlen in einzelne Metriken zusammengefasst werden. Im Wissensbereich des Architekturüberblicks können zunächst alle Kennzahlen, die die Struktur des Softwareprojekts beschreiben, zusammengefasst werden. Diese Strukturmetrik enthält die Anzahl der Klassen (engl. *classes*), Schnittstellen (engl. *interfaces*), Aufzählungstypen (engl. *enums*), Annotationen (engl. *annotations*), Methoden (engl. *methods*), Quellcodezeilen (engl. *Lines of Code*) sowie Felder (engl. *fields*). Um diese Kennzahlen für die Ausgabe der Strukturmetrik zu erhalten, wurde die in Listing 4.1 dargestellte *Cypher*-Abfrage formuliert.

```
1  OPTIONAL MATCH (t:Type:Class)-[:HAS_SOURCE]->(:File)
2  WITH count(t) as classes
3  // number of interfaces
4  OPTIONAL MATCH (t:Type:Interface)-[:HAS_SOURCE]->(:File)
5  WITH classes, count(t) as interfaces
6  // number of enums
7  OPTIONAL MATCH (t:Type:Enum)-[:HAS_SOURCE]->(:File)
8  WITH classes, interfaces, count(t) as enums
9  // number of annotations
10 OPTIONAL MATCH (t:Type:Enum)-[:HAS_SOURCE]->(:File)
11 WITH classes, interfaces, enums, count(t) as annotations
12 // number of methods and lines of code
13 OPTIONAL MATCH (t:Type)-[:HAS_SOURCE]->(:File), (t)-[:DECLARES]->(m:Method)
14 WITH classes, interfaces, enums, annotations, count(m) as methods,
15 sum(m.effectiveLineCount) as loc
16 // number of fields
17 OPTIONAL MATCH (t:Type)-[:HAS_SOURCE]->(:File), (t)-[:DECLARES]->(f:Field)
18 RETURN classes, interfaces, enums, annotations, methods, loc, count(f) as fields
```

Listing 4.1: *Cypher*-Abfrage für die Ausgabe der Strukturmetrik

[36] https://github.com/tmewes/nivo/commit/a00d96dfe673361875c78bf1f43f703dc4ae1c27, Zugriff am: 20.08.2018

Der Architekturüberblick kann außerdem noch um Kennzahlen bezüglich der Anzahl von Abhängigkeiten innerhalb des Softwareprojekts erweitert werden. Darunter fallen die Anzahl der Abhängigkeiten (engl. *dependencies*) selbst, aber auch die Anzahl der Vererbungen (engl. *extends*), Implementierungen (engl. *implements*), Methodenaufrufe (engl. *invocations*), Lese- (engl. *reads*) sowie Schreibvorgänge (engl. *writes*). Um diese Kennzahlen für die Ausgabe der Abhängigkeitsmetrik zu erhalten, wurde die in Listing 4.2 dargestellte *Cypher*-Abfrage formuliert.

```
1  OPTIONAL MATCH (t:Type)-[:HAS_SOURCE]->(:File), (t)-[d:DEPENDS_ON]->(:Type)
2  WITH count(d) as dependencies
3  // extends
4  OPTIONAL MATCH (t:Type)-[:HAS_SOURCE]->(:File), (t)-[e:EXTENDS]->(superType:Type)
5  WHERE superType.name <> "Object"
6  WITH dependencies, count(e) as extends
7  // implements
8  OPTIONAL MATCH (t:Type)-[:HAS_SOURCE]->(:File), (t)-[i:IMPLEMENTS]->(:Type)
9  WITH dependencies, extends, count(i) as implements
10 // calls
11 OPTIONAL MATCH (t:Type)-[:HAS_SOURCE]->(:File), (t)-[:DECLARES]->(m:Method)-[i:INVOKES]->(:Method)
12 WITH dependencies, extends, implements, count(i) as invocations
13 // reads
14 OPTIONAL MATCH (t:Type)-[:HAS_SOURCE]->(:File), (t)-[:DECLARES]->(m:Method)-[r:READS]->(:Field)
15 WITH dependencies, extends, implements, invocations, count(r) as reads
16 // writes
17 OPTIONAL MATCH (t:Type)-[:HAS_SOURCE]->(:File), (t)-[:DECLARES]->(m:Method)-[w:WRITES]->(:Field)
18 RETURN dependencies, extends, implements, invocations, reads, count(w) as writes
```

Listing 4.2: *Cypher*-Abfrage für die Ausgabe der Abhängigkeitsmetrik

Im Wissensbereich des Ressourcenmanagements werden alle Kennzahlen, die sich auf Aktivitäten beziehen, zusammengefasst. Diese Aktivitätsmetrik enthält die Anzahl der Autoren, die an dem Softwareprojekt beteiligt sind, die Anzahl der Commits ex- sowie inklusive Zusammenführungen (engl. *merges*). Um diese Kennzahlen für die Ausgabe der Aktivitätsmetrik zu erhalten, wurde die in Listing 4.3 dargestellte *Cypher*-Abfrage formuliert.

```
1  OPTIONAL MATCH (a:Author)
2  WITH count(a) as authors
3  // number of commits (without merges)
4  OPTIONAL MATCH (c:Commit)-[:CONTAINS_CHANGE]->()-[:MODIFIES]->(f:File)
5  WHERE NOT c:Merge
6  WITH authors, count(c) as commitsWithoutMerges
7  // number of commits (including merges)
8  OPTIONAL MATCH (c:Commit)-[:CONTAINS_CHANGE]->()-[:MODIFIES]->(f:File)
9  RETURN authors, commitsWithoutMerges, count(c) as commitsWithMerges
```

Listing 4.3: *Cypher*-Abfrage für die Ausgabe der Aktivitätsmetrik

Der nächste betrachtete Wissensbereich ist das Risikomanagement. Darin wird zunächst die Anzahl der Commit-Hotspots der Hotspotmetrik zugeordnet. Um diese Kennzahlen zu erhalten, wurde die in Listing 4.4 dargestellte *Cypher*-Abfrage formuliert.

```
MATCH (c:Commit)-[:CONTAINS_CHANGE]->()-[:MODIFIES]->(f:File)
WHERE NOT c:Merge WITH f, count(c) as commits
MATCH (t:Type)-[:HAS_SOURCE]->(f), (t)-[:DECLARES]->(m:Method)
RETURN t.fqn as fqn, sum(commits) as commits
ORDER BY fqn ASCENDING
```

Listing 4.4: *Cypher*-Abfrage für die Ausgabe der Hotspotmetrik

Der letzte identifizierte Wissensbereich ist das Qualitätsmanagement. Diesem werden die Metriken der statischen Quellcodeanalyse sowie der Testabdeckung zugeordnet. Die statische Quellcodeanalyse umfasst die Kennzahl der Verstöße gegen festgelegte Richtlinien, auf die im Abschnitt 4.2.4.7 näher eingegangen wird. Um diese Kennzahlen zu erhalten, wurde die in Listing 4.5 dargestellte *Cypher*-Abfrage formuliert.

```
MATCH (:Report)-[:HAS_FILE]->(file:File:Pmd)-[:HAS_VIOLATION]->(violation:Violation)
RETURN count(violation)
```

Listing 4.5: *Cypher*-Abfrage für die Ausgabe der statischen Quellcodeanalyse

Die Metrik der Testabdeckung enthält die Kennzahl der prozentualen Gesamttestabdeckung des Softwareprojekts, welche mithilfe der in Listing 4.6 abgebildeten *Cypher*-Abfrage erhalten werden kann.

```
MATCH (c:Jacoco:Class)-[:HAS_METHOD]->(m:Method:Jacoco)-[:HAS_COUNTER]->(t:Counter)
WHERE t.type='INSTRUCTION'
RETURN (sum(t.covered)*100)/(sum(t.covered)+sum(t.missed)) as coverage
```

Listing 4.6: *Cypher*-Abfrage für die Ausgabe der Testabdeckung

Um die Darstellung der Wissensbereiche zu trennen, wurden vier sogenannte *Cards* nebeneinander angelegt. Bei *Cards* handelt es sich um vorgefertigte *CoreUI*-Elemente, die mit Inhalt befüllt werden können. Die Ergebnisse der genannten Abfragen werden innerhalb von gruppierten Listen dargestellt. Außerdem wird an jeder *Card* noch ein Hilfesymbol in der Titelzeile ergänzt. Beim Klick auf dieses Symbol öffnet sich ein Pop-up mit Beschreibungstexten der jeweiligen *Card*. Abbildung 4.1 zeigt die fertig implementierte Startseite des Dashboards.

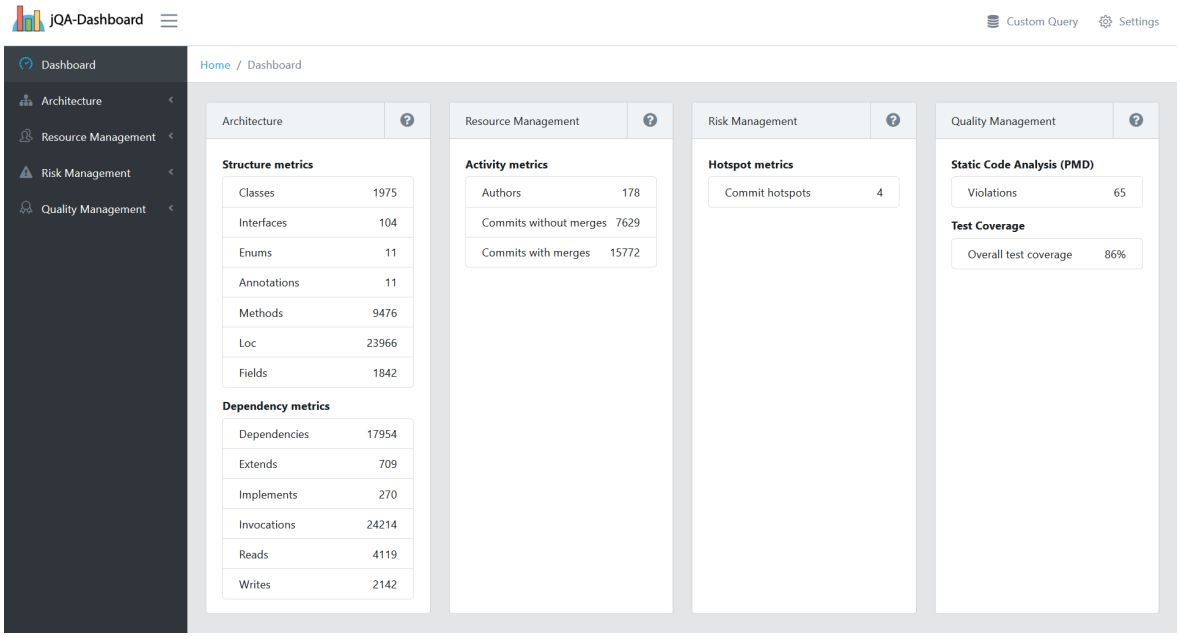

Abbildung 4.1: Startseite des Dashboards

4.2.2 Einstellungen

In diesem Abschnitt werden die verschiedenen Einstellungsmöglichkeiten des Dashboards erarbeitet. Zunächst soll der Benutzer des Dashboards Einstellungen bezüglich der Verbindung zu der *Neo4j*-Graphdatenbank vornehmen können. Die erste dazugehörige Einstellungsmöglichkeit ist die URL zu der *Neo4j*-Graphdatenbank. Dabei wird das eigens von den Entwicklern von *Neo4j* entwickelte Protokoll *Bolt* benutzt, um die Verbindung herzustellen. Außerdem wird die Angabe eines Benutzernamens sowie eines Passworts benötigt, welches zuvor in der Benutzungsoberfläche der *Neo4j*-Graphdatenbank festgelegt werden kann.
Neben den Einstellungsmöglichkeiten der Datenbankverbindung sollen auch projektspezifische Angaben gemacht werden können. Es kann ein frei definierbarer Projektname sowie der Schwellenwert für die Zählung der Commit-Hotspots angegeben werden.
Für alle Eingabefelder des Formulars der Einstellungsmöglichkeiten des Dashboards sind Standardwerte definiert. Sofern der Benutzer keine expliziten Anpassungen an den Einstellungen der *Neo4j*-Graphdatenbank oder spezifische Projekteinstellungen vornimmt, können die Standardwerte belassen und gespeichert werden.
Nach der Eingabe der erforderlichen Daten und dem anschließenden Klick auf den *Speichern*-Button wird die Verbindung zu der *Neo4j*-Graphdatenbank getestet. Nur nach einem erfolgreichen Verbindungstest können die weiteren Unterseiten des Dashboards aufgerufen werden, ansonsten erfolgt eine automatische Weiterleitung zurück zu dem Formular der Einstellungsmöglichkeiten. Diese Schutzvorrichtung wurde ergänzt, um Fehler in dem Dashboard zu vermeiden, die auf eine fehlgeschlagene Verbindung zu der *Neo4j*-Graphdatenbank zurückzuführen sind. Abbildung 4.2 zeigt die fertig implementierte Seite der Einstellungsmöglichkeiten des Dashboards.

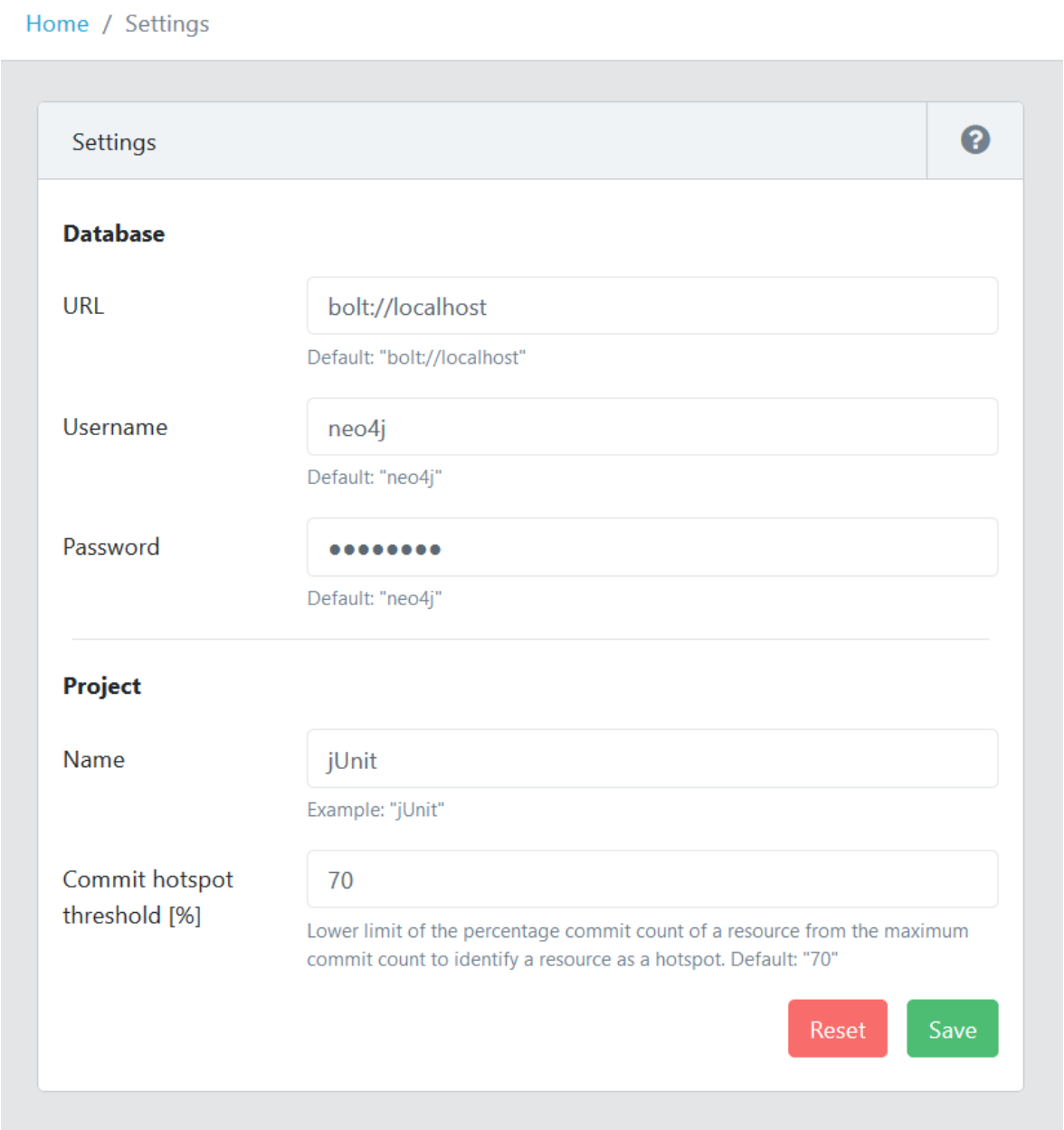

Abbildung 4.2: Einstellungsmöglichkeiten des Dashboards

4.2.3 Benutzerdefinierte Abfragen

Innerhalb des Dashboards soll es die Möglichkeit geben, benutzerdefinierte *Cypher*-Abfragen an die *Neo4j*-Graphdatenbank zu stellen. Als Grundlage dafür kann *Graph App Kit*[37], eine Zusammenfassung von *React*-Komponenten für die Erstellung von *Neo4j*-Applikationen, verwendet werden. Eine der Komponenten von *Graph App Kit* ist der *Cypher*-Editor. Diese Komponente stellt ein Eingabefeld bereit, welches die eingegebenen Daten bei dem Klick auf den *Absenden*-Button direkt an die *Neo4j*-Graphdatenbank sendet. Bei der Formulierung der Abfragen werden die einzelnen Bestandteile der *Cypher*-Abfrage farblich hervorgehoben, um die Lesbarkeit der Abfrage zu verbessern. Die Ausgabe der Abfrageergebnisse erfolgt

[37] `https://github.com/neo4j-apps/graph-app-kit`, Zugriff am: 19.08.2018

mittels einer Tabelle. Diese Tabelle unterstützt sowohl die Seitennummerierung als auch die Möglichkeit der Einstellung, wie viele Ergebniszeilen auf einer Seite angezeigt werden sollen, um die Abfrageergebnisse möglichst übersichtlich zu gestalten. Abbildung 4.3 zeigt die fertig implementierte Seite für benutzerdefinierte *Cypher*-Abfragen in dem Dashboard.

