TECHNISCHE UNIVERSITÄT DRESDEN

FAKULTÄT INFORMATIK
INSTITUT FÜR SOFTWARE- UND MULTIMEDIATECHNIK
PROFESSUR FÜR SOFTWARETECHNOLOGIE
PROF. DR. RER. NAT. HABIL. UWE ASSMANN

Diplomarbeit

zur Erlangung des akademischen Grades
Diplom-Informatiker

Architekturkonformitätsüberprüfung von Softwarelandschaften mittels ExplorViz

Tim Hackel
(Geboren am 8. August 1990 in Leipzig, Mat.-Nr.: 3680732)

Betreuer: M.Sc. C. Zirkelbach, Christian-Albrechts-Universität zu Kiel
Prof. Dr. U. Aßmann, Technische Universität Dresden
Prof. Dr. W. Hasselbring, Christian-Albrechts-Universität zu Kiel

Dresden, 11. September 2018

Aufgabenstellung

In dieser Arbeit soll untersucht werden, wie die Softwarearchitekturkonformitätsüberprüfung in ExplorViz umgesetzt werden kann.

Die einzelnen Teilaufgaben umfassen:

- Verschaffung einer grundlegenden Übersicht über:
  - vorhandene Softwarearchitekturkonfomitätsüberprüfungssoftware
  - ExplorViz
  - die Erweiterbarkeit von ExplorViz
  - Visualisierungskonzepte
  - Evaluationsmethoden von Software
- Konzeption:
  - Editor für die Modellerzeugung
  - Farbschema für die Visualisierung des Architekturkonformitätsüberprüfers
  - aufbauend auf Simolka [Sim15] Erweiterung des Architekturkonformitätsüberprüfers
- Umsetzung
  - Implementierung des Editors als Dummy-Extension-Anpassung im Backend
  - Implementierung des Editors für die Modellerzeugung als Ember-Erweiterung im Frontend
  - Implementierung des Architekturkonformitätsüberprüfers als Dummy-Extension-Anpassung im Backend, dabei soll die Backenderweiterung des Modelleditors enthalten sein
  - Implementierung des Architekturkonformitätsüberprüfers als Ember-Erweiterung im Frontend
- Evaluation
  - Erstellung eines Evaluationsplans
  - Durchführung der Evaluation mit 5 oder mehr Probanden
  - Darstellung und Diskussion der Ergebnisse

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die von mir am heutigen Tag dem Prüfungsausschuss der Fakultät Informatik eingereichte Arbeit zum Thema:

Architekturkonformitätsüberprüfung von Softwarelandschaften mittels ExplorViz

vollkommen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Dresden, den 11. September 2018

Tim Hackel

Kurzfassung

In der Softwarearchitektur verhält es sich wie auch in anderen Bereichen: Für ein Produkt wird zunächst ein Plan entworfen. Dieser kann nachfolgend umgesetzt werden. Hierbei ist es wichtig, möglichst viel Kontrolle über das Ergebnis der Umsetzung zu erlangen. Um dies zu erreichen, können Softwarearchitekturkonformitätsüberprüfer verwendet werden. Sie vergleichen den tatsächlichen mit dem angestrebten Zustand der Software. Im Rahmen der vorliegenden Arbeit wurde ein solcher Softwarearchitekturkonformitätsüberprüfer entwickelt und in ExplorViz eingebunden. Hierzu war es nötig, eine Möglichkeit zu schaffen, ein Modell einer Softwarelandschaft zu erstellen. Zu diesem Zweck wurde zunächst ein Modelleditor konzeptioniert und implementiert. Die entwickelte Software wurde anschließend evaluiert. Hier konnte gezeigt werden, dass sowohl der Modelleditor als auch der Softwarearchitekturkonformitätsüberprüfer erfolgreich implementiert wurden. Somit wurde mit dieser Arbeit eine Software bereitgestellt, die es lohnt auch zukünftig zu verwenden und auszubauen.

Abstract

A softwarearchitect faces the same obstacles as any other architect: At first there is a plan for a product. This plan shall alter be conducted. Within this process it is of upmost importance to secure accordance of plan and product. For this purpose softwarearchitectureconformancecheckers can be used as they compare the planed and the actual existing softwarestructure. For the present thesis a softwarearchitectureconformancechecker was built and implemented within ExplorViz. To achieve this, it was neccessary to be able to to design a modell of a softwarelandscape. Therefore a modeleditor was conceptualised and implemented. Afterwards the software was evaluated. Results show that the implementation can be seen as successful, which leads to the conclusion that the software is worth to be used and extended in the future.

Inhaltsverzeichnis

1 Einleitung **3**
1.1 Motivation . . . 3
1.2 Ziele . . . 3
1.3 Aufbau . . . 4

2 Verwandte Arbeiten **5**

3 Grundlagen **9**
3.1 Architecture Conformance Checking . . . 9
3.2 ExplorViz . . . 9
3.2.1 Daten des Backends . . . 11
3.2.2 Representational State Transfer API . . . 11
3.2.3 Rendering . . . 11
3.3 Live Architecture Conformance Checking in ExplorViz - Simolka 2015 . . . 12
3.4 Visualisierungsgrundlagen . . . 12

4 Ansatz **13**
4.1 Überblick . . . 13
4.2 Modelleditor . . . 14
4.2.1 Modelleditor - Backendansatz . . . 14
4.2.2 Modelleditor - Frontendansatz . . . 14
4.3 Architekturkonformitätsüberprüfung . . . 15
4.3.1 Architekturkonformitätsüberprüfung - Backendansatz . . . 19
4.3.2 Architekturkonformitätsüberprüfung - Frontendansatz . . . 21

5 Umsetzung **23**
5.1 Überblick . . . 23
5.2 Modelleditor . . . 24
5.2.1 Modelleditor - Backendumsetzung . . . 24
5.2.2 Modelleditor - Frontendumsetzung . . . 24
5.3 Architekturkonformitätsüberprüfer . . . 29
5.3.1 Architekturkonformitätsüberprüfer - Backendumsetzung . . . 29
5.3.2 Architekturkonformitätsüberprüfer - Frontendumsetzung . . . 31

6 Evaluation **33**
6.1 Ziel . . . 33
6.2 Methodik . . . 33

6.3 Experiment . . . 33
6.3.1 Fragebogenstruktur . . . 33
6.3.2 Experimentaufbau . . . 34
6.3.3 Experimentdurchführung . . . 34
6.3.4 Ergebnisse des Experiments . . . 35
6.3.4.1 Ergebnisse des qualitativen Tests des Modelleditors . . . 35
6.3.4.2 Ergebnisse des schriftlichen Interviews den Modelleditor betreffend . . . 38
6.3.4.3 Ergebnisse der ArchConfCheckEvaluation . . . 38
6.3.5 Diskussion der Ergebnisse des Experiments . . . 39
6.3.5.1 Diskussion des qualitativen Tests des Modelleditors . . . 39
6.3.5.2 Diskussion des schriftlichen Interviews den Modelleditor betreffend . . . 40
6.3.5.3 Diskussion der ArchConfCheckEvaluation . . . 41
6.3.6 Einschränkungen der Validität . . . 41
6.4 Zusammenfassung der Evaluation . . . 41