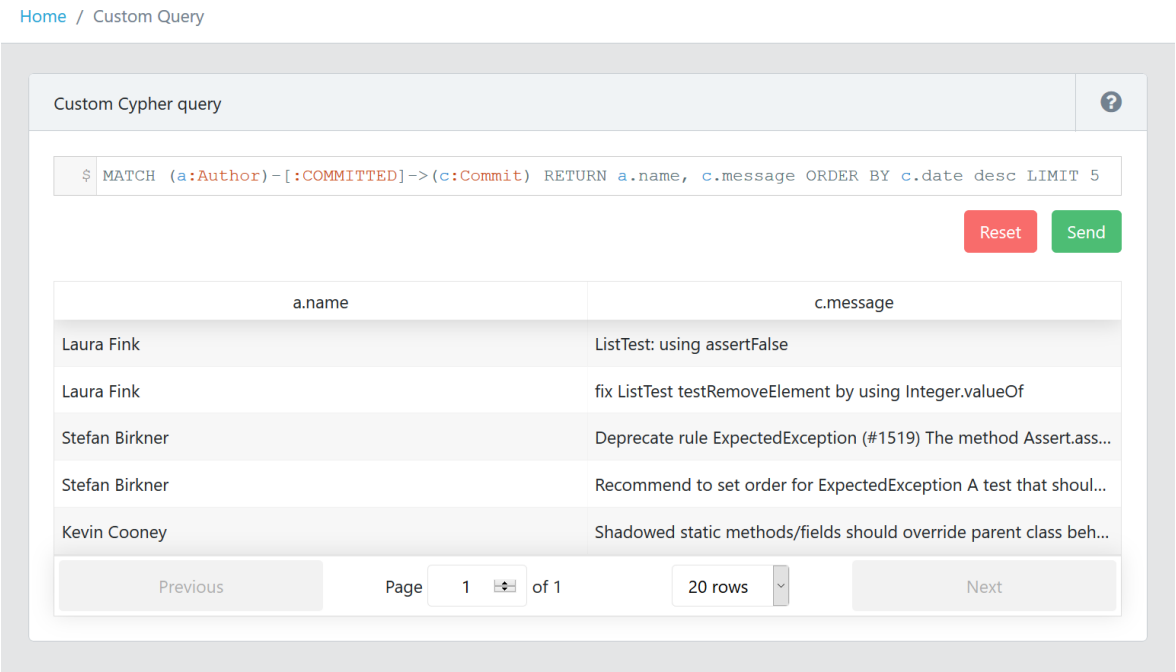

Abbildung 4.3: Benutzerdefinierte Abfragen des Dashboards

4.2.4 Visualisierungskomponenten

In diesem Abschnitt werden die einzelnen Visualisierungskomponenten des Dashboards näher beschrieben. Die Visualisierungskomponenten werden jeweils den einzelnen Wissensbereichen zugeordnet, um die Navigation innerhalb des Dashboards möglichst intuitiv zu gestalten und die bestmögliche Übersicht zu gewährleisten. Innerhalb des Wissensbereichs des Architekturüberblicks sollen die Struktur, die einzelnen Dateitypen sowie die Abhängigkeiten des zu analysierenden Softwareprojekts dargestellt werden. Die Darstellung der Aktivitäten sowie der Wissensverteilung des Softwareprojekts werden anschließend innerhalb des Wissensbereichs des Ressourcenmanagements näher betrachtet. Die Visualisierung der Hotspots fügt sich anschließend innerhalb des Wissensbereichs des Risikomanagements ein. Abschließend erfolgt die Beschreibung der Visualisierungskomponenten der statischen Quellcodeanalyse sowie der Testabdeckung, die sich innerhalb des Wissensbereichs des Qualitätsmanagements befinden.

4.2.4.1 Struktur

Die erste Visualisierungskomponente des Dashboards soll die Struktur des zu analysierenden Softwareprojekts darstellen. Dazu sollen die einzelnen Ressourcen des Softwareprojekts als geschachtelte Kreise dargestellt werden. Um zunächst die darzustellenden Daten zu erhalten, wurde die in Listing 4.7 dargestellte *Cypher*-Abfrage formuliert.

```
1 MATCH (c:Commit)-[:CONTAINS_CHANGE]->()-[:MODIFIES]->(f:File)
2 WHERE NOT c:Merge
3 WITH f, count(c) as commits
4 MATCH (t:Type)-[:HAS_SOURCE]->(f), (t)-[:DECLARES]->(m:Method)
5 RETURN t.fqn as fqn, sum(commits) as commits, sum(m.cyclomaticComplexity) as complexity, sum(m.
      effectiveLineCount) as loc
6 ORDER BY fqn ASCENDING
```

Listing 4.7: *Cypher*-Abfrage für die Ausgabe der Struktur mittels geschachtelter Kreise

Die von der *Neo4j*-Graphdatenbank zurückgegebenen Daten der Abfrage werden für die Erstellung einer Strukturanalyse basierend auf geschachtelten Kreisen verwendet [Tornhill 2018, S. 20]. *Nivo* bietet eine entsprechende Visualisierungskomponente, die für die Umsetzung genutzt werden kann. Den zurückgegebenen vollständig qualifizierten Namen der Ressourcen werden dabei jeweils die Anzahl der Quellcodezeilen zugeordnet, an welchen sich die Größe des jeweiligen Kreises bemisst. Die Farben der Kreise werden anhand der Tiefe der Verschachtelung der jeweiligen Ressource bestimmt, um eine optische Zuordnung der Verschachtelungstiefe zu ermöglichen. Der äußerste Kreis wird dabei am hellsten dargestellt. Bei jeder weiteren Verschachtelung verdunkelt sich die Farbe des Kreises. Beim Überfahren eines Datensatzes mit dem Mauszeiger werden mittels eines benutzerdefinierten Tooltips spezifische Informationen zu der ausgewählten Ressource angezeigt. Abbildung 4.4 zeigt die fertig implementierte Strukturanalyse des Dashboards.

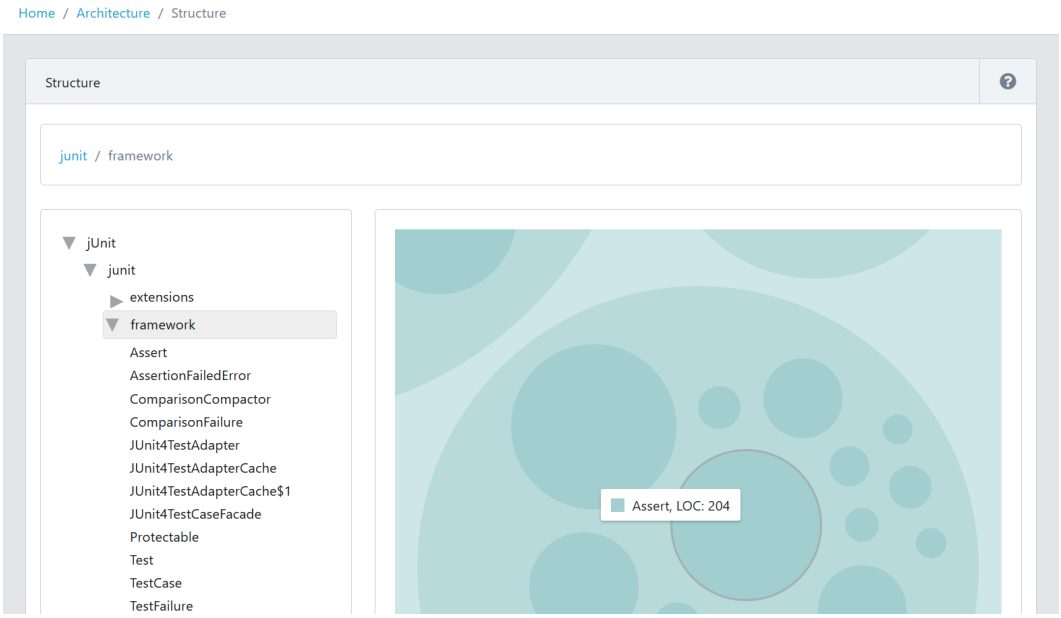

Abbildung 4.4: Struktur-Visualisierungskomponente des Dashboards

4.2.4.2 Dateitypen

Die Analyse der Dateitypen kann Projektleitern Aufschluss über die Verteilung der verwendeten Programmiersprachen eines Softwareprojekts geben. Die in Listing 4.8 abgebildete *Cypher*-Abfrage ermittelt die verschiedenen Dateitypen sowie die jeweils zugehörige Anzahl von Dateien.

```
1 MATCH (f:Git:File)
2 WITH f, split(f.relativePath, ".") as splittedFileName
3 SET f.type = splittedFileName[size(splittedFileName)-1]
4 RETURN f.type as filetype, count(f) as files
5 ORDER BY files DESCENDING
```

Listing 4.8: *Cypher*-Abfrage für die Analyse der Dateitypen

Die zurückgegebenen Daten werden verwendet, um ein Kreisdiagramm zu erstellen, welches die Verteilung der Dateitypen des Softwareprojekts darstellt. *Nivo* enthält eine entsprechende Visualisierungskomponente für die Erstellung von Kreisdiagrammen, mithilfe derer die Darstellung der Dateitypenanalyse umgesetzt werden kann. Bei dem Überfahren eines Dateityps mit dem Mauszeiger wird der Name des entsprechenden Dateityps sowie die zugehörige Anzahl der Dateien dieses Dateityps innerhalb eines benutzerdefinierten Tooltips angezeigt. Abbildung 4.5 zeigt die fertig implementierte Dateitypenanalyse des Dashboards.

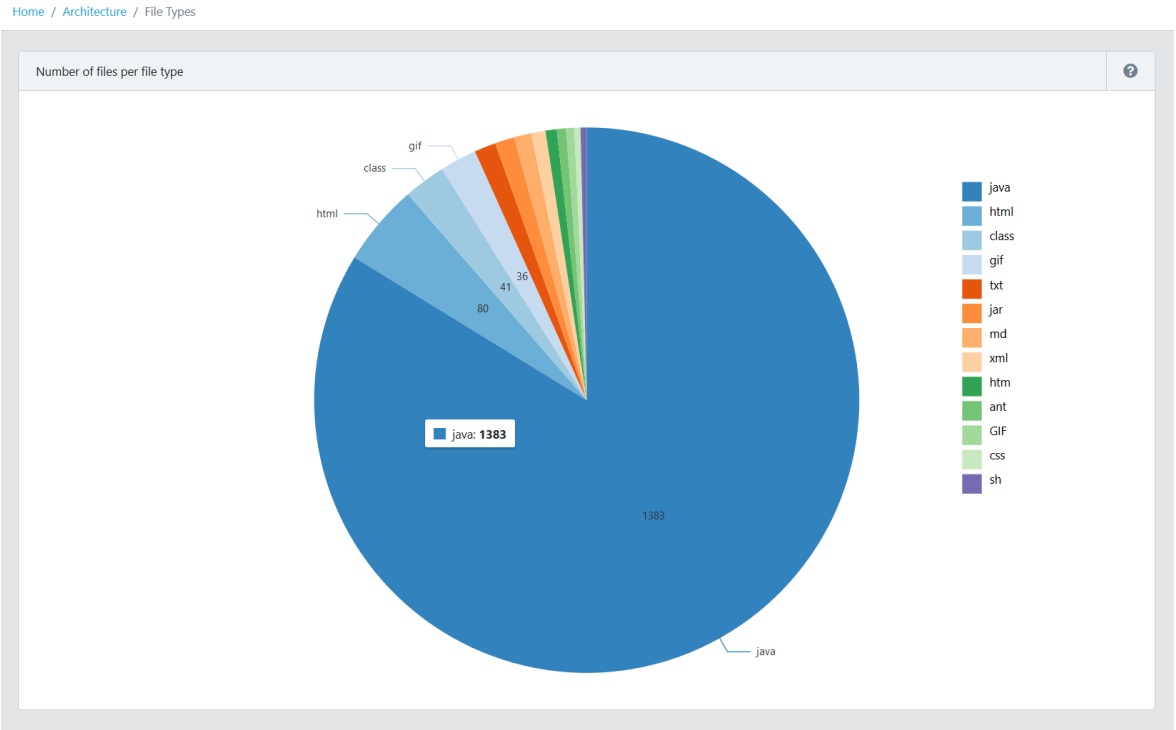

Abbildung 4.5: Dateitypen-Visualisierungskomponente des Dashboards

4.2.4.3 Abhängigkeiten

Eine Abhängigkeitsanalyse kann dazu genutzt werden, um die Kopplung sowie den Zusammenhalt eines Softwaresystems ableiten zu können. Listing 4.9 bildet die *Cypher*-Abfrage ab, welche den vollständig qualifizierten Namen für jede Komponente sowie deren Abhängigkeiten zurückgibt.

```
1 MATCH (dependent_package:Package)-[:CONTAINS]->(dependent:Type)-[depends:DEPENDS_ON]->(dependency:Type)<-[:
      CONTAINS]-(dependency_package:Package)
2 WHERE (dependent)-[:HAS_SOURCE]->(:File)
3 AND (dependency)-[:HAS_SOURCE]->(:File)
4 WITH dependent_package.fqn as dependent, dependency_package.fqn as dependency, count(dependency) as
      dependencies
5 RETURN dependent , dependency, dependencies
6 ORDER BY dependent, dependency
```

Listing 4.9: *Cypher*-Abfrage für die Abhängigkeitsanalyse

Die zurückgegebenen Daten werden verwendet, um eine Abhängigkeitsanalyse zu erstellen. Die Visualisierung der Daten soll dabei mittels eines Akkorddiagramms erfolgen, welches in *Nivo* enthalten ist. Die vollständig qualifizierten Namen der Ressourcen sind radial um einen Kreis angeordnet. Die Abhängigkeiten werden als Bögen innerhalb des Kreises dargestellt. Bei dem Überfahren einer Abhängigkeit mit dem Mauszeiger werden die Namen der Abhängigkeiten dieser Ressource sowie die zugehörige Anzahl von Abhängigkeiten innerhalb eines benutzerdefinierten Tooltips angezeigt. Die entsprechende Darstellung der Abhängigkeitsanalyse zeigt Abbildung 4.6.