7 Fazit **43**

8 Ausblick **45**

Literaturverzeichnis **47**

Abbildungsverzeichnis **49**

Tabellenverzeichnis **51**

A Backend Datenmodell **53**

B Bilder des Ansatzes im Großformat **55**

C Evaluationsbogen **61**
C.1 Testvorbereitung: . . . 61
C.1.1 Proband: . . . 61
C.1.2 ExplorViz Vorführung: . . . 61
C.1.3 Einführung in den ModelEditor: . . . 61
C.2 Testausführung: . . . 62
C.2.1 Qualitativer Test Modelleditor: . . . 62
C.2.2 Interview: . . . 63
C.2.3 Erklärung des ArchConfCheckers . . . 64
C.2.4 Durchführung der ArchConfCheckEvaluation: . . . 65

D Evaluationsvisualisierung **67**

E Rohdaten der Evaluation **69**

1 Einleitung

1.1 Motivation

In der Softwarearchitektur verhält es sich wie auch in anderen Bereichen: Für ein Produkt wird zunächst ein Plan entworfen. Dieser kann nachfolgend umgesetzt werden. Hierbei ist es wichtig, möglichst viel Kontrolle über das Ergebnis der Umsetzung zu erlangen. Dies kann beispielsweise durch Überwachung der Umsetzung erreicht werden. Die Kontrolle ist essentiell, da so Mängel und Defekte des Produktes vermieden werden können. Dies sei an einem Beispiel veranschaulicht: Ein Architekt entwirft entsprechend eines Kundenwunsches einen Bauplan für ein Haus. Dieser Bauplan wird anschließend von Handwerkern umgesetzt. Während dieser Umsetzung und auch final vergewissert sich der Architekt der korrekten Umsetzung. Er überwacht, ob der Bauplan, und damit auch indirekt der Wunsch des Kunden, korrekt ausgeführt werden. Dies ist wichtig, da bei fehlerhafter Umsetzung beispielsweise Sicherheitsgefährdungen sowie Unzufriedenheit des Kunden entstehen. Während der Kontrolle vergleicht der Architekt kontinuierlich seinen Bauplan mit dem aktuellen Zustand des Hauses. Hierbei kann dieser durchaus durch veränderte Kundenwünsche oder Umstände vom ursprünglichen Bauplan abweichen. Anschließend an den Vergleich sollte der Architekt die Möglichkeit haben, Differenzen und Übereinstimmungen zwischen dem Bauplan und dem Zustand des Hauses präzise zu kommunizieren.

Analog zu diesem Beispiel verhält es sich mit der Softwarearchitektur. Der Softwarearchitekt erstellt ein Modell der Software nach den Wünschen eines Kunden. Der Plan des Modells wird von Softwareerstellern umgesetzt. Anschließend kontrolliert der Softwarearchitekt die korrekte Umsetzung sowie Differenzen und Übereinstimmungen zwischen geplanten und tatsächlichem Zustand der Software. Dadurch sollen Sicherheitslücken in der Software entdeckt werden. Außerdem wird so sichergestellt, dass alle gewünschten und geplanten Elemente im tatsächlichen Zustand der Software enthalten sind.

1.2 Ziele

In Analogie zum in der Motivation genannten Beispiel lassen sich die Ziele der vorliegenden Arbeit ableiten.

Das erste Hauptziel dieser Arbeit besteht in dem Vergleich zweier Modelle der selbigen Software. Hierbei handelt es sich um den Ist- und den Soll-Zustand der Software.

In der Abbildung 1.1 sind die beiden betrachteten Softwarezustände ersichtlich. Der Ist-Zustand beschreibt den Teil der Software, welcher momentan implementiert ist. Der Soll-Zustand entspricht dem Modell der Software, welche vom Architekten gewünscht ist. Es ist erkennbar, dass beide Zustände eine Überschneidung aufweisen. Diese Überschneidung entspricht dem angestrebten Zustand, in welchem das Ist- und Soll-System identisch sind. Dies bedeutet, dass die real bestehende Software genau dem entspricht, was der Architekt vorgesehen hatte. Um den Vergleich zu ermöglichen, ist es zunächst nö-

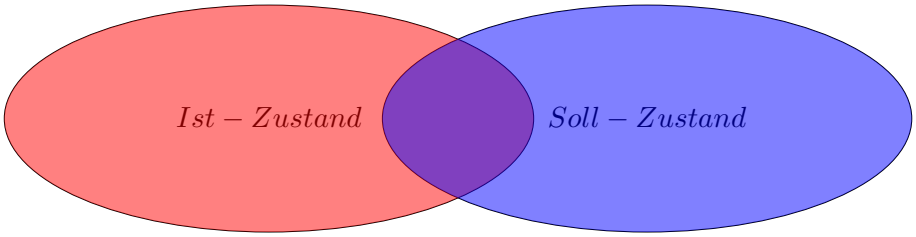

Abbildung 1.1: Venn-Diagramm der betrachteten Softwarezustände

tig, einen Modelleditor zu entwickeln. Mit Hilfe des Modelleditors kann daraufhin der Soll-Zustand der Software modelliert werden. Das Aussehen dieses Soll-Modells ist abhängig vom Benutzer und dessen individuellen Zielen und Ansprüchen. Die Entwicklung des Modelleditors ist Teilziel dieser Arbeit. Die Erstellung des Modells des Ist-Zustandes ist nicht Teilziel dieser Arbeit, da diese bereits Bestandteil von ExplorViz ist.

Beim zweiten Hauptziel dieser Arbeit handelt es sich um die visuelle Darstellung der Ergebnisse des Vergleichs der Softwarezustände. Hierbei wird die Einfärbung der Elemente und Kommunikationen je nach ihrem Zustand angestrebt. Hierdurch soll die Identifikation der Zustände optimiert werden. Außerdem soll dadurch die Kommunikation zwischen Softwarearchitekten und Softwareerstellern erleichtert werden.

Weiteres Ziel dieser Arbeit ist es, die Umsetzung mittels einer Evaluation in ihrer Güte einzuschätzen. Hierbei wird zunächst ein Evaluationsplan entwickelt. Dabei gibt der Evaluationsplan Auskunft über Durchführung und Auswertung. Diesem Plan entsprechend wird die Evaluation anschließend durchgeführt. Die Umsetzung des Evaluationsplanes ist Teil dieser Arbeit und wird im Detail im Kapitel Evaluation erläutert.

1.3 Aufbau

Die vorliegende Arbeit ist aus sechs Teilen aufgebaut. Zu allererst werden Arbeiten vorgestellt, welche verwandte Themen fokussieren (Kapitel 2). Es folgen eine Vorstellung der Grundlagen (Kapitel 3), auf welche sich diese Arbeit stützt. Anschließend wird erläutert, welche Ansätze für die Umsetzung angestrebt wurden (Kapitel 4). Daran anschließend wird die Umsetzung detailliert dargelegt (Kapitel 5). Daraufhin werden die Methode, Durchführung und Ergebnisse der Evaluation beschrieben (Kapitel 6). Die Arbeit schließt mit einem Fazit (Kapitel 7) und einem Ausblick (Kapitel 8) ab.

2 Verwandte Arbeiten

In diesem Kapitel werden Arbeiten vorgestellt, welche verwandte Themen fokussieren. Es wird begonnen mit einer Arbeit von Pruijt et al. [PKWB17], welche den “State of the Art” im behandelten Themengebiet darstellt. Anschließend wird eine Arbeit von Fekete et al. [FVWSN08] vorgestellt, in der es um die Farbwahl im Rahmen der Visualisierung geht. Abschließend wird eine Arbeit von Salvendry et al. [Sal12] präsentiert, welche sich mit der Evaluation von Software beschäftigt.

Bei der Arbeit von Pruijt et al. handelt es sich um eine Darlegung des aktuellen Forschungsstandes im Bereich der Softwarearchitekturkonformitätsüberprüfung. Es wurden insgesamt zehn Softwarearchitekturkonformitätsüberprüfer auf die Richtigkeit und Genauigkeit ihrer Aussagen getestet.

Table IV. Benchmark test—detection of direct dependencies (0 = not detected; 1 = detected).

Dependency type	ConQAT	Dependometer	dTangler	JITTAC	Lattix	Macker	SAVE	Sonar ARE	Sonargraph	Structure101	
Import											
Class import	0	0	0	1	0	0	1	0	0	0	2
Declaration											
Instance variable	0	1	1	1	1	1	1	1	1	1	9
Class variable	0	1	1	1	1	1	1	1	1	1	9
Local variable, initialized	0	1	0	1	1	0	1	0	1	1	6
Parameter	0	1	1	1	1	1	1	1	1	1	9
Return type	0	1	1	1	1	1	0	1	1	1	8
Exception	1	1	1	1	1	1	1	1	1	1	10
Type cast	1	1	1	1	1	1	0	1	1	1	9
Call											
Instance method	1	1	1	1	1	1	1	1	1	1	10
Instance method, inherited	1	1	1	1	1	1	1	1	1	1	10
Class method	1	1	1	1	1	1	1	1	1	1	10
Constructor	1	1	1	1	1	1	1	1	1	1	10
Inner class method (instance)	1	1	1	1	1	1	0	1	1	1	9
Interface method	1	1	1	1	1	1	1	1	1	1	10
Library class method	1	1	1	1	1	1	1	1	1	1	10
Access											
Instance variable (read, write)	1	1	1	1	1	1	0	1	1	1	9
Instance variable, inherited	1	1	1	1	1	1	0	1	1	1	9
Class variable	1	1	1	1	1	1	0	1	1	1	9
Constant variable	0	1	0	1	0	0	0	0	1	0	3
Enumeration	1	1	1	1	1	1	0	1	1	1	9
Object reference, param. value	0	1	0	0	0	0	0	0	1	1	3
Inheritance											
Extends class	1	1	1	1	1	1	1	1	1	1	10
Extends abstract class	1	1	1	1	1	1	1	1	1	1	10
Implements interface	1	1	1	1	1	1	1	1	1	1	10
Annotation											
Class annotation	0	1	0	1	1	0	0	0	1	1	5
Detected (out of 25)	16	24	20	24	22	20	15	20	24	23	
Sensitivity (in %) (average = 83)	64	96	80	96	88	80	60	80	96	92	

Abbildung 2.1: Ergebnisse der Detektierung von direkten Abhängigkeiten (Vgl:[PKWB17])

Beide Tabellen 2.1 und 2.2 lesen sich wie folgt. In der Kopfzeile befinden sich zehn verglichenen Softwarearchitekturkonformitätsüberprüfer. In der ersten Spalte sind die verschiedenen architekturverletzenden Abhängigkeitstypen aufgelistet. Sollte ein Softwarearchitekturkonformitätsüberprüfer eine architekturverletzende Abhängigkeit nicht detektiert haben, wird dies mit “0” notiert, anderenfalls ist eine “1” notiert. In der letzten Spalte ist ersichtlich wie viele Softwarearchitekturkonformitätsüberprüfer die jeweilige architekturverletzende Abhängigkeit erkannten. Den untersten beiden Zeilen lässt sich die absolute und relative Detektierungsrate eines jeden Tools entnehmen. Die relative Detektierungsrate entspricht

Table V. Benchmark test—detection of indirect dependencies (0 = not detected; 1 = detected).

Dependency type	ConQAT	Dependometer	dTangler	JITTAC	Lattix	Macker	SAVE	Sonar ARE	Sonargraph	Structure101	
Call											
Instance method	1	1	1	1	1	1	1	1	1	1	10
Instance method, inherited	0	0	0	1	0	0	1	0	0	1	3
Class method	1	1	1	1	1	1	1	1	1	1	10
Access											
Instance variable	1	1	1	1	1	1	0	1	1	1	9
Instance variable, inherited	0	0	0	1	0	0	0	0	0	1	2
Class variable	1	1	1	1	1	1	0	1	1	1	9
Object reference—reference var.	1	1	1	1	1	1	0	1	1	1	9
Object reference—return value	0	1	0	0	0	0	0	0	0	1	2
Inheritance											
Extends—implements variations	0	0	0	0	0	0	0	0	0	0	0
Detected (out of 9)	5	6	5	7	5	5	3	5	5	8	
Sensitivity (in %) (average = 60)	56	67	56	78	56	56	33	56	56	89	

Abbildung 2.2: Ergebnisse der Detektierung von indirekten Abhängigkeiten(Vgl:[PKWB17])

der Sensitivität. In den aus [PKWB17] entnommenen Tabellen ist ersichtlich, dass kein Tool dazu in der Lage war, alle Architekturverletzungen aufzudecken. Es gibt jedoch Architekturverletzungen, welche von allen zehn Tools aufgedeckt wurden. Es wird deutlich, dass große Unterschiede in der Genauigkeit von verschiedenen Tools bestehen. Während JITTAC und Sonograph 96% aller direkten architekturverletzenden Abhängigkeiten detektierten, konnten durch SAVE lediglich 60% aufgezeigt werden. Ähnlich verhielt es sich bei der Detektion von indirekten architekturverletzenden Abhängigkeiten. Hierbei konnten durch Structure101 89% und durch SAVE 33% der Architekturverletzungen aufgezeigt werden. Die Ergebnisse zeigen auf, dass insgesamt 77% aller Softwarearchitekturverletzungen durch die zehn Tools aufgedeckt werden konnten. Während der Testung der Tools kam es in keinem Fall zu einer Deklaration eines architekturkonformen Elements als architekturverletzend. Pruijt et al. konkludieren, dass ein höherer Grad an Genauigkeit bei allen zehn Tools wünschenswert wäre.

Wie Fekete et al. bereits 2008 in ihrer Arbeit herausstellten, eignen sich farbliche Unterschiede dazu, bereits präkognititiv die Aufmerksamkeit auf bestimmte Bereiche der Visualisierung zu lenken. So werden frühzeitig Points of Interest innerhalb des Diagramms herausgearbeitet. Daran folgend kann der Nutzer genau diese Bereiche im Detail betrachten. Diese visuellen Reize führen zu schnellerer und intuitiverer Verarbeitung von Informationen [FVWSN08]. Fekete et al. beschrieben weiterhin Unterschiede zwischen verschiedenen Farben und deren Effekte. Dabei stellten sie heraus, dass Rot als Signalfarbe von erhöhter Aufmerksamkeitserregung profitiert. Aus diesem Grund wird in der vorliegenden Arbeit die Farbe Rot dazu genutzt, die Aufmerksamkeit auf die Bereiche zu lenken, welche von erhöhter Wichtigkeit sind. Im konkreten Fall sind das Elemente der Softwarelandschaften, die potentielle Eindringlinge darstellen, weil sie im Ist-Zustand vorhanden sind, aber nicht modelliert wurden. Bei der Visualisierung ist darauf zu achten, dass Farben nicht für alle Menschen gleichgut unterscheidbar sind. Dafür gibt es spezielle Farbpaletten, welche auf den Websites von Colorbrewer und Visolve zur Verfügung gestellt werden [BH09, Sol13].