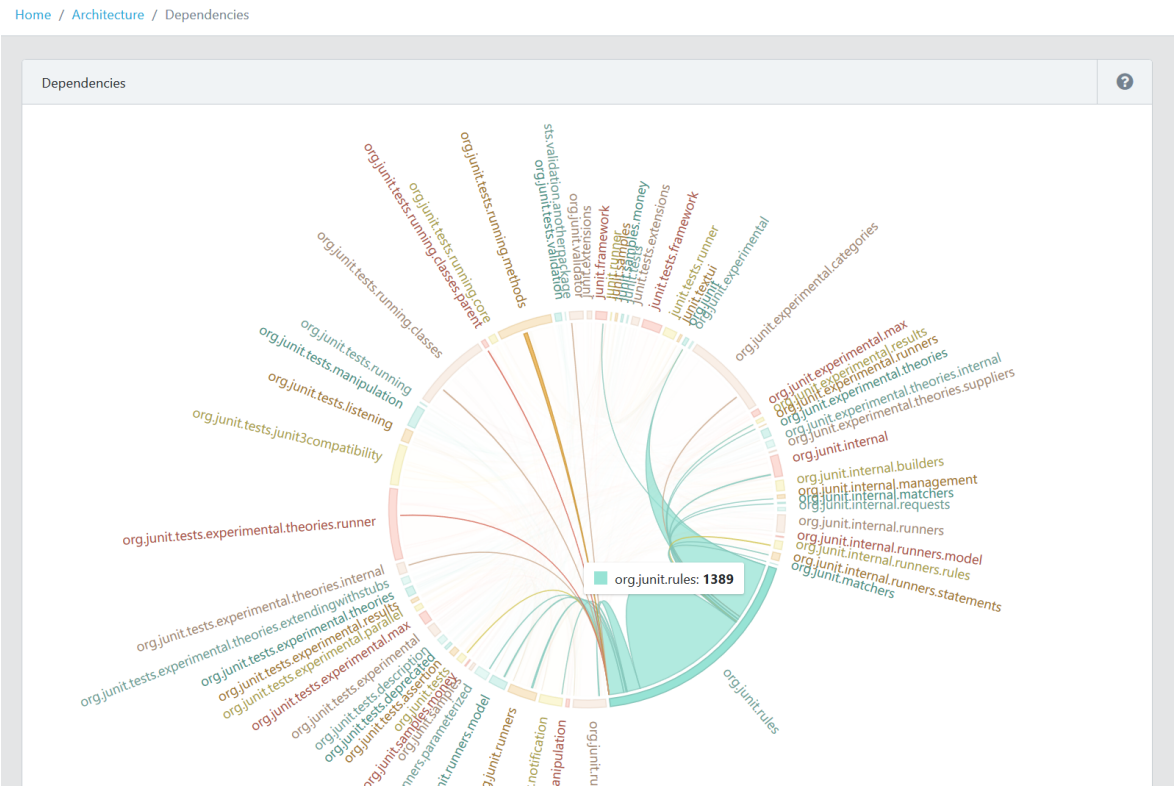

Abbildung 4.6: Abhängigkeiten-Visualisierungskomponente des Dashboards

4.2.4.4 Aktivitäten

Die Analyse der Aktivitäten des Softwareprojekts soll dem Projektleiter einen Überblick über verschiedene Veränderungen geben und somit den Wissensbereich des Ressourcenmanagements unterstützen. Daraus kann beispielsweise abgeleitet werden, woran die einzelnen Autoren gegenwärtig arbeiten. Die in Listing 4.10 abgebildete *Cypher*-Abfrage ermittelt die Namen der Autoren sowie die Anzahl deren jeweiliger Commits.

```
1 MATCH (a:Author)-[:COMMITTED]->(c:Commit)-[:CONTAINS_CHANGE]->(:Change)-[:MODIFIES]->(file:File)
2 WHERE NOT c:Merge
3 AND c.date >= {startDate}
4 AND c.date <= {endDate}
5 RETURN a.name as author, count(distinct c) as commits
6 ORDER BY commits DESCENDING
```

Listing 4.10: *Cypher*-Abfrage für die Ausgabe der Commits pro Autor

Analog zu dieser Abfrage werden in der in Listing 4.11 abgebildeten *Cypher*-Abfrage ebenfalls die Namen der Autoren sowie die jeweils zugehörige Anzahl von geänderten Dateien in dem Softwareprojekt ermittelt.

```
1 MATCH (a:Author)-[:COMMITTED]->(c:Commit)-[:CONTAINS_CHANGE]->(:Change)-[:MODIFIES]->(file:File)
2 WHERE NOT c:Merge
3 AND c.date >= {startDate}
4 AND c.date <= {endDate}
5 RETURN a.name as author, count(file) as files
6 ORDER BY files DESCENDING
```

Listing 4.11: *Cypher*-Abfrage für die Ausgabe der geänderten Dateien pro Autor

Mithilfe der Ergebnisse dieser beiden Abfragen kann jeweils eine Visualisierungskomponente erstellt werden, welche die Autoren nach der Anzahl ihrer Commits beziehungsweise geänderten Dateien sortiert darstellt. Die Umsetzung erfolgt mithilfe der entsprechenden *Nivo*-Komponente für die Erstellung eines Balkendiagramms. Bei dem Überfahren eines Balkens mit dem Mauszeiger wird der Name dieses Autors sowie die Anzahl der zugehörigen Commits beziehungsweise geänderten Dateien innerhalb eines benutzerdefinierten Tooltips angezeigt. Abbildung 4.7 zeigt die fertig implementierten Visualisierungskomponenten in dem Dashboard.

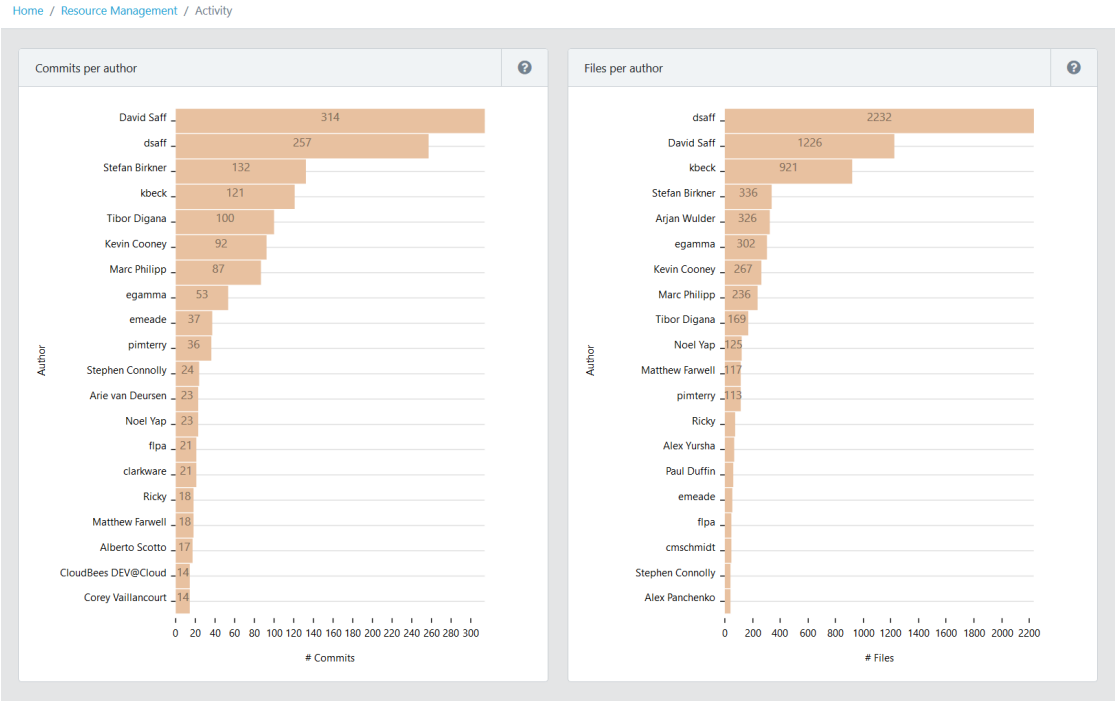

Abbildung 4.7: Visualisierungskomponenten Commits pro Autor sowie Dateien pro Autor

Es können zudem noch weitere Visualisierungen erstellt werden, um die Aktivitäten des Softwareprojekts zu analysieren. Die in Listing 4.12 abgebildete *Cypher*-Abfrage ermittelt die Commits des Softwareprojekts inklusive ihres jeweiligen Erstellungsdatums.

```
1 MATCH (a:Author)-[:COMMITTED]->(c:Commit)-[:CONTAINS_CHANGE]->()-[:MODIFIES]->(f:File), (c)-[:OF_DAY]->(d)
      -[:OF_MONTH]->(m)-[:OF_YEAR]->(y)
2 WHERE NOT c:Merge
3 RETURN y.year as year, m.month as month, d.day as day, count(distinct c) as commits, count(f) as files
4 ORDER BY year, month, day
```

Listing 4.12: *Cypher*-Abfrage für die Ausgabe der Commits sowie zugehörigen Erstellungsdaten

Nivo enthält eine Visualisierungskomponente für die Erstellung von Kalendern. Diese kann dazu verwendet werden, die Commits über den Zeitverlauf der Erstellung des Softwareprojekts darzustellen. Bei dem Überfahren eines Tages in dem Kalender mit dem Mauszeiger wird der jeweilige Tag inklusive der Anzahl der an diesem Tag getätigten Commits innerhalb eines benutzerdefinierten Tooltips angezeigt.
Um dem Projektleiter Informationen über die Inhalte der jeweiligen Commits zur Verfügung zu stellen, können die in dem Softwareprojekt getätigten Commit-Nachrichten zudem noch tabellarisch dargestellt werden. Die in Listing 4.13 abgebildete *Cypher*-Abfrage ermittelt die Namen der Autoren, das jeweilige Datum sowie die zugehörige Commit-Nachricht.

```
MATCH (a:Author)-[:COMMITTED]->(c:Commit)-[:CONTAINS_CHANGE]->(:Change)-[:MODIFIES]->(file:File)
WHERE NOT c:Merge
AND c.date >= {startDate}
AND c.date <= {endDate}
RETURN DISTINCT a.name, c.date, c.message
ORDER BY c.date DESCENDING
LIMIT 20
```

Listing 4.13: *Cypher*-Abfrage für die Ausgabe der letzten 20 Commits

Die zurückgegebenen Daten werden für die Erstellung einer Tabelle verwendet, welche mithilfe der frei verwendbaren *React*-Komponente *react-table* umgesetzt wird. Abbildung 4.8 zeigt die fertig implementierten Visualisierungskomponenten des Commit-Kalenders sowie die Darstellung der letzten 20 Commits in dem Dashboard.

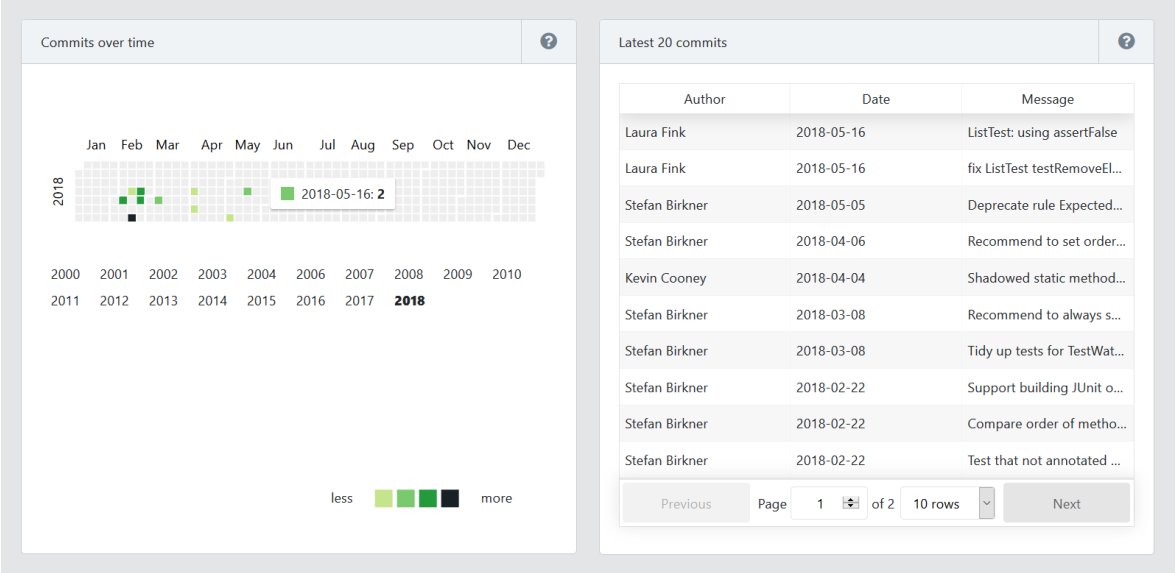

Abbildung 4.8: Visualisierungskomponenten Commitkalender sowie letzte 20 Commits

Da sich die vier beschriebenen Visualisierungskomponenten der Aktivitätsanalyse bisher immer auf das vollständige Zeitintervall der Erstellung des Softwareprojekts beziehen und somit noch keine Filterung auf ein spezifisches Zeitintervall möglich ist, soll dies mittels eines interaktiven Kalenders im Kopfbereich des Dashboards ermöglicht werden. Dieser kann mithilfe der *React*-Komponente *react-bootstrap-daterangepicker* umgesetzt werden. Bei der Auswahl eines Zeitintervalls in dem Kalender sollen alle vier Visualisierungkomponenten der Aktitvitätsanalyse aktualisiert und die Daten entsprechend gefiltert werden. Dies kann mit der Verwendung von Events innerhalb der in Abbildung 3.9 bereits dargestellten *Flux*-Architektur umgesetzt werden. Dazu reagieren die vier bereits vorhandenen Visualisierungskomponenten auf die *Action DATERANGEPICKER_MODIFIED*, welche sich anpasst, sobald Veränderungen des Intervalls in dem Kalender stattfinden. Das in dem Kalender ausgewählte Intervall wird bei dem Klick auf *Übernehmen* somit an die vier Visualisierungskomponenten übergeben. Daraufhin werden diese automatisch neu gerendert und grenzen somit die jeweiligen Darstellungen auf das ausgewählte Zeitintervall ein. Die *Cypher*-Abfragen

müssen für die Umsetzung dieser Filterung nicht erneut angepasst werden, da diese bereits mit den Parametern *{startDate}* beziehungsweise *{endDate}* ausgestattet sind, welche dynamisch übergeben werden. Abbildung 4.9 zeigt die fertig implementierten Visualisierungskomponenten des Kalenders für die Auswahl eines Zeitintervalls, nach welchem die Visualisierungskomponenten der Aktivitätsanalyse gefiltert werden können.

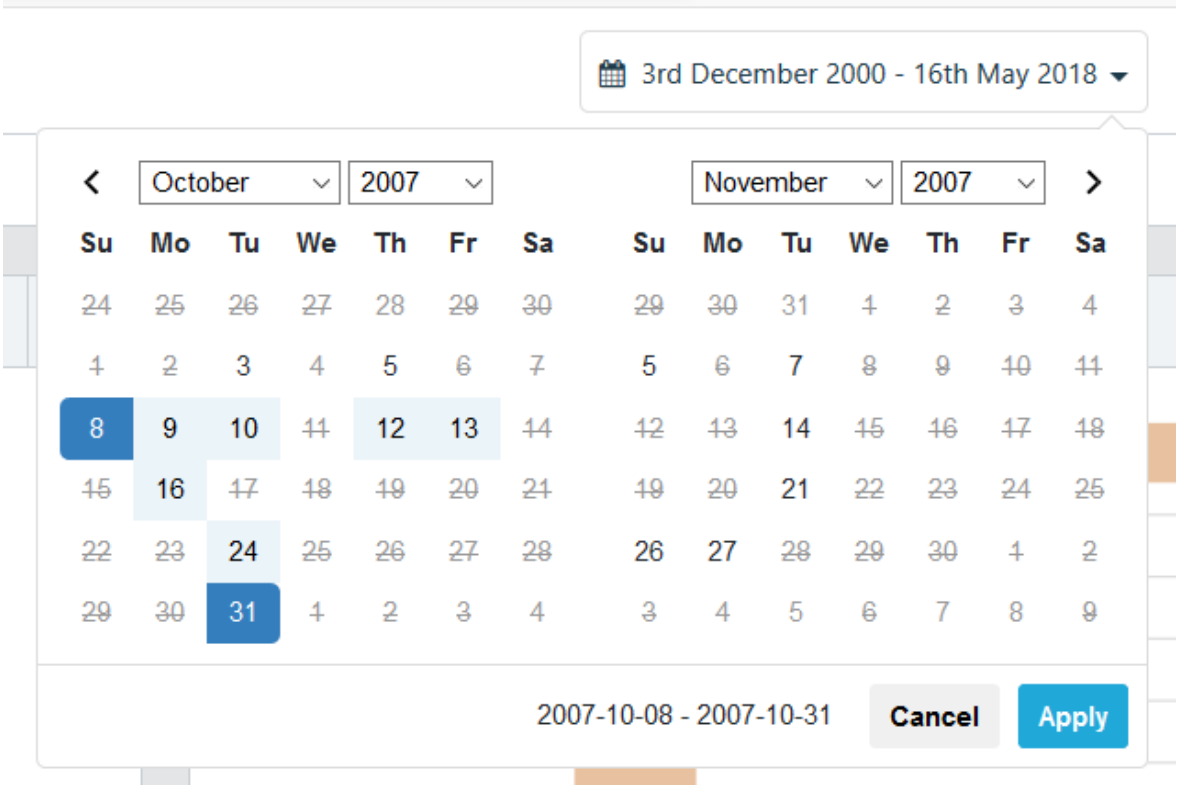

Abbildung 4.9: Kalenderkomponente zur Filterung der Aktivitäten

4.2.4.5 Wissensverteilung

Die Analyse der Wissensverteilung kann Projektleiter bei der Zuweisung von Ressourcen unterstützen. Die in Listing 4.14 abgebildete *Cypher*-Abfrage ermittelt die verschiedenen Dateitypen des Softwareprojekts, die Namen der Autoren sowie die Anzahl der Dateien des jeweiligen Dateityps.

```
1 MATCH (a:Author)-[:COMMITTED]->(c:Commit)-[:CONTAINS_CHANGE]->(:Change)-[:MODIFIES]->(file:File)
2 WHERE NOT c:Merge
3 RETURN file.type as filetype, a.name as author, count(file) as files
4 ORDER BY files DESCENDING, filetype
```

Listing 4.14: *Cypher*-Abfrage für die Analyse der Wissensverteilung

Die zurückgegebenen Daten werden verwendet, um ein gestapeltes Balkendiagramm zu erstellen, welches die am Softwareprojekt beteiligten Autoren mitsamt deren jeweils in den Commits enthaltenen Dateitypen darstellt. Die Länge des jeweiligen Balkens wird durch die Anzahl der Dateien des Dateityps bestimmt. Auch hier wird bei der Umsetzung auf die entsprechende *Nivo*-Komponente zurückgegriffen. Bei dem Überfahren eines Dateityps mit

dem Mauszeiger wird der Name dieses Dateityps sowie die zugehörige Anzahl von Dateien innerhalb eines benutzerdefinierten Tooltips angezeigt. Abbildung 4.10 zeigt die fertig implementierte Analyse der Wissensverteilung des Dashboards.