Ein ebenfalls wichtiger Bestandteil der vorliegenden Arbeit ist die Evaluation der Implementierung. Salvendry hat 2012 in einem Handbuch für Usability Studien eine Vielzahl von Studien zum Thema Evaluation von Software vereint und sich dabei auf den Faktor Mensch in Experimenten fokussiert. Er konkludierte, dass es mindestens 31 Faktoren im Zusammenhang zwischen Menschen und Experimenten gebe, welche alle Einfluss auf die Ergebnisse nehmen können ohne, dass das Experiment verändert werde. Hierzu zählen beispielsweise individuelle Heuristiken, Müdigkeit sowie vorausgegangene Erfah-

rungen der Probanden [Sal12]. Ziel einer jeden Evaluation sollte es deshalb sein, möglichst viele dieser Faktoren zu minimieren ,um so ein möglichst aussagekräftiges Ergebnis zu erhalten. Diese Aufgabe wurde von Lewis et al 2014 aufgegriffen. Er fokussierte seine Arbeit auf das Erstellen von mehreren Fragekatalogen. Diese Fragekataloge zielen darauf ab, bei der Erstellung von Evaluationen eine Leitlinie zu bieten, an welcher sich orientiert werden kann. Dabei gelang es ihm, einen Fragenkatalog zum Thema Softwareusability Studie herauszuarbeiten, welcher für die vorliegende Arbeit als Grundlage dienen wird, um die erstellte Benutzerstudie zu evaluieren [Lew14].

3 Grundlagen

3.1 Architecture Conformance Checking

Architecture Conformance Checking ist die Bezeichnung für die Überprüfung von Softwarearchitekturen auf ihre Korrektheit. Eine Softwarearchitektur ist dabei die Gesamtheit der implementierten Software und deren systematischer Bestandteile. Beim Architecture Conformance Checking werden meist visuell die Eigenschaften und Beziehungen von Systemkomponenten zueinander dargestellt. Die fundamentale Frage einer Architecture Conformance Checking Anfrage ist dabei, ob ein System sich der Intention des Softwarearchitekten entsprechend verhält. Dabei ist es essentiell, dass die Überprüfung nicht nur statisch, sondern auch während der Laufzeit kontinuierlich ausgeführt werden kann. Dadurch wird gewährleistet, dass Änderungen am Soll- oder auch Ist-Zustand des Systems innerhalb kurzer Zeit detektiert werden können.

3.2 ExplorViz

In der Arbeit von Fittkau et al. 2013 [FWWH13] wird ExplorViz erstmals vorgestellt. Dabei handelt es sich um eine Software die zur Darstellung von Softwarearchitekturen benutzt wird. ExplorViz erstellt dabei dynamisch eine Landschaftsansicht der zu untersuchenden Software, welche die einzelnen beteiligten Elemente und deren Kommunikationen darstellt. Die Kommunikationen (Aufrufe innerhalb der Software) werden durch Linien dargestellt. Dabei verhält sich die Dicke der Linie proportional zur Quantität der Kommunikation. Wenn man innerhalb der Landschaftsansicht eine Applikation auswählt, so öffnet sich die entsprechende Applikationsansicht, welche sich der Stadtaufbauanalogie bedient (siehe 3.1). Die Stadtaufbauanalogie bezeichnet dabei eine Visualisierungsform, wobei die grünen Kästen Stadtgebiete darstellen und die vertikalen violetten Quader Gebäuden entsprechen. Der Vorteil einer solchen metaphorischen Visualisierung liegt darin, dass Benutzer mit der Metapher der Stadt vertraut sind. Dadurch ist die Visualisierung besser zugänglich und verständlich. Laut Fittkau et al. [FWWH13, FRH15, FKH17] ergibt sich dadurch eine gute Übersichtlichkeit und eine gute Skalierbarkeit, wobei sich auf Alam et al. und Caserta et al. bezogen wird [AD07, CZB11].

ExplorViz hält auch weitere Funktionalitäten zur kollaborativen Zusammenarbeit an Software bereit. Das erste Feature beinhaltet die Visualisierung von live-Datenströmen. Hierdurch können IT-Mitarbeiter auf Grundlage von realen Daten miteinander kommunizieren und visuell greifbar darstellen, worauf sie sich beziehen. Der nächste Vorteil besteht darin, dass neuen Mitarbeitern die Einarbeitung in neue Software durch visuelle Darbietung erleichtert wird. Der letzte und neuste Funktionsumfang, der bei kollaborativen Arbeiten eine Rolle spielt, ist das kollaborative Explorieren von Software mittels ExplorViz VR. Dabei tragen mehrerer Kollaborateure VR-Brillen, welche es ihnen ermöglichen, sich frei in einem virtuellen Raum und um die Softwarelandschaft herum zu bewegen [FKH15]. Es sei erwähnt, dass in-

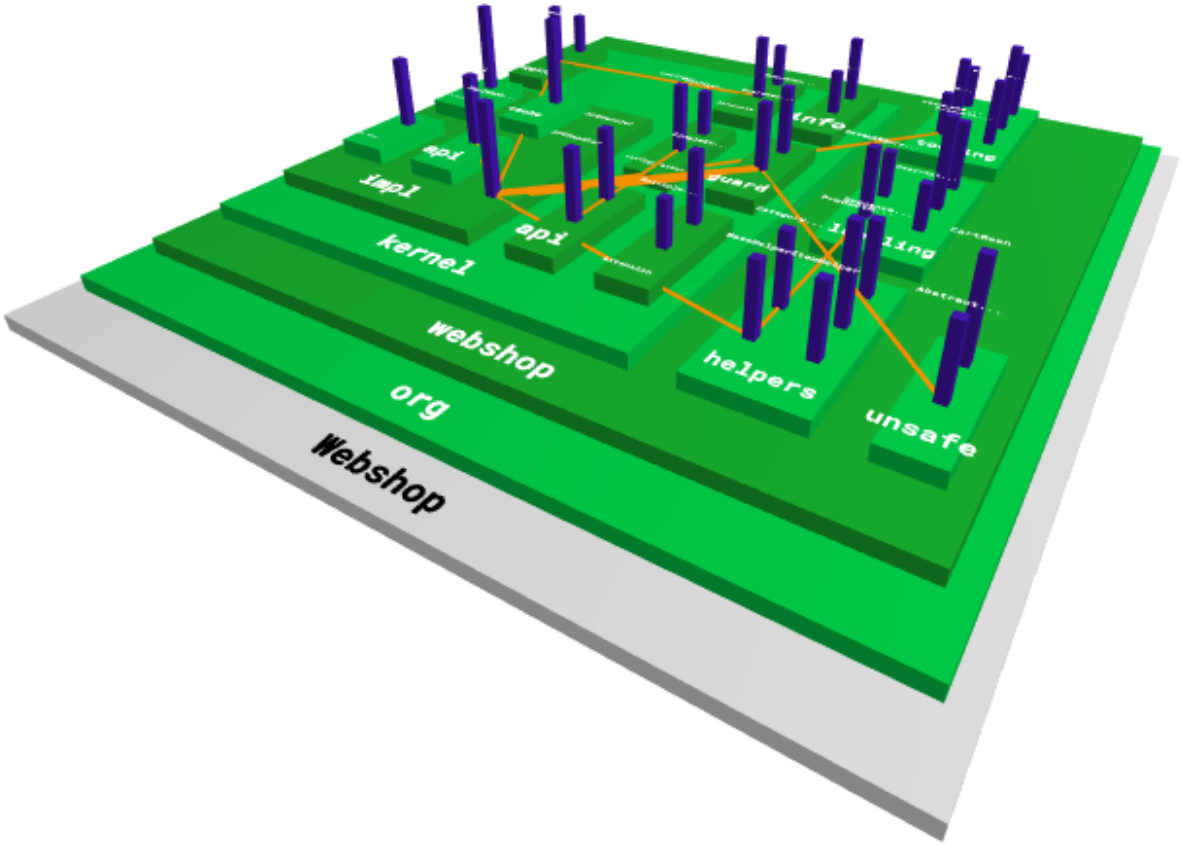

Abbildung 3.1: Applikationsansicht innerhalb von ExplorViz

nerhalb von ExplorViz bereits Softwarelevelagreementsverletzungen detektiert und visualisiert wurden. Softwarelevelagreementsverletzungen spiegeln auch eine Form der Verletzung der Architektur wider, ähnlich wie sie auch in dieser Arbeit thematisiert werden [HJZ+17]. Weiterhin ist von Bedeutung, dass die Architektur von ExplorViz vor kurzem modernisiert wurde [ZKH18]. Dabei wurde auf eine Mikroservicearchitektur umgerüstet, um Erweiterungen weitestgehend schwellenarm zu ermöglichen.

3.2.1 Daten des Backends

Das UML Diagramm zum Datenmodell des Backends befindet sich im Anhang (siehe A.1). Innerhalb des Datenmodells wird dargestellt, dass das zentrale Objekt eine Landschaft (`Landscape`) ist. Dieses enthält Systeme (`Systems`). Als Beispiel eines Systems könnte man *Standorte von IT-Gebäuden* anbringen. Systeme enthalten wiederum Knotengruppen (`NodeGroups`). Knotengruppen stellen zum Beispiel *Räume innerhalb des IT-Gebäudes* beziehungsweise *IP-Adressräume* dar. Knotengruppen enthalten Knoten (`Nodes`). Hierfür wären *Server* oder *PC*s geeignete Beispiele. Knoten enthalten wiederum Applikationen (`Applications`). Diese stellen *Anwendung*en auf dem Computer dar. Applikationen enthalten Komponenten (`Components`). Komponenten können wiederum Komponenten oder Klassen (`clazzes`) beinhalten. Komponenten sind dabei *Softwarepakete* (`packages`) und Klassen bedienen sich derselben Definition wie im Softwareumfeld üblich. Das Datenmodell im Frontend entspricht weitestgehend dem des Backends und wird daher nicht näher beleuchtet.

3.2.2 Representational State Transfer API

Als Representational State Transfer (REST) API versteht man eine API, welche zustandslos ist und eine einheitlich definierte Schnittstelle besitzt. Für die Kommunikation zwischen Front- und Backend innerhalb von ExplorViz wurde eine REST API verwendet. Es werden beim Austausch zwischen Client und Server fest definierte Zustände gehandelt. Folglich muss zu jedem Zeitpunkt dieselbe Antwort vom Server zurückgesendet werden, wenn eine Anfrage mit identischen Eingabeparametern gestellt wird. Diese Konvention ist mit Ausnahme von `latestLandscape` eingehalten worden. Das `latestLandscape` stellt jedoch einen speziellen Fall einer Anfrage des aktuellen Landscapes dar. Diesem liegt eine Anfrage einer Landschaft mit Zeitstempel (`timestamps`) zugrunde und diese erfüllt die Anforderungen einer REST API. Die auszutauschenden Zustände werden als Ressourcen bezeichnet. Diese liegen an verschiedenen Punkten innerhalb des Programmes vor. Diese Punkte werden als Routen bezeichnet. Routen können auch außerhalb der API-Anfragen mittels URL angesteuert werden. Das bedeutet, wenn man auf die ExplorViz Visualisierung zugreift, kann man im Browser auch die einzelnen Ressourcen abfragen. Dies wird aus Sicherheitsgründen durch Token und Authentifizierungsprotokolle kontrolliert. Die Übertragung der Landschaften wird in ExplorViz auf Grundlage von JSON Dateien umgesetzt. Aus Gründen der Minimierung von Übertragungskapazitäten, werden die JSON Dateien mittels BASE64 kodiert und auf der empfangenden Seite wieder dekodiert.

3.2.3 Rendering

Im Rendering-Kern (`rendering-core`) vom ExplorViz-Frontend ist bereits eine auf ThreeJS basierende Rendering Pipeline vorhanden. Es ist möglich diese an jeder beliebigen Stelle zu verändern

und zu erweitern. Auf diesen Umstand wird im Verlauf dieser Arbeit, beim Einfärben der Visualisierung, zurück gegriffen.

3.3 Live Architecture Conformance Checking in ExplorViz - Simolka 2015

Eine Grundlage der vorliegenden Arbeit bietet [Sim15]. Auch Simolka führte im Rahmen seiner Arbeit eine Architekturkonformitätsüberprüfungen durch. Diese Arbeit wurde auf einem veralteten Framework geschrieben und wird daher neu implementiert. Die Arbeit Simolkas thematisierte bereits die Einfärbung der Kommunikationslinien. Hierbei wurden bereits Farbwahlen getroffen, welche für die vorliegende Arbeit beibehalten wurden. Simolkas Analysearbeiten waren nicht vollständig ausgereift und ließen Fragen offen, deren Beantwortung das Ziel der vorliegenden Arbeit ist. Weiterhin enthält die Arbeit Simolkas im Kapitel Ausblick drei wichtige Punkte die innerhalb der vorliegenden Arbeit genauer thematisiert werden. Hierbei handelt es sich um:

1. Das Einbeziehen der Applikationsebene
2. Die Färbung der Kommunikationslinien
3. Das Erstellen von Knotengruppen

In seiner Arbeit von 2015 bezog Simolka nur die Softwarearchitektur Landschaftsansicht ein. Diese wurde nicht vollständig (wie in Punkt 2 und 3 zu sehen) umgesetzt. Applikationsansichten wurden in Simolkas Arbeit nicht berücksichtigt. Der Zweite Punkt impliziert, dass es Komplikationen bei der Einfärbung der Kommunikationslinien gab. Zum einen wurden diese teilweise übereinander gezeichnet was die Färbung der unterliegenden Linien verbarg und zum anderen wurden die Linien sehr grob gezeichnet, sodass keine eleganten Übergänge vom Knoten zur Linie gegeben waren. Ferner wurde in Simolkas Arbeit darauf verzichtet Knotengruppen in ihren beiden Ausprägungen zu betrachten. Das heißt, obwohl es in ExplorViz sowohl Knoten als auch Knotengruppen gibt, wurden von Simolka lediglich Knoten betrachtet.