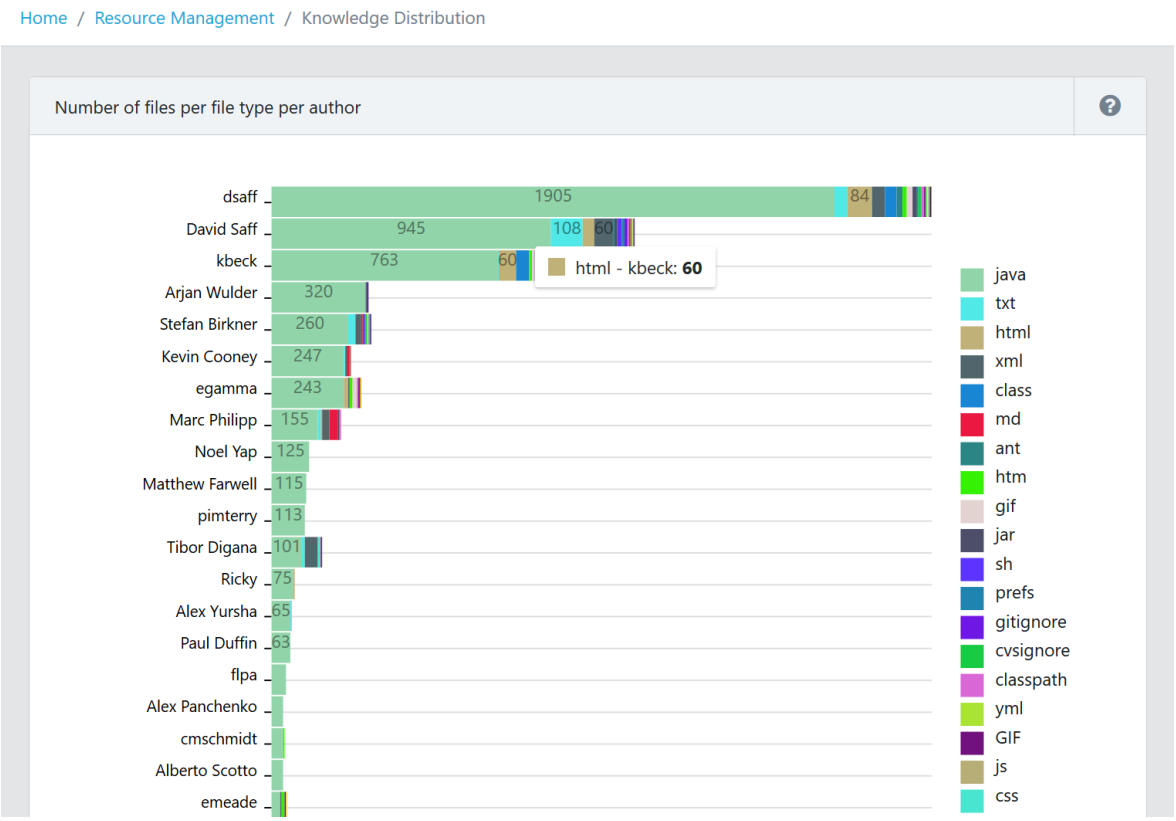

Abbildung 4.10: Visualisierungskomponente zur Darstellung der Wissensverteilung

4.2.4.6 Hotspots

Als Hotspots werden komplexe Teile des Quellcodes bezeichnet, die sich oft ändern [Tornhill 2018, S. 19]. Im Regelfall sind diese Kandidaten für Verbesserungen oder Refaktorisierung. Im Kontext dieser Arbeit werden unter Hotspots Commit-Hotspots verstanden, also Teile des Quellcodes, die sehr häufig in den Commits des Softwareprojekts auftauchen.
Die *Cypher*-Abfrage der Struktur des zu analysierenden Softwareprojekts wurde so formuliert, dass diese für die Visualisierungskomponente der Hotspots vollständig wiederverwendet werden kann. Für die Datengewinnung der Hotspots wird demnach keine zusätzliche *Cypher*-Abfrage benötigt. Die von der *Neo4j*-Graphdatenbank zurückgegebenen Daten der Abfrage werden zur Erstellung einer Hotspot-Analyseansicht basierend auf geschachtelten Kreisen [Tornhill 2018, S. 20] verwendet. *Nivo* bietet eine entsprechende Visualisierungskomponente, die für die Umsetzung genutzt werden kann. Den zurückgegebenen vollständig qualifizierten Namen der Ressourcen werden dabei jeweils die Anzahl der Quellcodezeilen sowie die Anzahl der Commits dieser Ressource zugeordnet. Die Größe des jeweiligen Kreises bemisst sich an der Anzahl der Quellcodezeilen und die Anzahl der Commits legt die Farbe des Kreises fest. Um die konkrete Farbe des Kreises festzulegen, wird ein Farbverlauf

von dem Grünton der jeweiligen Verschachtelungstiefe der Ressource bis hin zu Signalrot definiert. Dieser Farbverlauf kann nun als eine Skala von 0, also dem Grünton, bis 1, also dem Signalrotton, betrachtet werden. Die konkrete Farbe des Kreises ergibt sich aus der Anzahl der Commits der betrachteten Ressource geteilt durch die maximale Anzahl von Commits für eine Ressource des Softwareprojekts. Bei dem Überfahren einer Ressource mit dem Mauszeiger wird der Name dieser Ressource, die zugehörige Anzahl von Quellcodezeilen sowie die Anzahl der Commits innerhalb eines benutzerdefinierten Tooltips angezeigt.
Oberhalb der Darstellung der geschachtelten Kreise befindet sich die Anzeige der Breadcrumbs, also der jeweiligen Pfade der Ressourcen. Beim Klick auf einen Teilpfad der Breadcrumbs wird der Fokus auf das ausgewählte Paket gesetzt und der entsprechende Kreis im Browser des Nutzers zentriert sowie vergrößert. Neben dem Kreisdiagramm befindet sich der Paketexplorer. Darin werden die Ressourcen des Projekts in Form eines Baumdiagramms abgebildet. Beim Klick auf eine Ressource wird die Anzeige der aktuellen Breadcrumbs angepasst sowie analog zum Klick auf einen Teil der Breadcrumbs die ausgewählte Ressource in den Fokus gesetzt. Abbildung 4.11 zeigt die fertig implementierte Hotspot-Analyse in dem Dashboard.

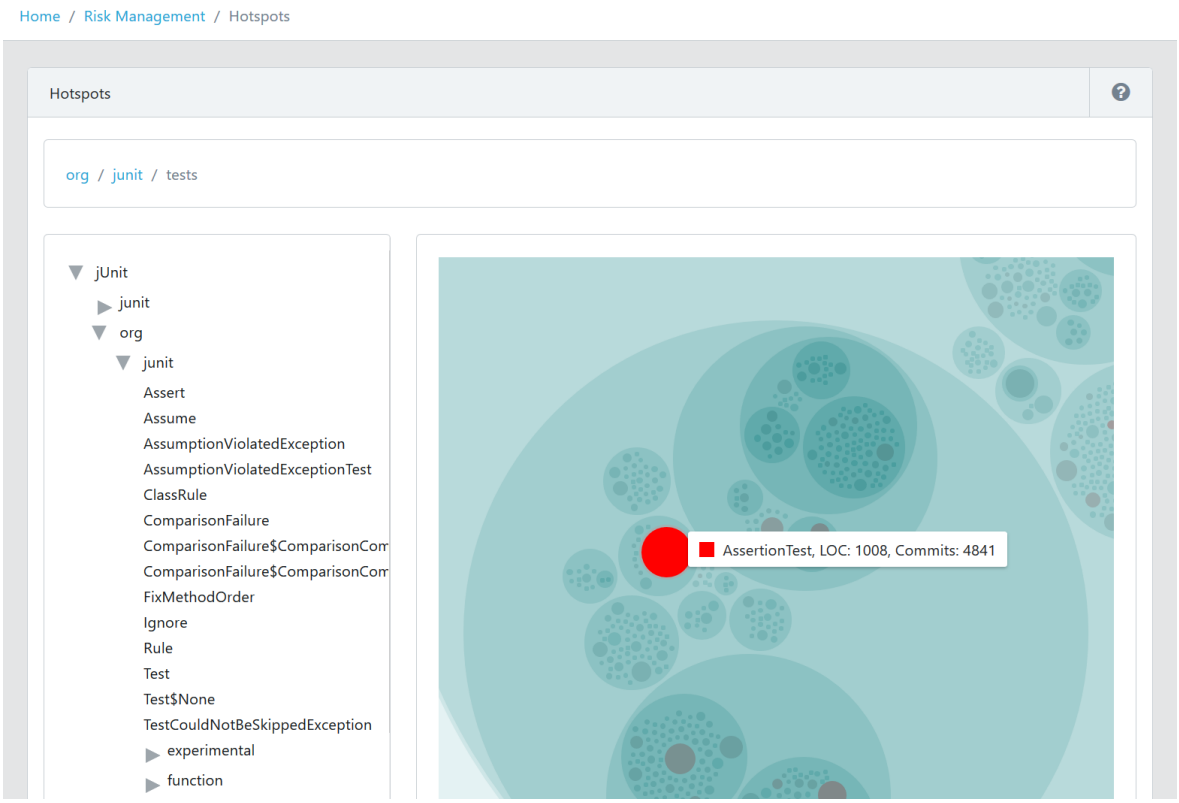

Abbildung 4.11: Hotspots-Visualisierungskomponente des Dashboards

4.2.4.7 Statische Quellcodeanalyse

Die statische Quellcodeanalyse ist ein Mittel für die Bewertung der Qualität eines Softwareprojekts. Die Software *PMD*[38] bietet Kontrollen hinsichtlich vordefinierter Regeln. Dazu zählen gängige Praktiken (engl. *best practices*), Quellcodestil, Design, Fehleranfälligkeit sowie Multithreading. Mittels eines *jQAssistant*-Scanner-Plugins können die durch *PMD* ermittelten Daten für das jeweilige Softwareprojekt in der *Neo4j*-Graphdatenbank ergänzt werden. Die in Listing 4.15 abgebildete *Cypher*-Abfrage ermittelt den jeweiligen vollständig qualifizierten Namen der Ressourcen sowie weitere Daten für die Beschreibung etwaiger Verstöße gegen die genannten Regeln.

```
1 MATCH (:Report)-[:HAS_FILE]->(file:File:Pmd)-[:HAS_VIOLATION]->(violation:Violation)
2 RETURN file.fqn, violation.package, violation.className, violation.method, violation.beginLine, violation.
      endLine, violation.beginColumn, violation.endColumn, violation.message, violation.ruleSet, violation.
      priority, violation.externalInfoUrl
```

Listing 4.15: *Cypher*-Abfrage für die statische Quellcodeanalyse

Die zurückgegebenen Daten werden verwendet, um ein Radardiagramm zu erstellen, welches die Anzahl der Verstöße in den genannten Kategorien darstellt. Auch hier wird bei der Umsetzung auf die entsprechende *Nivo*-Komponente zurückgegriffen. Bei dem Überfahren einer Kategorie mit dem Mauszeiger wird der Name dieser Kategorie sowie die zugehörige Anzahl von Verstößen gegen diese innerhalb eines benutzerdefinierten Tooltips angezeigt. Abbildung 4.12 zeigt die fertig implementierte Visualisierungskomponente der statischen Quellcodeanalyse in dem Dashboard.

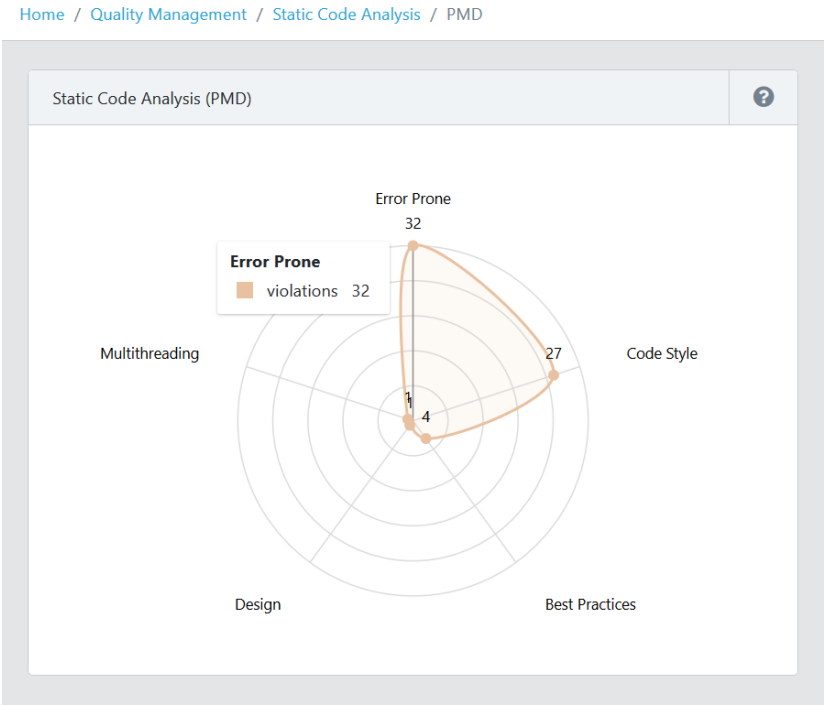

Abbildung 4.12: Visualisierungskomponente für die Darstellung der statischen Quellcodeanalyse

[38] `https://pmd.github.io/`, Zugriff am: 21.08.2018

Die konkreten Verstöße werden außerdem unterhalb der beschriebenen Visualisierungskomponente gruppiert nach ihrer jeweiligen Kategorie aufgeführt und in einzelnen Boxen dargestellt. Jeder Verstoß wird entsprechend seiner Priorität gefärbt und mit spezifischen Informationen zur Auffindung und Behebung des Verstoßes versehen. Abbildung 4.13 zeigt die exemplarische Darstellung dieser Detailansicht der Verstöße.

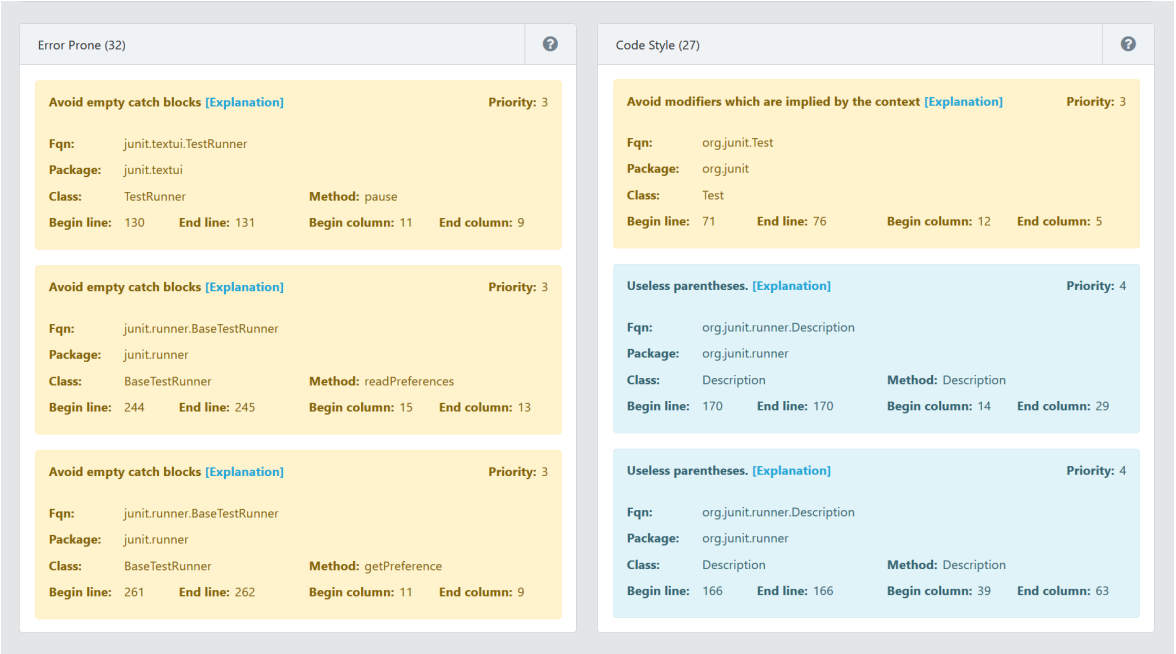

Abbildung 4.13: Darstellung der Details der einzelnen Verstöße im Quellcode

4.2.4.8 Testabdeckung

Die Analyse der Testabdeckung kann Projektleiter bei der Beurteilung der Qualität eines Softwareprojekts unterstützen. Die in Listing 4.16 dargestellte *Cypher*-Abfrage ermittelt die vollständig qualifizierten Namen für alle Klassen, die Signatur der jeweiligen Methoden, die entsprechende prozentuale Testabdeckung der Ressource sowie die zugehörige Anzahl der Quellcodezeilen.

```
1 MATCH (c:Jacoco:Class)-[:HAS_METHOD]->(m:Method:Jacoco)-[:HAS_COUNTER]->(cnt:Counter)
2 WHERE cnt.type='INSTRUCTION'
3 RETURN c.fqn as fqn, m.signature as signature,(cnt.covered*100)/(cnt.covered+cnt.missed) as coverage, cnt.
      covered+cnt.missed as loc
4 ORDER BY fqn, signature ASCENDING
```

Listing 4.16: *Cypher*-Abfrage für die Analyse der Testabdeckung

Die zurückgegebenen Daten werden für die Erstellung einer Analyse der Testabdeckung mittels einer Baumkarte (engl. *treemap*) verwendet, welche mithilfe der entsprechenden *Nivo*-Visualisierungskomponente umgesetzt wird. Die vollständig qualifizierten Namen der Typen und Methodensignaturen werden verschachtelten Rechtecken zugeordnet. Die Größe des Rechtecks wird durch die Anzahl der Quellcodezeilen definiert. Die prozentuale Testabdeckung bestimmt die Farbe des Rechtecks analog der geschachtelten Kreise der Hotspot-

Visualisierung. Bei dem Überfahren einer Ressource mit dem Mauszeiger wird der vollqualifizierte Name dieser Ressource, der Name der zugehörigen Methode, die zugehörige Anzahl von Quellcodezeilen sowie die prozentuale Testabdeckung der Ressource innerhalb eines benutzerdefinierten Tooltips angezeigt. Abbildung 4.14 zeigt die fertig implementierte Seite der Hotspots in dem Dashboard.