3.4 Visualisierungsgrundlagen

Ein weiterer grundlegender Teilaspekt dieser Arbeit ist die Visualisierung von Unterschieden und markanten Punkten innerhalb einer Softwarelandschaft. Dabei muss die Nutzbarkeit im Vordergrund stehen. Ist das Userinterface komplex und schwer verständlich, kann daraus eine verminderte Nutzbarkeit der finalen Software einhergehen. Die Farben der Visualisierung der Architekturkonformitätsüberprüfung werden im Stil von ExplorViz gehalten, um dem Nutzer die Erweiterung so schwellenarm wie möglich darzubieten.

4 Ansatz

Das folgende Kapitel beschäftigt sich mit den inhaltlichen Ansätzen, welche verfolgt werden müssen, um eine erfolgreiche Implementierung zu gewährleisten. Der Kern der Aufgabe ist, wie bereits erwähnt, der Vergleich zweier Softwarearchitekturen. Unterschiede in Softwarearchitekturen sind bereits Gegenstand zahlreicher Arbeiten [PW92, PTV+10, KP07]. Gemeinsamer Konsens dieser Arbeiten in Bezug auf die Visualisierung ist die Verwendung farblicher Unterschiede zum Erzeugen von Aufmerksamkeit. Hierbei wird die Farbe Rot als sehr markant angesehen. Diese Farbe wird im ExplorViz Farbschema derzeit nur beim Selektieren in der Anwendungsansicht verwendet. Daher steht die Farbe Rot in der Landschaftsansicht zur Verfügung um auf fehlende oder überschüssige Komponenten hinzuweisen. Bei der Prüfung auf Massentauglichkeit wurde festgestellt, dass Visualisierungen in jedem Fall zu einer Verbesserung der Informationsaufnahme führen, doch diese sei nicht immer messbar [Pla04]. Einer der wichtigen Punkte, die Plaisant in der Arbeit ansprach, ist dabei, dass Visualisierungen für sehbeeinträchtigte Nutzer schwer fassbar sind. Die Umsetzung von anderweitigen Visualisierungen innerhalb von ExplorViz ist nicht Aufgabe dieser Arbeit, dennoch ist bei der Visualisierung darauf zu achten, dass zum Beispiel Farben nicht für alle Menschen gleichgut unterscheidbar sind. Dafür gibt es spezielle Farbpaletten, die unter anderem von Colorbrewer zur Verfügung gestellt werden [BH09].

4.1 Überblick

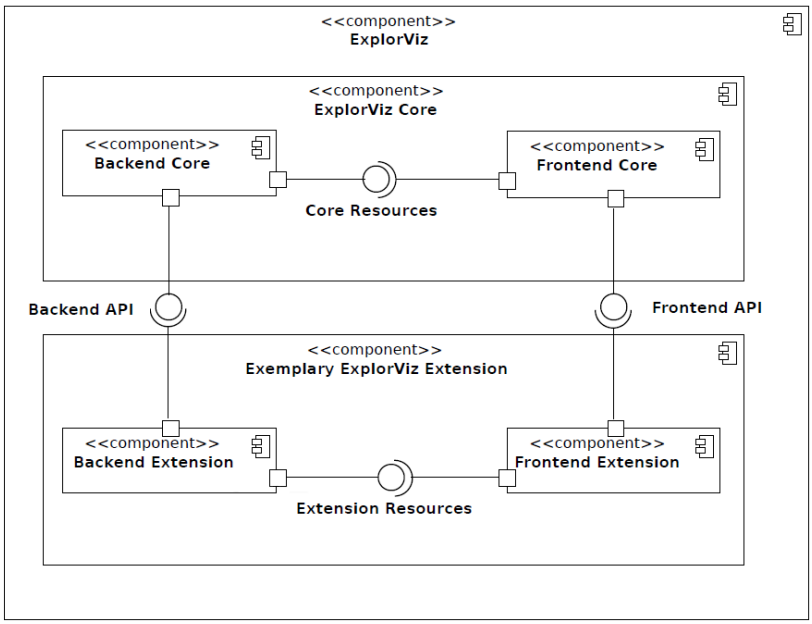

Abbildung 4.1: UML Komponentendiagram von ExplorViz (Vgl:[FKH17])

In der Abbildung 4.1 ist der grundsätzliche Aufbau von ExplorViz und seinem Erweiterungsverhalten ersichtlich. Dabei fällt auf, dass es zwei Grundbereiche gibt, welche erweitert werden können. Zum einen sehen wir links im Bild das Backend mit einem Kern und einer Erweiterung, welche über eine entsprechende Backend API an das Grundgerüst angebunden ist (siehe 3.2.2). Dabei werden zunächst neue Ressourcen angelegt, welche für die Speicherung von Landschaften und die Bereitstellung des ebenfalls im Backend durchgeführten Vergleiches von Softwarelandschaften genutzt werden. Der Vergleich der Softwarelandschaften wird durch die rekursive Untersuchung der Landschaften erfolgen. Hierbei werden den einzelnen Elementen der Softwarelandschaften Status zugeordnet, welche ihre Zustände widerspiegeln. Das Frontend, im Bild 4.1 rechts weist eine ähnliche Anbindung an den Frontendkern auf. Im Frontend wird die Kernlogik des Modelleditors umgesetzt werden. Weiterhin wird die aus dem Backend entnommene Softwarelandschaft visualisiert. Dabei wird auf die gesetzten Status eingegangen. Dies resultiert in der Einfärbung der Elemente und Kommunikationen. In den folgenden Abschnitten werden sowohl die fachlichen als auch die technischen Ansätze des Modelleditors und des Architekturkonformitätsüberprüfers thematisiert.

4.2 Modelleditor

4.2.1 Modelleditor - Backendansatz

Dem Komponentendiagramm 4.1 lässt sich entnehmen, dass die Backenderweiterung über die Backend API an den Backendkern angeschlossen wird. Die zentrale Aufgabe besteht darin, eine Möglichkeit zur Speicherung von im Frontend erstellten Landschaftsobjekten im Backend zu ermöglichen. Das heißt, es muss eine neue POST oder PATCH Ressource angelegt werden, welche Landschaftsobjekte entgegennimmt und diese persistent speichert. Zur nachhaltigen Verwendung des Editors muss eine Möglichkeit geschaffen werden, die erzeugten Modelle zu exportieren, beziehungsweise schon erstellte Modelle zur erneuten Editierung zu importieren.