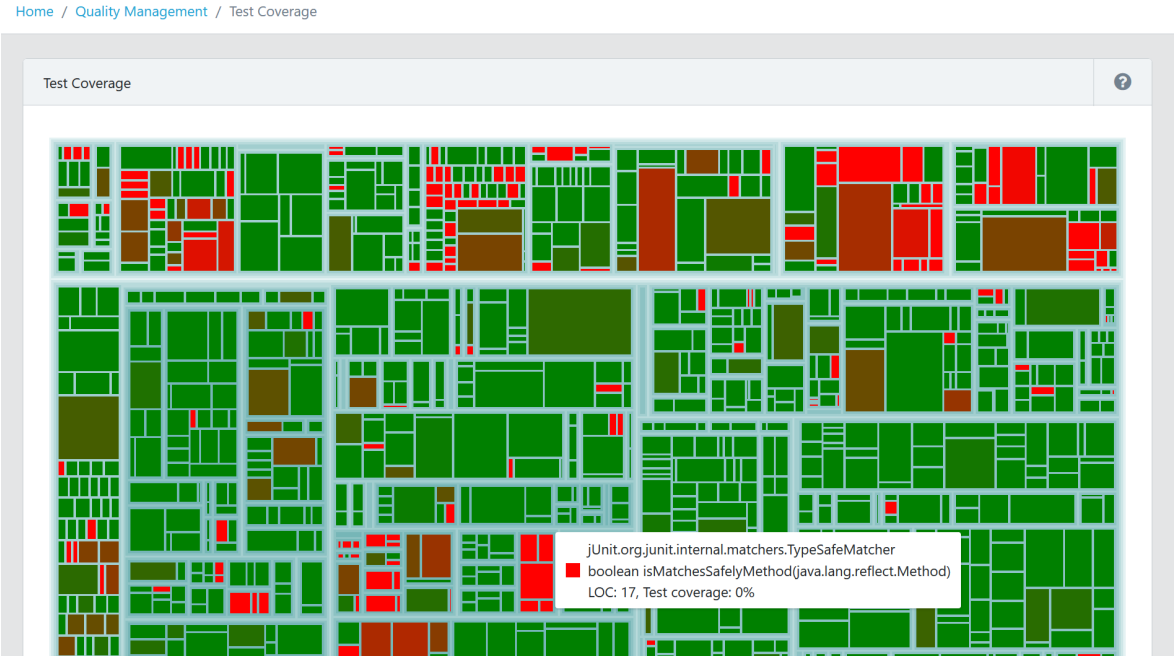

Abbildung 4.14: Visualisierungskomponente zur Darstellung der Testabdeckung

4.2.4.9 Erstellung

In diesem Abschnitt wird die Erstellung einer neuen Visualisierungskomponente am Beispiel der Dateitypen innerhalb des Wissensbereichs des Architekturüberblicks aus Abschnitt 4.2.4.2 erläutert. Für die bessere Nachvollziehbarkeit der genannten Dateipfade und -namen ist eine detaillierte Beschreibung der Dateistruktur des Dashboards in Anhang B.1 zu finden. Zunächst müssen die Anforderungen der Visualisierungskomponente festgelegt werden. Dazu muss die Frage beantwortet werden, welche Daten visualisiert werden sollen, und eine entsprechende *Cypher*-Abfrage formuliert werden. Danach wird die Art der Datenvisualisierung festgelegt. Für das Beispiel der Dateitypen soll ein Kreisdiagramm umgesetzt werden. Anschließend folgt die Auswahl der *React*-Visualisierungskomponente. Für die Ausgabe der Dateitypen wurde die *Nivo*-Pie-Chart ausgewählt. Letztendlich muss der zu erstellenden Visualisierungskomponente eines der vier Wissensgebiete zugewiesen werden. Die Analyse der Dateitypen wurde dem Wissensgebiet des Architekturüberblicks zugewiesen. Damit stehen die Anforderungen an die Visualisierungskomponente fest. Nun müssen die entsprechenden Dateien in dem Dashboard angelegt werden. Zunächst sollte ein Unterverzeichnis für die Visualisierung im jeweiligen Verzeichnis des Aufgabenbereichs angelegt werden, also beispielsweise `views/Dashboard/Architecture/FileTypes/`. Danach wird in-

nerhalb dieses Verzeichnisses das Unterverzeichnis `visualization` angelegt, also beispielsweise `views/Dashboard/Architecture/FileTypes/visualization/`. Darin wird nun die Visualisierungskomponente erstellt und `FileType.js` genannt. Es folgt nun das Importieren der benötigten Ressourcen `React` sowie `DashboardAbstract`, also der Klasse, die die Verbindung zu der *Neo4j*-Graphdatenbank herstellt. Des Weiteren sollte die Programmierschnittstelle (API), welche die Datenbankabfrage sowie die Datenaufbereitung enthält, importiert werden, nachdem diese implementiert wurde. Die jeweilige Komponente für die Darstellung, welche im Falle eines Kreisdiagramms `Nivo ResponsivePie` entspricht, schließt die zu importierenden Ressourcen dieser Datei ab. Daraufhin wird die Klasse der Visualisierungskomponente definiert. Diese wird immer von `DashboardAbstract` abgeleitet, um innerhalb der Klasse eine Verbindung mit der *Neo4j*-Graphdatenbank herstellen zu können. Danach erfolgt die Implementierung des Lebenszyklus der *React*-Komponente. Mittels `componentWillMount` kann festgelegt werden, was kurz vor der Einbindung der Komponente geschehen soll. Hier sollte in jedem Fall die Elternklasse aufgerufen werden, also `super.componentWillMount`. Mittels `componentDidMount` kann festgelegt werden, was direkt nach der Einbindung der Komponente geschehen soll. An dieser Stelle sollten die Instanziierung des `Models` sowie der Aufruf der Datenrecherche implementiert werden. Mittels `render` kann festgelegt werden, was bei der direkten Ausgabe der Komponente geschieht. Hier sollte immer eine Weiterleitung zu der Seite der Einstellungsmöglichkeiten des Dashboards eingebunden werden, falls keine Verbindung zu der *Neo4j*-Graphdatenbank hergestellt werden kann. Anschließend erfolgt die Rückgabe der jeweiligen Komponente, um die eigentliche Visualisierung auszugeben. Abschließend wird die Komponente exportiert, um diese für die anderen Komponenten des Dashboards nutzbar zu machen.

Analog zu der soeben angelegten Komponente sollte zudem eine Wrapper-Komponente angelegt werden, in der die gewünschten Hilfskomponenten für die Anzeige der Visualisierung aus der in *CoreUI* enthaltenen Komponente `react-strap` importiert und eingebunden werden. Die Visualisierungskomponente der Dateitypen wird innerhalb einer `Card` dargestellt. Für die Dateitypen wurde die zugehörige Wrapper-Komponente unter `views/Dashboard/Architecture/FileTypes/FileTypes.js` angelegt.

Danach erfolgt die Anlage der Datei zur Abfrage der Datenbank und anschließenden Datenaufbereitung. Am Beispiel der Dateitypen wurde die Datei `FileTypes.js` im Verzeichnis `api/models/` angelegt. Darin wird `DashboardAbstract` sowie `D3.js` importiert und anschließend eine Funktion ergänzt, die die Datenbankabfrage ausführt sowie die erhaltenen Daten aufbereitet. Bei mehrfach genutzten Hilfsfunktionen sollten diese in das Verzeichnis `api/utils/` ausgelagert werden.

Der nächste Schritt ist die Festlegung des Aufrufpfads sowie der Verlinkung. In der Datei `routes.js` im Hauptverzeichnis des Dashboards ist dazu der Aufrufpfad der Visualisierung zu ergänzen. Außerdem muss in der Datei `_nav.js` die Verlinkung zu der Wrapper-Komponente der Visualisierung in dem entsprechenden Aufgabenbereich ergänzt werden.
Abschließend sollte ein Test für die Visualisierungskomponente erstellt werden, um die Testabdeckung des Dashboards und damit die Qualität des Quellcodes zu erhöhen. Ein solcher Test wird in Abschnitt 4.3.1.1 beispielhaft beschrieben. Damit ist die Erstellung der neuen Visualisierungskomponente erfolgreich abgeschlossen.

4.3 Eingesetzte Werkzeuge

In diesem Abschnitt wird das Dashboard um einige Werkzeuge erweitert, die dabei helfen können, das Dashboard weiterzuentwickeln und zu testen sowie die Qualitätsziele der Installier- und Wartbarkeit weiter zu unterstützen.

4.3.1 Entwicklung und Test

Um die Entwicklung des Dashboards zu vereinfachen und das Testen der einzelnen Komponenten des Dashboards zu ermöglichen sowie anschließend zu automatisieren, wird eine Reihe von Werkzeugen in das Dashboard eingebaut.

4.3.1.1 Jest

Um die Funktion der erstellten *React*-Komponenten zu testen, wird das eigens von dem Unternehmen *Facebook* entwickelte Testframework *Jest*[39] genutzt. An dem Beispiel der Komponente für die Darstellung der Breadcrumbs innerhalb des Dashboards wird nun ein Test erstellt. Dieser soll sicherstellen, dass die an die Komponente übergebenen Elemente korrekt als Links ausgegeben werden. Dazu wird die Visualisierungskomponente mit Beispieldaten befüllt und anschließend gerendert. Danach wird das tatsächliche mit dem erwarteten Ergebnis mittels des Befehls *expect* verglichen. Listing 4.17 zeigt den vollständigen Test der Breadcrumbs-Komponente des Dashboards.

[39] `https://jestjs.io/en/`, Zugriff am: 21.08.2018

```
import React from "react";

import { shallow, mount, render } from "enzyme";
import Enzyme from "enzyme";
import { expect } from "chai";
import Adapter from "enzyme-adapter-react-16";
Enzyme.configure({ adapter: new Adapter() });

//component to test
import DynamicBreadcrumb from "../../../../views/Dashboard/DynamicBreadcrumb/DynamicBreadcrumb";

describe("DynamicBreadcrumbTest", () => {
  it("should render without throwing an error", () => {
    var wrapper = shallow(
      <DynamicBreadcrumb items={["a", "b"]} separator={"-"} />
    );
    var html = wrapper.html();

    expect(html).to.contain(
      '<a id="a" class="breadcrumb-item breadcrumb-item">a</a>'
    );
    expect(html).to.contain(
      '<a id="a-b" class="breadcrumb-item breadcrumb-item">b</a>'
    );
  });
});
```

Listing 4.17: Test der Breadcrumb-Komponente des Dashboards

Analog wurden Tests für alle in dem Rahmen der Arbeit erstellten *React*-Komponenten angelegt. Die Tests können mithilfe des Befehls `npm run test-dashboard` ausgeführt werden.

4.3.1.2 Codecov

Um die Qualität des Quellcodes des Dashboards und somit auch die Qualitätsziele der Wartbarkeit sowie der Erweiterbarkeit zu unterstützen, wird die Testabdeckung des Dashboard-Quellcodes mithilfe des kostenfreien Werkzeugs *Codecov*[40] analysiert. Nach Ausführung der Tests des Dashboards wird das Verzeichnis *coverage* im Hauptverzeichnis des Dashboard-Quellcodes angelegt, in dem sich die Ergebnisse der ausgeführten Tests befinden. Auf die Daten dieses Verzeichnisses kann *Codecov* zugreifen und die Testergebnisse analysieren. Abbildung 4.15 zeigt einen Ausschnitt der Auswertung des Dashboard-Quellcodes[41]. Die Gesamttestabdeckung des Quellcodes liegt zu dem Zeitpunkt der Erstellung der Arbeit bei 50,88%.

[40] `https://codecov.io/`, Zugriff am: 08.08.2018

[41] `https://codecov.io/gh/softvis-research/jqa-dashboard/tree/master/src/views/Dashboard`, Zugriff am: 21.08.2018

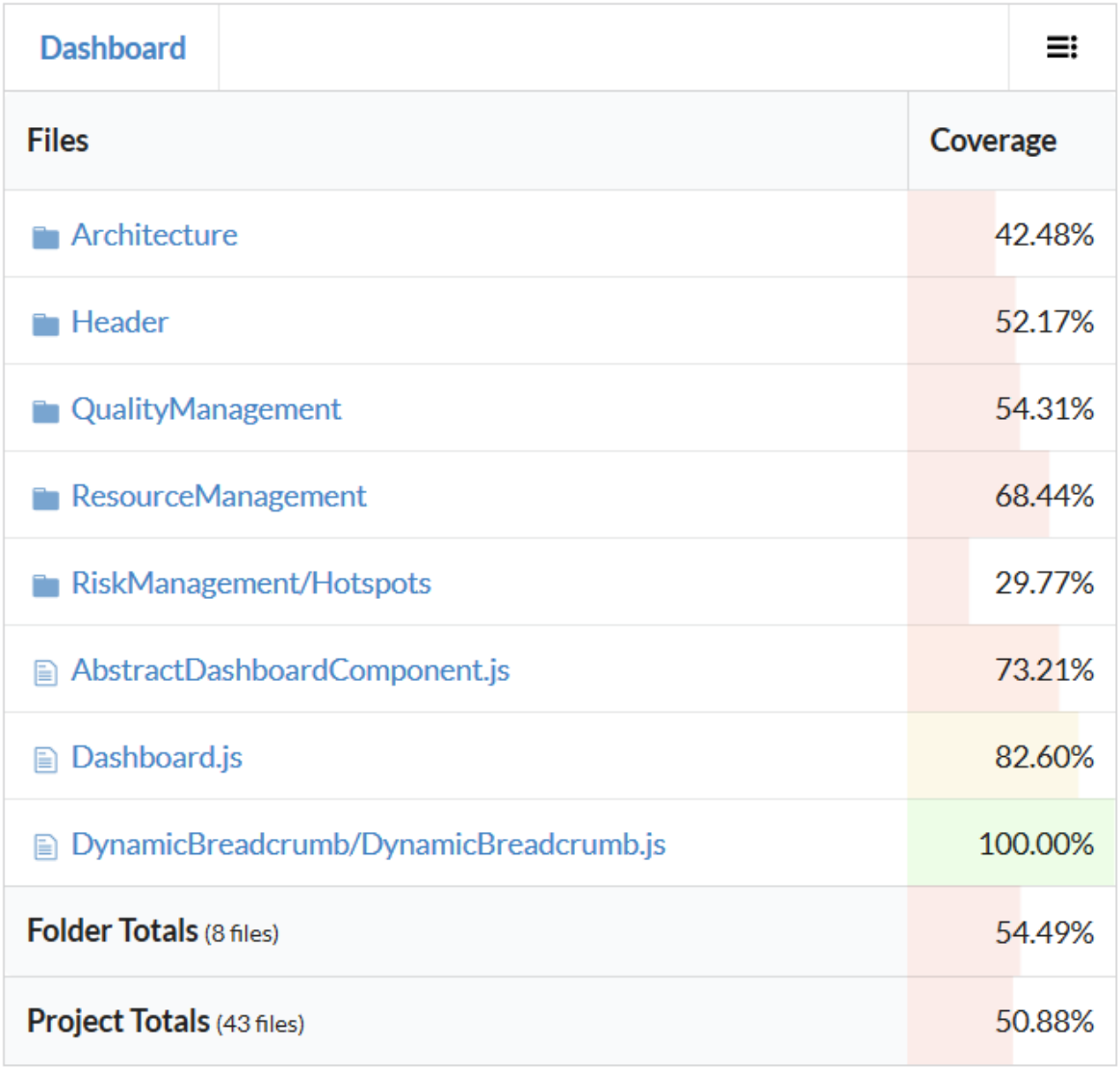

Abbildung 4.15: Graphische Darstellung der Testabdeckung des Dashboard-Quellcodes

4.3.1.3 Travis CI

Um die Wartbarkeit und somit die Qualität des Dashboards weiter zu erhöhen, sollen die in Abschnitt 4.3.1.1 beschriebenen Tests bei jeder neuen Anpassung des Dashboard-Quellcodes ausgeführt werden. Für die Umsetzung dieser automatischen Qualitätskontrolle des Dashboards wird *Travis CI* eingesetzt. *Travis CI* ist eine Open-Source-Software für die kontinuierliche Integration und kann kostenfrei für Softwareprojekte, die auf *GitHub* bereitgestellt sind, verwendet werden. Nach jedem neuen Commit im Quellcode des Dashboards werden mittels *Travis CI* alle angefertigten Tests ausgeführt und anschließend entsprechend an dem Commit gekennzeichnet. Abbildung 4.16 zeigt diese Kennzeichnung an einem Commit, welcher alle Tests erfolgreich bestanden hat, gefolgt von einem Commit mit fehlgeschlagenen Tests.