4.2.2 Modelleditor - Frontendansatz

Laut Komponentendiagramm 4.1 gibt es im Frontend die Möglichkeit einer Erweiterung. Diese kann sowohl auf Frontendkernkomponenten als auch auf Backenderweiterungskomponenten zugreifen. Dafür müssen die jeweiligen Aufrufe der Ressourcen des Backends im Frontend bereitgestellt werden. Im Frontend werden außerdem leere beziehungsweise importierte Landschaftsmodellobjekte nach und nach durch Nutzereingaben mit Elementen gefüllt. Dabei muss darauf geachtet werden, dass Konventionen wie das applikationsgleiche Aussehen von Knoten beachtet werden. Eine weitere zu beachtende Konvention ist es, dass Elemente, welche gleiche parentale Elemente besitzen nicht namensgleich sein dürfen. Bei der Erstellung einer namensgleichen Komponente sollte es zur Anzeige einer Fehlermeldung kommen, damit der Nutzer auf seine Eingabe hin ein direktes visuelles Feedback bekommt.

Die typischen Funktionen eines Editors, wie das Erstellen neuer Elemente und deren Bearbeitung, sind Voraussetzung für eine gelungene Umsetzung. Wie in Abbildung 4.2 zu sehen, gibt es mehrere Bereiche, welche nachfolgend einzeln behandelt werden.

Der generelle Aufbau der Seite folgt dem vorhanden ExplorViz-Schema. Zentral befindet sich ein Can-

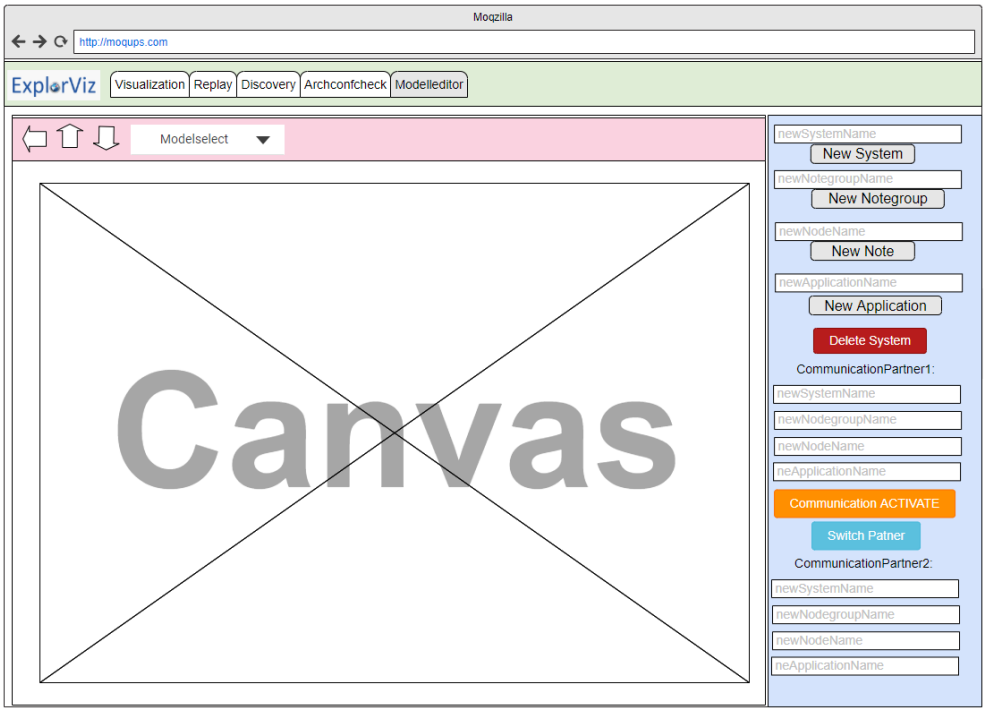

Abbildung 4.2: Mockup des Modelleditors

vas, in welchem das entstehende Model angezeigt wird. Unter der URL-Leiste lässt sich eine Reiterleiste (grün) ausmachen. Diese wird zur Navigation innerhalb von ExplorViz verwendet. Direkt darunter ist eine Toolleiste (rot) zu finden. Hier findet die Auswahl sowie das Hoch- & Runterladen von vorhandenen Landschaftsmodellen statt. Weitere Button zur Kamerasteuerung und Einstellungsmöglichkeiten befinden sich ebenfalls in dieser Toolleiste (rot). Bei der Toolleiste rechts (blau) handelt es sich um den Bereich, der die Funktionen zur Erstellung sowie Veränderung von Landschaftsmodellen beinhaltet. Die Oberflächensprache soll Englisch sein und sich somit der schon vorhandenen ExplorViz-Umgebung anpassen.

4.3 Architekturkonformitätsüberprüfung

Die Hauptaufgabe der Softwarearchitekturkonformitätsüberprüfung besteht in dem Vergleich zweier Versionen ein und derselben Software. Dabei handelt es sich meist um ein Modell, welches den gewünschten Soll-Zustand der Software widerspiegelt und um ein Abbild der tatsächlich existierenden Softwareumsetzung. Beim Vergleichen zweier Softwarelandschaften, diese werden hier mit Ist- und Soll-Zustand bezeichnet, kann man grundlegend vier Fälle unterscheiden:

1. Ein Teilsystem, das im Soll-Zustand gefordert wurde, liegt im Ist-Zustand vor
2. Ein Teilsystem, das im Soll-Zustand nicht gefordert wurde, liegt im Ist-Zustand nicht vor
3. Ein Teilsystem, das im Soll-Zustand gefordert wurde, liegt im Ist-Zustand nicht vor
4. Ein Teilsystem, das im Soll-Zustand nicht gefordert wurde, liegt im Ist-Zustand vor

Im ersten Fall liegen keine Probleme vor. Er ist somit der gewünschte Zustand des Systems. Der zweite

Fall ist vielmehr ein theoretischer und wird daher in dieser Arbeit nicht genauer behandelt. Der dritte Fall impliziert das Fehlen eines gewünschten Teilsystems. Der vierte Fall beschreibt das Gegenteil des dritten Falls und impliziert damit das Vorhandensein eines unerwünschten Teilsystems. Diese Fälle werden in der Visualisierung aufgenommen und dabei wird auf vorhandene Konzepte zurückgegriffen. Der erste Fall soll innerhalb der Visualisierung den gewünschten Normalfall darstellen, daher wird auf eine extra Umfärbung verzichtet. Das Farbschema von ExplorViz wird in diesem Fall übernommen. Da der dritte Fall eine gewünschte aber nicht vorhandene Struktur darstellt, kann man eine Analogie zu einem *Geist* (folgend als GHOST bezeichnet) herstellen. Ein Geist ist ebenfalls etwas was nicht tatsächlich vorhanden ist, daher wird die Umfärbung der Systeme, die in den dritten Fall fallen, mittels eines hellen Blaus vorgenommen. Ein System, welches dem vierten Fall unterliegt, stellt eine Bedrohung dar, ähnlich eines unerwünschten Eindringlings oder Schadsoftware. Hierbei handelt es sich um architekturverletzende Softwarekomponenten, welche nicht im Modell konzipiert, aber in der realen Umsetzung vorhanden sind (nachfolgend als WARNING bezeichnet). Dies trifft unter anderem auch auf Viren, Würmer und eventuell gefährliche Backdoors im Code zu. Daher wird für die Umfärbung die Signalfarbe Rot gewählt, welche intuitiv Aufmerksamkeit erzeugt. Diese Färbung wurde bereits in vorangegangenen Arbeiten verwendet (Vgl. [Sim15]).