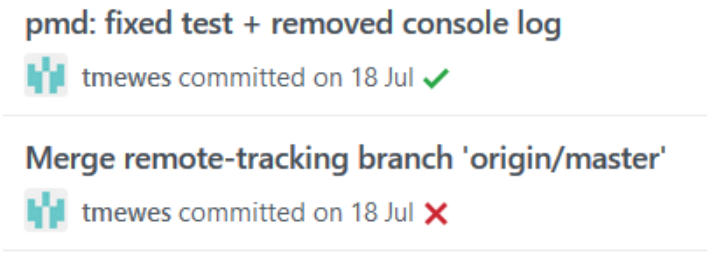

Abbildung 4.16: Kennzeichnung von Commits mittels *Travis CI*

Durch diese automatische und kontinuierliche Qualitätskontrolle des Dashboards sollen negative Seiteneffekte und Fehler bei zukünftigen Anpassungen des Dashboards vermieden und die Qualität des Quellcodes gefördert werden.

4.3.1.4 Prettier

Um das Qualitätsziel der Wartbarkeit weiter zu unterstützen, soll der Quellcode des Dashboards einheitlich formatiert werden. Dazu wird die Open-Source-Software *Prettier*[42] verwendet. *Prettier* formatiert Quellcode anhand vordefinierter Standards. Um alle Dateien des Dashboards einheitlich zu formatieren, kann in dem Hauptverzeichnis des Dashboards der Befehl `npm run pretty` ausgeführt werden.
Zusätzlich wurde *Prettier* in der *Pre-Commit-Hook* des *Git-Repository* eingebaut, sodass vor jedem Commit der oben genannte Befehl automatisch ausgeführt wird. Dabei wird erkannt, welche Dateien angepasst wurden. Der Quellcode dieser angepassten Dateien wird dann ohne weiteres Zutun vereinheitlicht. So kann sichergestellt werden, dass sich jeder Mitwirkende des Dashboards an den einheitlichen Quellcodestil hält.

4.3.1.5 Docker

Um das Kapitel der Implementierung abzuschließen und das Qualitätsziel der Installierbarkeit zu fördern, wird ein *Docker-Image* für das Dashboard erzeugt, mit welchem das Dashboard mittels eines einzigen Befehls installiert werden kann. *Docker* ist eine Open-Source-Software zur Isolierung von Anwendungen mit Containervirtualisierung, die sich durch besonders benutzerfreundliche Eigenschaften auszeichnet und den Begriff *Docker-Container* als Alternative zu virtuellen Maschinen populär gemacht hat. Ein Container fasst eine einzelne Anwendung mitsamt aller Abhängigkeiten wie Bibliotheken, Hilfsprogrammen und statischer Daten in einer Image-Datei zusammen, ohne jedoch ein komplettes Betriebssystem zu beinhalten. Daher lassen sich Container mit einer leichtgewichtigen Virtualisierung vergleichen [Augsten 2017]. Um ein *Docker-Image* zu erstellen, welches das Dashboard installieren und ausführen soll, wurde zunächst die Datei `Dockerfile` mit dem in Listing 4.18 dargestellten Inhalt in dem Hauptverzeichnis des Dashboards angelegt.

[42] `https://prettier.io/`, Zugriff am: 21.08.2018

```
# Specify the version of Node.js
FROM node:8.11.1

# Copy all local files into the image.
COPY . .

# Install and run the dashboard
RUN npm run install-dashboard
CMD npm run dashboard
EXPOSE 3000
```

Listing 4.18: Dockerfile des Dashboards

Anschließend wurde mittels des Befehls aus Listing 4.19 ein neues *Docker-Image* namens *dashboard* erstellt.

```
$ docker build -t dashboard .
```

Listing 4.19: Befehl für die Erstellung des *Docker-Images*

Die erstellten *Docker-Images* wurden auf *DockerHub*, einem Onlinedienst zu der Veröffentlichung von *Docker-Images*, bereitgestellt[43]. Um das Dashboard mittels *Docker* zu installieren und auszuführen, muss lediglich der in Listing 4.20 dargestellte Befehl ausgeführt werden.

```
$ docker run -it -p 3000:3000 tmewes/jqa-dashboard:dashboard
```

Listing 4.20: Befehl für die Installation des Dashboards mittels *Docker*

4.3.2 Installation und Wartung

Um das Dashboard zu installieren, muss zunächst wie bereits beschrieben *Graph App Kit* installiert werden. Anschließend werden mittels des Befehls `npm install` alle benötigten Abhängigkeiten nachinstalliert, um das Dashboard starten zu können. In der Datei *package.json* im Hauptverzeichnis des Dashboards wurden die benötigten Befehle zu einem zusammengefasst, sodass ausschließlich der Befehl `npm run install-dashboard` ausgeführt werden muss, um das Dashboard zu installieren. Damit soll das Qualitätsziel der Installierbarkeit weiter unterstützt werden.
Für die Aktualisierung der abhängigen Pakete des Dashboards muss die in Listing 4.21 dargestellte Reihe von Befehlen ausgeführt werden, um die weitere Funktion des Dashboards zu gewährleisten.

[43] `https://hub.docker.com/r/tmewes/jqa-dashboard/tags/`, Zugriff am: 21.08.2018

```
// Step 1/5: Get Graph App Kit to prevent uninstall errors
$ npm install graph-app-kit --registry https://neo.jfrog.io/neo/api/npm
// Step 2/5: Install all packages
$ npm install
// Step 3/5: Uninstall Graph App Kit to enable npm update
$ npm uninstall graph-app-kit
// Step 4/5: Update packages
$ npm update
// Step 5/5: Reinstall Graph App Kit
$ npm install graph-app-kit --registry https://neo.jfrog.io/neo/api/npm
```

Listing 4.21: Benötigte Befehle für die Aktualisierung des Dashboards

Um das Qualitätsziel der Wartbarkeit weiter zu unterstützen, wurde diese Reihe von Befehlen ebenfalls zu einem Befehl zusammengefasst. Für die Aktualisierung des Dashboards muss nun ausschließlich der Befehl `npm run update-dashboard` ausgeführt werden.

5 Evaluation

In diesem Kapitel wird das prototypisch implementierte Dashboard evaluiert. Um die Funktion des Dashboards sicherzustellen, wird die Evaluationsmethode einer Fallstudie anhand von Open-Source-Projekten verwendet. Diese Methode wird in Müller und Zeckzer [2015] genauer beschrieben und darin als *case study (example)* bezeichnet. Das Dashboard wird für die Evaluation nacheinander mit verschiedenen Beispieldaten befüllt und jeweils auf seine Funktion überprüft. Bei den Beispieldaten handelt es sich um die zwei Open-Source-Projekte *Spring Petclinic*[44] sowie *jUnit*[45].
Für die Evaluation des Dashboards wurde für *Spring Petclinic* sowie *jUnit* jeweils ein *Docker-Image* angelegt, sodass die zugehörige *Neo4j*-Graphdatenbank automatisch installiert und mit Daten befüllt werden kann, um das Dashboard zu testen. Dazu wurde im Hauptverzeichnis des Dashboards jeweils ein Unterverzeichnis `data` sowie darin die Datei `Dockerfile` angelegt. In Listing 5.1 ist der Inhalt dieser Datei beispielhaft für das Softwareprojekt *Spring Petclinic* dargestellt.

```
1  FROM neo4j:latest
2
3  ENV NEO4J_PASSWD neo4j
4  ENV NEO4J_AUTH neo4j/${NEO4J_PASSWD}
5
6  COPY petclinic.dump /var/lib/neo4j/import/
7
8  VOLUME /data
9
10 CMD bin/neo4j-admin set-initial-password ${NEO4J_PASSWD} || true && \
11 bin/neo4j-admin load --from=/var/lib/neo4j/import/petclinic.dump --force && \
12 bin/neo4j start && sleep 5 && \
13 tail -f logs/neo4j.log
```

Listing 5.1: Dockerfile für die Evaluation des Dashboards am Beispiel von *Spring Petclinic*

Die zugehörigen *Docker-Images* wurden auf *DockerHub* bereitgestellt[46].
Um die *Neo4j*-Graphdatenbank mit den eingelesenen Daten des Softwareprojekts *Spring Petclinic* auszuführen, muss lediglich der in Listing 5.2 dargestellte Befehl ausgeführt werden.

```
1  $ docker run -it -p 7474:7474 -p 7687:7687 tmewes/jqa-dashboard:neo4j-petclinic
```

Listing 5.2: Befehl für die Installation der vorbefüllten *Neo4j*-Graphdatenbank mittels *Docker*

Die Daten von *jUnit* können analog durch den Austausch von `petclinic` durch `junit` innerhalb des Befehls eingelesen werden.

44 `https://github.com/spring-projects/spring-petclinic`, Zugriff am: 16.01.2018
45 `https://github.com/junit-team/junit4`, Zugriff am: 16.01.2018
46 `https://hub.docker.com/r/tmewes/jqa-dashboard/tags`, Zugriff am: 01.09.2018

Die Evaluation ist erfolgreich, wenn die Visualisierungskomponenten des Dashboards sowohl mit den eingelesenen Daten von *Spring Petclinic* als auch von *jUnit* fehlerfrei dargestellt werden. Zunächst werden die Daten des Softwareprojekts *Spring Petclinic* in die *Neo4j*-Graphdatenbank geladen und das Dashboard gestartet. In Abbildung 5.1 werden am Beispiel der Startseite des Dashboards die Daten des Softwareprojekts *Spring Petclinic* dargestellt.

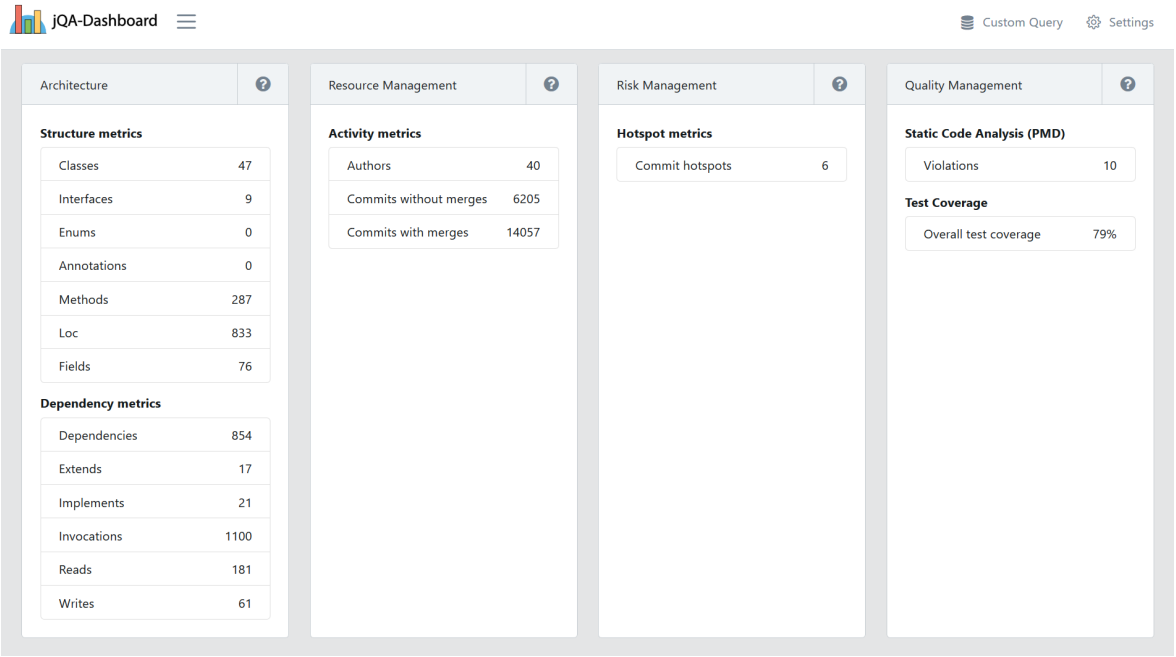

Abbildung 5.1: Startseite des Dashboards mit den eingelesenen Daten von *Spring Petclinic*

Die implementierten Visualisierungskomponenten des Dashboards wurden anschließend auf ihre Funktion sowie fehlerfreie Darstellung überprüft. Alle Visualisierungskomponenten des Dashboards konnten mit den eingelesenen Daten von *Spring Petclinic* fehlerfrei dargestellt und erfolgreich auf ihre Funktion überprüft werden. Daher wird nun die *Neo4j*-Graphdatenbank gestoppt, die Daten des Softwareprojekts *jUnit* eingespielt und die *Neo4j*-Graphdatenbank danach wieder gestartet. In Abbildung 5.2 werden erneut am Beispiel der Startseite des Dashboards die Daten des Softwareprojekts *jUnit* dargestellt.

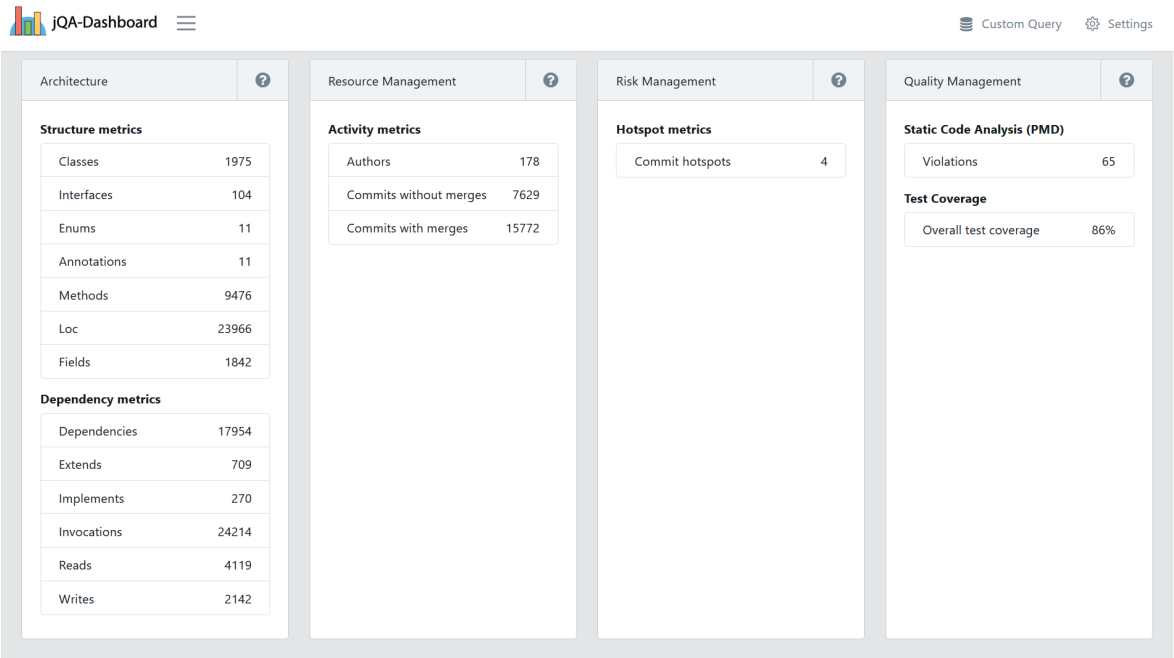

Abbildung 5.2: Startseite des Dashboards mit den eingelesenen Daten von *jUnit*

Analog zu *Spring Petclinic* wurden die implementierten Visualisierungskomponenten des Dashboards anschließend ebenfalls auf ihre Funktion sowie fehlerfreie Darstellung überprüft. Alle Visualisierungskomponenten des Dashboards konnten mit den eingelesenen Daten von *jUnit* fehlerfrei dargestellt und erfolgreich auf ihre Funktion überprüft werden. Damit wurde das Dashboard erfolgreich evaluiert.

6 Fazit und Ausblick

In diesem letzten Kapitel wird zunächst beschrieben, wie das Dashboard unter Beachtung der Qualitätsziele umgesetzt wurde. Darauf aufbauend wird ein Fazit gezogen sowie abschließend ein Ausblick gegeben, wie das Dashboard noch erweitert werden könnte.
Einen zusammenfassenden Überblick der gesetzten Qualitätsziele sowie deren jeweilige Umsetzung bietet Tabelle 6.1.

Qualitätsziel	**Umsetzung**
Erweiterbarkeit	*React* *arc42* *Travis CI* *Codecov*
Installierbarkeit	*Node.js* *Docker*
Gebrauchstauglichkeit	Literaturrecherche zu typischen Aufgaben von Projektleitern Analyse existierender Dashboard-Werkzeuge *Bootstrap*
Wartbarkeit	*Node.js* *arc42* *Codecov* *Prettier*

Tabelle 6.1: Gegenüberstellung der Qualitätsziele und deren jeweiliger Umsetzung

Nachfolgend wird die Umsetzung der einzelnen Qualitätsziele näher beschrieben. Die Erweiterbarkeit des Dashboards ist durch die Verwendung von wiederverwendbaren und leicht erweiterbaren *React*-Komponenten gegeben. Außerdem ist durch die Verwendung von *arc42* und der damit einhergehenden Dokumentation das Dashboard leicht erweiterbar. Bei jeder neuen Anpassung des Dashboards werden mittels *Travis CI* alle Tests ausgeführt und die Testabdeckung mithilfe von *Codecov* analysiert, was die Erweiterbarkeit ebenfalls begünstigt. Die Installierbarkeit wird durch *Node.js* unterstützt, welches mit *NPM* eine eigene Paketverwaltung mitbringt. Außerdem wurde ein *Docker-Image* für das Dashboard angelegt, sodass dieses einfach mit einem einzigen *Docker*-Befehl installiert werden kann.
Das Qualitätsziel der Gebrauchstauglichkeit wurde durch die durchgeführte Literaturrecherche unterstützt, da diese die Aufgabenbereiche für Projektleiter lieferte. Durch die Analyse

existierender Dashboard-Werkzeuge konnten die vom Dashboard unterstützten Aufgabenbereiche abgeleitet und umgesetzt werden, um die Gebrauchstauglichkeit sicherzustellen. Zudem wurde *Bootstrap* in das Dashboard integriert, um die Nutzererfahrung zu verbessern. Die Wartbarkeit wird durch die einfache Aktualisierungsmöglichkeit von *NPM* begünstigt. Außerdem können die Dokumentation des Dashboards mittels *arc42* und die Analyse der Testabdeckung mithilfe von Codecov ebenfalls dabei helfen, das Dashboard zu warten. Abschließend wurde das Werkzeug *Prettier* in das Dashboard integriert, welches den Quellcode des Dashboards automatisch anhand vordefinierter Regeln formatiert. Damit wird ein einheitlicher Quellcodestil sichergestellt und die Wartbarkeit des Dashboards vereinfacht.
Wie beschrieben konnten die definierten Ziele der Arbeit erfolgreich umgesetzt werden. Das konzipierte und prototypisch implementierte Dashboard erfüllt zudem die festgelegten Qualitätsziele nach ISO 25010. Teile der Arbeit konnten bereits veröffentlicht werden [Müller et al. 2018]. Dabei handelt es sich um eine wissenschaftliche Publikation mit dem Ziel, eine einheitliche Datenquelle zur Softwareanalyse und -visualisierung durch einen erweiterbaren Stapel von Open-Source-Software zu erzeugen.
Abschließend folgen im Rahmen des Ausblicks nun noch einige Ansätze, wie das Dashboard zukünftig erweitert werden könnte.
Die Visualisierungskomponente der Dateitypen könnte um eine zusätzliche Sortierung nach *Lines of Code (LOC)* erweitert werden. Dazu müsste allerdings erst *jQAssistant* dahingehend angepasst werden, dass die *LOC* je Dateityp erfasst werden können. Außerdem könnten diverse Funktionserweiterungen des Dashboards durch weitere *jQAssistant*-Scanner-Plugins umgesetzt werden. So könnte man beispielsweise die *GitHub-Issues* des Softwareprojekts in das Dashboard integrieren. Zudem wäre die Analyse von Informationen zur Laufzeit des Softwareprojekts denkbar. Diese Analyse könnte mittels der Auswertung von Log-Dateien des Monitoring-Werkzeugs *Kieker*[47] umgesetzt werden. Derzeit können ausschließlich in *Java* geschriebene Softwareprojekte eingelesen werden. Mithilfe von weiteren *jQAssistant*-Scanner-Plugins könnten auch weitere Programmiersprachen wie *PHP* unterstützt werden. Zu allen drei genannten *jQAssistant*-Scanner-Plugins existieren bereits erste Ansätze für deren Umsetzung[48, 49, 50]. Zudem wäre der Ausbau des Dashboards zu einer *Progressive Web App* möglich. Konkret könnten *Service Worker* entwickelt werden, welche das Zwischenspeichern der ausgegebenen Daten des Dashboards verbessern könnten und das Dashboard somit noch schneller machen. Außerdem könnte die Erstellung von neuen Visualisierungskomponenten vereinfacht werden. Vorstellbar wäre, dass neue Visualisierungen direkt in einem Formular innerhalb des Dashboards erstellt werden könnten. In diesem Formular müssten lediglich die Art der Visualisierung, also beispielsweise ein Kreisdiagramm, sowie die zugehörige *Cypher*-Abfrage angegeben werden. Aus diesen Informationen würde die Visualisierung dann automatisch erstellt werden.

[47] `https://kieker-monitoring.net/framework`, Zugriff am: 21.08.2018
[48] `https://github.com/softvis-research/jqa-githubissues-plugin`, Zugriff am: 21.08.2018
[49] `https://github.com/softvis-research/jqa-php-plugin`, Zugriff am: 21.08.2018
[50] `https://github.com/softvis-research/jqa-kieker-plugin`, Zugriff am: 21.08.2018

Anhang

A jQAssistant-Konzeptregeln

Listing A.1 zeigt die *jQAssistant*-Konzeptregeln des Dashboards, welche auf *GitHub* bereitgestellt wurden[51].

```
[[dashboard]]
[role=group, includesConcepts="dashboard:Authors, dashboard:Merge, dashboard:Timetree,dashboard:
     TypeHasSourceGitFile, dashboard:Filetype"]
== Dashboard

These concepts prepares the data for the dashboard.

=== Concepts

The root package of the application is "org.springframework.samples.petclinic".

[[dashboard:Authors]]
[source,cypher,role=concept]
.Removes duplicate authors.
----
MATCH
  (a:Author)
WITH
  a.name as name, collect(a) as authors
WITH
  head(authors) as author, tail(authors) as duplicates
UNWIND
  duplicates as duplicate
MATCH
  (duplicate)-[:COMMITTED]->(c:Commit)
MERGE
  (author)-[:COMMITTED]->(c)
DETACH DELETE
  duplicate
RETURN
  author.name, count(duplicate)
----

[[dashboard:Merge]]
[source,cypher,role="concept"]
.Labels merge commits.
----
MATCH
  (c:Commit)-[:HAS_PARENT]->(p:Commit)
WITH
  c, count(p) as parents
WHERE
```

[51] https://github.com/jqassistant-demo/junit4/blob/jqassistant/vissoft-2018/jqassistant/dashboard.adoc, Zugriff am: 23.09.2018

```
  parents > 1
SET
  c:Merge
RETURN
  count(c)
----
[[dashboard:Timetree]]
[source,cypher,role="concept"]
.Creates a time tree for commits.
----
MATCH
  (c:Commit)
WITH
  c, split(c.date, "-") as parts
MERGE
  (y:Year{year:parts[0]})
MERGE
  (m:Month{month:parts[1]})-[:OF_YEAR]->(y)
MERGE
  (d:Day{day:parts[2]})-[:OF_MONTH]->(m)
MERGE
  (c)-[:OF_DAY]->(d)
RETURN
  y, m, d
----

[[dashboard:GitFileName]]
[source,cypher,role="concept",verify="aggregation"]
.Copies the relativePath property of ‘:Git:File‘ nodes to the indexed property ‘fileName‘ for faster lookup.
----
MATCH
  (f:Git:File)
SET
  f.fileName = f.relativePath
RETURN
  count(f)
----

[[dashboard:TypeHasSourceGitFile]]
[source,cypher,role="concept",requiresConcepts="dashboard:GitFileName"]
.Creates relates HAS_SOURCE between types and git files.
----
MATCH
  (p:Package)-[:CONTAINS]->(t:Type)
WITH
  t, p.fileName + "/" + t.sourceFileName as sourceFileName // e.g. "/org/dukecon/model/Location.java"
MATCH
  (f:Git:File)
WHERE
  f.fileName ends with sourceFileName
MERGE
  (t)-[:HAS_SOURCE]->(f)
RETURN
  f.fileName, collect(t.fqn)
----
[[dashboard:Filetype]]
[source,cypher,role="concept"]
.Sets file types.
----
MATCH
```

```
102   (f:Git:File)
103 WITH
104   f, split(f.relativePath, ".") as splittedFileName
105 SET
106   f.type = splittedFileName[size(splittedFileName)-1]
107 RETURN
108   f.type as filetype, count(f) as files
```

Listing A.1: *jQAssistant*-Konzeptregeln für die Datenvorverarbeitung

B Dateistruktur

Abbildung B.1 zeigt die Dateistruktur des Dashboards.

Dashboard-Wurzelverzeichnis
- `.dockerignore` ..... Definition von Ressourcen, die von *Docker* ignoriert werden
- `.editorconfig` ..... Optionale Konfigurationsdatei für die Entwicklungsumgebung
- `.env` ..... Konfiguration des Dashboard-Ports
- `.gitignore` ..... Definition von Ressourcen, die von Git ignoriert werden
- `.travis.yml` ..... Informationen für die Ausführung von *Travis CI*
- `CHANGELOG.md` ..... *CoreUI*-Änderungsprotokoll
- `CONTRIBUTING.md` ..... Informationen für die Erweiterung des Dashboards
- `CRA.md` ..... Informationen über *create-react-app*
- `Dockerfile` ..... Beschreibung für den Aufbau des Dashboard-*Docker-Image*
- `LICENSE` ..... Lizenztext *Apache License 2.0*
- `package-lock.json` ..... Vollständiger Abhängigkeitsbaum des Dashboards
- `package.json` ..... Primäre Projektkonfiguration des Dashboards
- `REACT.md` ..... Informationen über die *React*-Version von *CoreUI*
- `README.md` ..... Allgemeine Informationen über das Dashboard
- `data`
  - `junit`
    - `Dockerfile` ..... Beschreibung für den Aufbau des *jUnit-Docker-Image*
    - `junit.dump` ..... Speicherauszug für *Neo4j*-Graphdatenbank
  - `petclinic`
    - `Dockerfile` ..... Beschreibung für den Aufbau des *Petclinic-Docker-Image*
    - `petclinic.dump` ..... Speicherauszug für *Neo4j*-Graphdatenbank
- `public`
  - `favicon.ico` ..... Favicon des Dashboards
  - `index.html` ..... *HTML*-Grundgerüst des Dashboards
- `src`
  - `App.css` ..... *CoreUI-CSS*-Anweisungen
  - `App.js` ..... *CoreUI*-Wrapperkomponente
  - `App.test.js` ..... Test der *CoreUI*-Wrapperkomponente
  - `AppDispatcher.js` ..... *Flux*-Dispatcher
  - `index.css` ..... *CoreUI-CSS*-Anweisungen
  - `index.js` ..... Rendern der *App*-Komponente
  - `polyfill.js` ..... Quellcode-Baustein für die Unterstützung älterer Browser
  - `registerServiceWorker.js` ..... Vorbereitungen *Progressive Web App*
  - `routes.js` ..... Festlegung des *URL-Routing*
  - `setupTests.js` ..... Konfigurationsdatei für die Ausführung der Tests

- _nav.js ..... Festlegung der Links in der Sidebar
- api
  - models
    - CommitsPerAuthor.js ..... Abfrage und Aufbereitung der Daten
    - CommitsTimescale.js ..... Abfrage und Aufbereitung der Daten
    - Dashboard.js ..... Abfrage und Aufbereitung der Daten
    - Dependencies.js ..... Abfrage und Aufbereitung der Daten
    - FilesPerAuthor.js ..... Abfrage und Aufbereitung der Daten
    - FilesPerFiletypePerAuthor.js .... Abfrage und Aufbereitung der Daten
    - FileTypes.js ..... Abfrage und Aufbereitung der Daten
    - Hotspots.js ..... Abfrage und Aufbereitung der Daten
    - LatestCommits.js ..... Abfrage und Aufbereitung der Daten
    - PMD.js ..... Abfrage und Aufbereitung der Daten
    - TestCoverage.js ..... Abfrage und Aufbereitung der Daten
  - utils
    - CirclePacking.js ..... Hilfsfunktionen für geschachtelte Kreise
- assets
  - img
    - brand
      - favicon.ai ..... Rohdatei des Dashboard-Favicons
      - logo.ai ..... Rohdatei des Dashboard-Logos
      - logo.svg ..... Dashboard-Logo
      - sygnet.svg ..... *CoreUI*-Logo
- containers
  - index.js ..... Im- und Export der gesamten *CoreUI*-Layout-Komponente
  - DefaultLayout
    - DefaultFooter.js ..... *CoreUI*-Footer-Komponente
    - DefaultHeader.js ..... *CoreUI*-Header-Komponente
    - DefaultLayout.js ..... *CoreUI*-Layout-Komponente
    - index.js ..... Im- und Export der einzelnen *CoreUI*-Layout-Komponenten
    - package.json ..... Konfigurationsdatei des *CoreUI*-Layouts
    - __tests__
      - DefaultFooter.test.js ..... *CoreUI*-Test Footer
      - DefaultHeader.test.js ..... *CoreUI*-Test Header
- scss
  - style.css ..... *CoreUI-CSS*-Anweisungen
  - style.scss ..... *CoreUI-SCSS*-Anweisungen
  - _custom.scss ..... *SCSS*-Anweisungen des Dashboards
  - _ie-fix.scss ..... Fehlerbehebung der Darstellung im Internet Explorer
  - _variables.scss ..... Überschreibungen der *CoreUI-SCSS*-Variablen
  - vendors
    - _variables.scss ..... Überschreibung der *Bootstrap*-Variablen
    - chart.js
      - chart.css ..... Überschreibung der *chart.js*-Variablen
      - chart.scss ..... Überschreibung der *chart.js*-Variablen
- tests

```
    ├── routes.test.js ..... Test der URL-Routing-Komponente
    ├── api
    │   └── utils
    │       └── CirclePacking.test.js ..... Test der Hilfsfunktionen
    └── views
        └── Dashboard
            ├── Dashboard.test.js ..... Test der Dashboardkomponente
            ├── Architecture
            │   ├── Dependencies
            │   │   ├── Dependencies.test.js ..... Test der Abhängigkeitskomponente
            │   │   └── visualizations
            │   │       └── DependencyChord.test.js ..... Test der nivo-Komponente
            │   ├── FileTypes
            │   │   ├── FileTypes.test.js ..... Test der Dateitypenkomponente
            │   │   └── visualizations
            │   │       └── FileType.test.js ..... Test der nivo-Komponente
            │   └── Structure
            │       └── Structure.test.js ..... Test der Strukturkomponente
            ├── DynamicBreadcrumb
            │   └── DynamicBreadcrumb.test.js ... Test der Breadcrumb-Komponente
            ├── Header
            │   ├── CustomQuery.test.js ..... Test der benutzerdefinierten Abfrage
            │   └── Settings.test.js ..... Test der Einstellungsmöglichkeiten
            ├── QualityManagement
            │   ├── StaticCodeAnalysis
            │   │   └── PMD
            │   │       └── PMD.test.js ..... Test der PMD-Komponente
            │   └── TestCoverage
            │       ├── TestCoverage.test.js .... Test der Testabdeckungskomponente
            │       └── visualizations
            │           └── TestCoverageTreeMap.test.js .. Test der nivo-Komponente
            ├── ResourceManagement
            │   ├── Activity
            │   │   ├── Activity.test.js ..... Test der Aktivitätskomponente
            │   │   └── visualizations
            │   │       ├── CommitsPerAuthor.test.js ..... Test der nivo-Komponente
            │   │       ├── CommitsTimescale.test.js ..... Test der nivo-Komponente
            │   │       ├── FilesPerAuthor.test.js ..... Test der nivo-Komponente
            │   │       └── LatestCommits.test.js ..... Test der nivo-Komponente
            │   └── KnowledgeDistribution
            │       ├── KnowledgeDis.test.js .... Test Wissensverteilungskomponente
            │       └── visualizations
            │           └── FilesFiletypeAuthor.test.js .. Test der nivo-Komponente
            └── RiskManagement
                └── Hotspots
                    └── Hotspots.test.js ..... Test der Hotspot-Komponente
└── views
    ├── index.js ..... Im- und Export der Dashboard-Komponente
    └── Dashboard
```

- AbstractDashboardComponent.js ..... Schnittstelle zur Graphdatenbank
- Dashboard.js ..... Startseitenkomponente des Dashboards
- Dashboard.test.js ..... Test der Startseitenkomponente des Dashboards
- package.json ..... Konfigurationsdatei des *CoreUI*-Dashboards
- Architecture
  - Dependencies
    - Dependencies.js ..... Abhängigkeitskomponente des Dashboards
    - visualizations
      - DependencyChord.js ..... *nivo*-Komponente
  - FileTypes
    - FileTypes.js ..... Dateitypenkomponente des Dashboards
    - visualizations
      - FileType.js ..... *nivo*-Komponente
  - Structure
    - Structure.js ..... Strukturkomponente des Dashboards
    - TreebeardCustomTheme.js ..... Paketexplorer-Komponente
    - visualization
      - StructureBubble.js ..... *nivo*-Komponente
- DynamicBreadcrumb
  - DynamicBreadcrumb.js ..... Breadcrumb-Komponente des Dashboards
- Header
  - CustomQuery.js ..... Komponente für benutzerdefinierte Abfragen
  - Settings.js ..... Komponente der Einstellungsmöglichkeiten
- QualityManagement
  - StaticCodeAnalysis
    - PMD
      - PMD.js ..... PMD-Komponente des Dashboards
      - visualization
        - PmdRadar.js ..... *nivo*-Komponente
  - TestCoverage
    - TestCoverage.js ..... Testabdeckungskomponente des Dashboards
    - visualizations
      - TestCoverageTreeMap.js ..... *nivo*-Komponente
- ResourceManagement
  - Activity
    - Activity.js ..... Aktivitätskomponente des Dashboards
    - visualizations
      - CommitsPerAuthor.js ..... *nivo*-Komponente
      - CommitsTimescale.js ..... *nivo*-Komponente
      - FilesPerAuthor.js ..... *nivo*-Komponente
      - LatestCommits.js ..... *nivo*-Komponente
  - KnowledgeDistribution
    - KnowledgeDistribution.js ... Wissensverteilungskomponente des Dashboards
    - visualizations
      - FilesPerFiletypePerAuthor.js ..... *nivo*-Komponente
- RiskManagement
  - Hotspots
    - Hotspots.js ..... Hotspot-Komponente des Dashboards

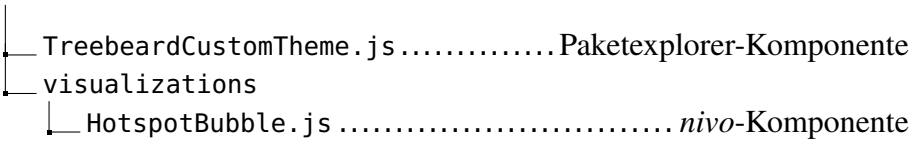

Abbildung B.1: Dateistruktur des Dashboards

Glossar

Bolt

Bolt ist ein verbindungsorientiertes Netzwerkprotokoll, welches für die Client-Server-Kommunikation in Datenbankanwendungen verwendet wird[52]. [35]

Breadcrumbs

Breadcrumbs sind Benutzungsoberflächenelemente, die entwickelt wurden, um die Navigation einfach und intuitiv zu gestalten. Sie werden von Betriebssystemen, Softwareprojekten und Webseiten verwendet. Breadcrumbs zeigen den Verzeichnispfad des aktuellen Ordners oder der aktuellen Webseite an und ermöglichen die Navigation zu übergeordneten Verzeichnissen per Mausklick.[53] [21, 46, 51]

Codecov

Codecov ist ein Werkzeug, mit dem bestimmt werden kann, welche Quellcodezeilen bei dem Durchlauf aller Tests eines Softwareprojekts ausgeführt werden[54]. [29, 52, 60, 61]

Commit

Ein Commit ist eine individuelle Änderung an einer Datei oder einem Satz von Dateien. Commits enthalten normalerweise eine Commit-Nachricht, die eine kurze Beschreibung der vorgenommenen Änderungen enthält.[55] [24, 33, 41–46, 53, 54]

Commit-Hotspots

Commit-Hotspots sind Teile des Quellcodes, die sehr häufig in den Commits eines Softwareprojekts auftauchen. [22, 34, 35, 45]

Cypher

Cypher ist eine deklarative Graph-Abfragesprache, die eine aussagekräftige und effiziente Abfrage und Aktualisierung des Graphen ermöglicht[56]. [2, 5, 23, 24, 32–34, 36, 38–45, 48, 49, 61]

D3.js

D3.js ist eine *JavaScript*-Bibliothek, welche die Bearbeitung und Visualisierung von Datensätzen im Web ermöglicht[57]. [4, 6, 22, 23, 25–29, 31]

52 `https://boltprotocol.org`, Zugriff am: 15.09.2018
53 `https://techterms.com/definition/breadcrumbs`, Zugriff am: 15.09.2018
54 `https://docs.codecov.io/docs/about-code-coverage`, Zugriff am: 15.09.2018
55 `https://help.github.com/articles/github-glossary/#commit`, Zugriff am: 15.09.2018
56 `https://neo4j.com/docs/developer-manual/current/cypher/#cypher-introduction`, Zugriff am: 15.09.2018
57 `https://d3js.org/`, Zugriff am: 15.09.2018

Dashboard

Ein Dashboard ist eine visuelle Anzeige der wichtigsten Informationen, die benötigt werden, um ein oder mehrere Ziele zu erreichen [Few 2004]. [1–4, 7–32, 34–39, 41, 43, 45–47, 49–61]

Docker

Docker ist eine Open-Source-Software für die Erstellung, Bereitstellung und Verwaltung von virtualisierten Anwendungscontainern[58]. [29, 54, 55, 60]

Docker-Container

Ein *Docker-Container* ist eine Laufzeitinstanz eines *Docker-Image*[59]. [54]

Docker-Image

Docker-Images sind die Grundlage für *Docker-Container*. Ein *Docker-Image* ist eine geordnete Sammlung von Root-Dateisystemänderungen und den entsprechenden Ausführungsparametern zur Verwendung innerhalb der Container-Laufzeit. Ein *Docker-Image* hat keinen Status und ändert sich nie.[60] [54, 55, 57, 60]

Framework

Im Software-Engineering ist ein Framework ein modernes Rahmenwerk, das dem Programmierer den Entwicklungsrahmen für seine Anwendungsprogrammierung zur Verfügung stellt und damit die Softwarearchitektur der Anwendungsprogramme bestimmt[61]. [3, 6, 25–30]

Git

Git ist eine Open-Source-Software für die Verfolgung von Änderungen in Textdateien und ist die Kerntechnologie, auf der *GitHub* aufgebaut ist[62]. [5, 15]

Git-Pull-Request

Git-Pull-Requests sind vorgeschlagene Änderungen an einem *Git-Repository*, die von einem Benutzer eingereicht und von den Mitwirkenden eines *Git-Repository* akzeptiert oder abgelehnt werden können[63]. [32]

Git-Repository

Ein *Git-Repository* enthält alle Projektdateien einschließlich der Dokumentation und speichert den Versionsverlauf jeder Datei[64]. [23, 54]

---

58 `https://searchitoperations.techtarget.com/definition/Docker`, Zugriff am: 15.09.2018
59 `https://docs.docker.com/glossary/?term=container`, Zugriff am: 15.09.2018
60 `https://docs.docker.com/glossary/?term=image`, Zugriff am: 15.09.2018
61 `https://www.itwissen.info/Framework-framework.html`, Zugriff am: 15.09.2018
62 `https://help.github.com/articles/github-glossary/#git`, Zugriff am: 15.09.2018
63 `https://help.github.com/articles/github-glossary/#pull-request`, Zugriff am: 15.09.2018
64 `https://help.github.com/articles/github-glossary/#repository`, Zugriff am: 15.09.2018

GitHub-Issues

GitHub-Issues sind vorgeschlagene Verbesserungen, Aufgaben oder Fragen im Zusammenhang mit dem *Git-Repository*[65]. [61]

Historie

Die Historie eines Softwareprojekts bezieht sich auf dessen Entwicklungsprozess [Diehl 2007, S. 3 f.]. Die Informationen bezüglich der Historie werden üblicherweise von Versionskontrollsystemen wie Git bereitgestellt. [4, 5, 8]

Hotspots

Hotspots sind komplexe Teile des Quellcodes, die sich oft ändern [Tornhill 2018, S. 19]. [4, 15, 21, 37, 45]

Jest

Jest ist ein von dem Unternehmen *Facebook* entwickeltes Werkzeug, um *JavaScript* Quellcode zu testen. Es findet vor allem Anwendung bei Softwareprojekten, welche mittels *React* realisiert werden.[66] [51]

jQAssistant

jQAssistant ist ein Werkzeug für die Qualitätsanalyse von Quellcode, mit dem projektspezifische Regeln auf struktureller Ebene definiert und validiert werden können. Es basiert auf der *Neo4j*-Graphdatenbank und kann einfach in den Build-Prozess integriert werden, um die Erkennung von Verstößen zu automatisieren und Berichte über benutzerdefinierte Konzepte und Metriken zu generieren.[67] [1, 2, 4, 5, 23, 47, 61]

Mockup

Ein Mockup ist ein großformatiges Modell von etwas, das noch nicht gebaut wurde und zeigt, wie es aussehen oder funktionieren wird[68]. [9, 10, 19, 23]

Neo4j

Neo4j ist eine in *Java* implementierte Open-Source-Graphdatenbank[69]. [2, 4, 5, 21–25, 32, 35, 36, 38, 45, 47, 50, 57, 58]

PMD

PMD ist ein Werkzeug für die statische Quellcodeanalyse von Quelltexten. Derzeit werden neben Java unter anderem JavaScript und XML sowie davon abgeleitete Dialekte unterstützt. Der Name PMD selbst hat offiziell keine ausgeschriebene Bedeutung.[70] [47]

---

65 `https://help.github.com/articles/github-glossary/#issue`, Zugriff am: 15.09.2018
66 `https://jestjs.io`, Zugriff am: 15.09.2018
67 `https://jqassistant.org/`, Zugriff am: 15.09.2018
68 `https://dictionary.cambridge.org/de/worterbuch/englisch/mock-up`, Zugriff am: 15.09.2018
69 `https://www.infoq.com/news/2008/06/neo4j`, Zugriff am: 15.09.2018
70 `https://pmd.github.io/pmd-6.8.0`, Zugriff am: 15.09.2018

Pre-Commit-Hook

Die Anweisungen der *Pre-Commit-Hook* werden automatisch ausgeführt, bevor eine Commit-Nachricht mit den entsprechenden Änderungen verknüpft wird. In der *Pre-Commit-Hook* kann beispielsweise der Quellcodestil unter Verwendung von Werkzeugen wie *Prettier* vereinheitlicht werden.[71] [54]

Prettier

Prettier ist ein Werkzeug für die automatische und konsistente Formatierung von Quellcode[72]. [29, 54, 60, 61]

Prototyping

Prototyping ist eine wissenschaftliche Vorgehensweise, bei der eine Vorabversion eines Anwendungssystems entwickelt und evaluiert wird [Wilde und Hess 2007]. [2]

React

React ist eine deklarative, effiziente und flexible *JavaScript*-Bibliothek zur Erstellung von Benutzungsoberflächen[73]. [3, 6, 28–31, 36, 43, 49–52]

Refaktorisierung

Refaktorisierung ist der Prozess, bei dem ein Softwaresystem so verändert wird, dass es das äußere Verhalten des Codes nicht verändert, aber seine interne Struktur verbessert [Fowler 2008, S. 53]. [31, 45]

Stakeholder

Als Stakeholder wird eine Person oder Gruppe bezeichnet, die ein berechtigtes Interesse am Verlauf oder Ergebnis eines Prozesses oder Projekts hat [Gessler et al. 2009, S. 71]. [1, 3, 12]

Stores

Stores sind *JavaScript*-Objekte, welche es *React*-Komponenten ermöglichen, ihren Status zu teilen[74]. [28]

Struktur

Die Struktur eines Softwareprojekts umfasst dessen Quellcode, Datenstrukturen, statische Aufrufgraphen, Beziehungen sowie Organisation [Diehl 2007, S. 3 f.]. [4]

Subversion

Subversion ist eine freie Software für die zentrale Versionsverwaltung von Dateien und Verzeichnissen[75]. [15]

---

71 `https://git-scm.com/book/uz/v2/Customizing-Git-Git-Hooks`, Zugriff am: 15.09.2018
72 `https://github.com/prettier/prettier`, Zugriff am: 15.09.2018
73 `https://reactjs.org/`, Zugriff am: 15.09.2018
74 `https://learn.co/lessons/react-stores`, Zugriff am: 15.09.2018
75 `https://subversion.apache.org/faq.html#why`, Zugriff am: 15.09.2018

Tooltip

Ein Tooltip ist ein grafisches Benutzungsoberflächenelement, welches in Verbindung mit dem Mauszeiger verwendet wird, um Informationen zu einem Element anzuzeigen, ohne darauf klicken zu müssen[76]. [31, 38–42, 45–47, 49]

Travis CI

Travis CI ist eine kontinuierliche Integrationsplattform und unterstützt den Entwicklungsprozess von Softwareprojekten, indem es Quellcodeänderungen automatisch erstellt und testet sowie unmittelbar Rückmeldung über den Erfolg der Änderungen gibt[77]. [16, 29, 53, 60]

Verhalten

Das Verhalten eines Softwareprojekts umfasst dessen Ausführung mit realen und abstrakten Daten [Diehl 2007, S. 3 f.]. [4]

Widget

Ein Widget ist eine Komponente einer grafischen Benutzungsoberfläche[78]. [14, 15]

[76] `https://www.techopedia.com/definition/5482/tooltip`, Zugriff am: 15.09.2018
[77] `https://docs.travis-ci.com/user/for-beginners/`, Zugriff am: 15.09.2018
[78] `https://dev.w3.org/2006/waf/widgets-land/`, Zugriff am: 15.09.2018

Literaturverzeichnis

Angermeier, G., 2005, Aufgabe, Definition im Projektmanagement-Glossar des Projekt Magazins. https://www.projektmagazin.de/glossarterm/aufgabe, Zugriff am: 15.08.2018 [Zitiert auf Seite 11]

Augsten, S., 2017, Was sind Docker-Container? https://www.dev-insider.de/was-sind-docker-container-a-597762/, Zugriff am: 21.08.2018 [Zitiert auf Seite 54]

Balzert, H., 2009, *Was ist Softwaretechnik?* Spektrum Akademischer Verlag, Heidelberg, ISBN 978-3-8274-2247-7, S. 17–22 [Zitiert auf Seite 1]

Bourque, F., 2014, *Guide to the software engineering body of knowledge version 3.0*. ISBN 9780769551661 [Zitiert auf Seite 10, 11, und 12]

Brath, Richard und Peters, M., 2004, Dashboard Design: Why Design is Important. *Data Mining Review* [Zitiert auf Seite 7]

Deutsch, S., 2015, Die Flux-Architektur und React. https://reactjs.de/artikel/react-flux-architektur/, Zugriff am: 12.08.2018 [Zitiert auf Seite 28]

Diehl, Stephan und Telea, A. C., 2014, *Multivariate Graphs in Software Engineering*, Bd. 8380 of *Lecture Notes in Computer Science*, Kap. 2. Springer International Publishing, Cham, ISBN 978-3-319-06793-3, S. 13–36 [Zitiert auf Seite 5]

Diehl, S., 2002, *Software Visualization, International Seminar Dagstuhl Castle, Germany, May 20-25, 2001, Revised Lectures*, Bd. 2269. Springer, ISBN 3-540-43323-6 [Zitiert auf Seite 4]

Diehl, S., 2007, *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer Berlin Heidelberg, ISBN 9783540465041 [Zitiert auf Seite 4, XX, XXI, und XXII]

Few, S., 2004, Dashboard Confusion. *Perceptual Edge*, S. 1–6, ISSN 15243621 [Zitiert auf Seite 7 und XIX]

Few, S., 2006, *Information Dashboard Design*. ISBN 0-596-10016-7 [Zitiert auf Seite 7]

Fowler, M., 2008, *Refactoring: improving the design of existing code*. Addison-Wesley object technology series, Addison-Wesley, Boston ; Munich u.a., 22. print. Aufl. [Zitiert auf Seite XXI]

Gessler, M., für Projektmanagement, D. G., Association, S. P. M., 2009, *Kompetenzbasiertes Projektmanagement (PM3): Handbuch für die Projektarbeit, Qualifizierung und Zertifizierung auf Basis der IPMA Competence Baseline Version 3.0*. Bd. 5, GPM, Deutsche Gesellschaft für Projektmanagement, ISBN 9783924841409 [Zitiert auf Seite XXI]

Kerzner, H., 2017, *Project management: a systems approach to planning, scheduling, and controlling*. 12 Aufl., ISBN 9780470278703 [Zitiert auf Seite 10 und 11]

Klocwork Inc., 2014, structure101 - Discover and define your codebase architecture. `https://www.roguewave.com/sites/rw/files/resources/klocwork-factsheet-structure101.pdf`, Zugriff am: 12.08.2018 [Zitiert auf Seite 18]

Kua, P., 2014, The Definition of a Tech Lead. `https://www.thekua.com/atwork/2014/11/the-definition-of-a-tech-lead/`, Zugriff am: 27.05.2018 [Zitiert auf Seite 11]

Maletic, Jonathan I. und Marcus, A. u. C. M. L., 2002, A task oriented view of software visualization. *Proceedings - 1st International Workshop on Visualizing Software for Understanding and Analysis, VISSOFT 2002*, S. 32–40 [Zitiert auf Seite 10, 11, und 12]

Müller, R., 2015, Software Visualization in 3D - Implementation, Evaluation, and Applicability. *Proceedings of the 6th International Conference on Information Visualization Theory and Applications (IVAPP-2015)*, S. 126 [Zitiert auf Seite 1]

Müller, R., Mahler, D., Hunger, M., Nerche, J., Harrer, M., 2018, Towards an Open Source Stack to Create a Unified Data Source for Software Analysis and Visualization. *Proceedings of the 6th IEEE Working Conference on Software Visualization* [Zitiert auf Seite 61]

Müller, R., Zeckzer, D., 2015, Past, Present, and Future of 3D Software Visualization - A Systematic Literature Analysis. *Proceedings of the 6th International Conference on Information Visualization Theory and Applications*, S. 63–74 [Zitiert auf Seite 57]

Project Management Institute, 2017, *A Guide to the Project Management Body of Knowledge (PMBOK) — Sixth Edition and Agile Practice Guide*. Project Management Institute, Newtown Square, Pennsylvania, 6 Aufl., ISBN 9781628253900 [Zitiert auf Seite 10, 11, und 13]

Singer, J., 2010, An Examination of Software Engineering Work Practices. *CASCON First Decade High Impact Papers*, S. 174–188, ISSN 0098-5589 [Zitiert auf Seite 10, 11, und 12]

Tornhill, A., 2018, *Software Design X-Rays - Fix Technical Debt with Behavioral Code Analysis*. The Pragmatic Bookshelf [Zitiert auf Seite 38, 45, und XX]

Wilde, T., Hess, T., 2007, Forschungsmethoden der Wirtschaftsinformatik: Eine empirische Untersuchung. *Wirtschaftsinformatik*, Bd. 49, S. 280–287 [Zitiert auf Seite 2 und XXI]

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich diese Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.
Darüber hinaus versichere ich, dass die elektronische Version der Masterarbeit mit der gedruckten Version übereinstimmt.

Leipzig, den 11.10.2018 Tino Mewes