Beim Vergleich der Softwarearchitekturen werden die inneren Komponenten eines jeden Knotenpunktes genauso Bestandteil der Analyse, wie die äußeren Beziehungen der Teilsysteme untereinander. Im Folgenden wird gezeigt, wie Umfärbungen einer vorhandenen ExplorViz Visualisierung umgesetzt werden können. In der ersten Abbildung 4.3 wird eine Visualisierung gezeigt, welche aus der Beispielanwendung mit ExplorViz angezeigt wird.

Das Beispielsystem von ExplorViz wurde in den nachfolgenden Abschnitten und Abbildungen in seiner Farbigkeit der einzelnen Elemente manipuliert. Die Abbildungen dienen dazu, ein Gefühl für die verwendeten Farben und die Interaktion von Elementen und Kommunikationslinien zu erlangen. Ziel ist es, mögliche Visualisierungen zu präsentieren ohne dabei semantische Richtigkeit anzustreben. Das heißt, es kann aus den Abbildungen kein Rückschluss auf etwaige zugrundeliegende Softwarelandschaften gezogen werden. Die einzelnen Abbildungen sind isoliert zu betrachten. Die einzelnen Ansätze wurden im Verlauf der Arbeit nicht prozedural und nicht im Sinne von Trial-and-Error durchlaufen. Die Abbildungen sind lediglich dazu erstellt worden, um Vor- und Nachteile des jeweiligen Ansatzes herauszuarbeiten. Die Visualisierung des Beispielsystems wurde nun auf vier verschiedene Arten verändert. Es sei erwähnt, dass sich die nun folgenden 5 Bilder in größerem Format im Anhang befinden (siehe B). Die erste Veränderung bezieht sich auf die Einfärbung bestimmter Kommunikationen als GHOSTS oder WARNING. Um in den Abbildungen einzelne Elemente eindeutig zu bestimmen wird folgende Notation festgelegt: `Systemname//Nodename//Applikationsname`. Beispielsweist ist die Kommunikation zwischen `Pubflow//10.0.0.9//Neo4j` und `Pubflow//10.0.0.4//Provenance` als GHOST eingefärbt. Die Kommunikation zwischen `Pubflow//10.0.0.1//Jira` und `Pubflow//10.0.0.3//PostgreSQL` wurde als WARNING markiert. In Abbildung 4.4 sieht man was passiert, wenn man die eingefärbten Linien einfach zusammenlaufen lässt und dabei die Anordnung der Linien nicht verändert. Dieses Zusammenlaufen geschieht also ohne hierarchische Konventionen. Dabei besteht die Gefahr, dass sehr viele Kommunikationslinien mit geringer Aufrufzahl, also auch geringer Dicke, und unterschiedlichen Färbungen zusammengeführt werden. Sobald die Anzahl der zusammengeführten Kommunikationslinien größer ist als die Anzahl der Pixel der Dicke der zusam-

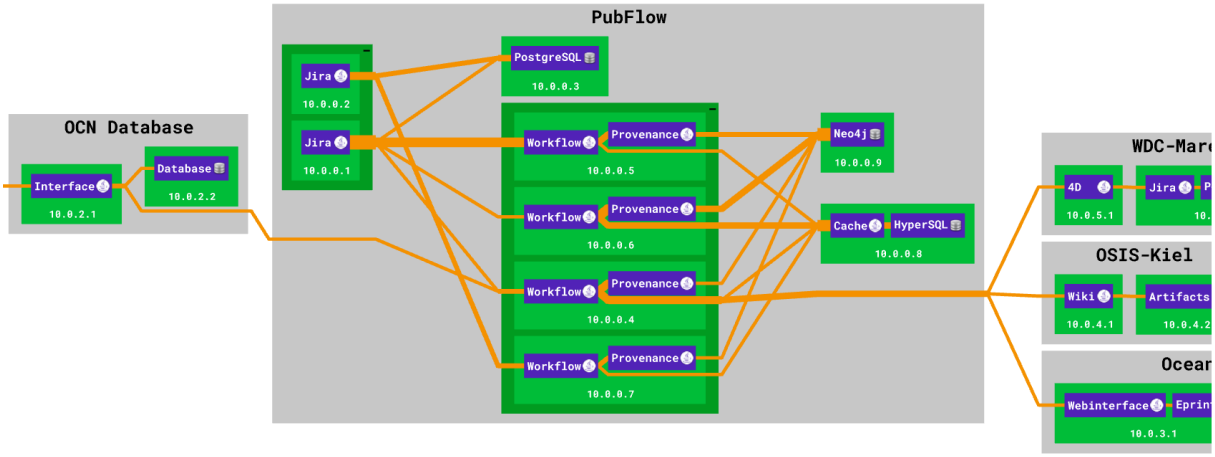

Abbildung 4.3: Beispielhafte Visualisierung einer Softwarelandschaft ohne Unterschiede zwischen Soll- und Ist-Zustand

mengeführten Kommunikation, kann es zu visuellen Artefakten und fehlerhaften Darstellungen kommen. Grund dafür ist, dass die Dicke der Teilkommunikationslinien nun geringer als ein Pixel ist. Daher wird diese triviale Herangehensweise an die Einfärbung der Kommunikationslinien verworfen. In der

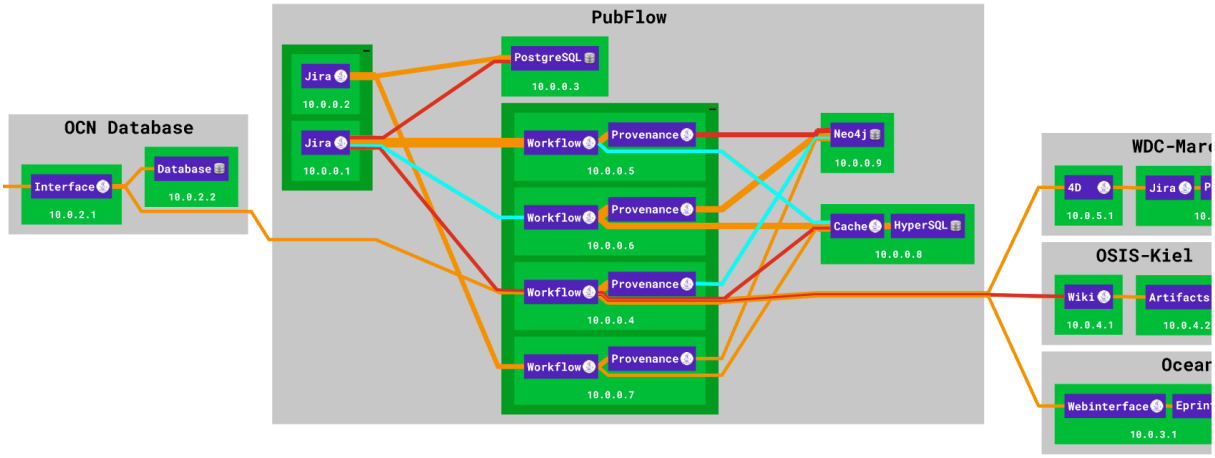

Abbildung 4.4: Eine Veränderung der Beispielszene 4.3 in der die Kommunikationen randomisiert eingefärbt wurden, und deren relative Lage nicht verändert wurde.

Abbildung 4.5 sieht man, was passiert, wenn man die Linien immer in einer bestimmten Reihenfolge koloriert. Dabei ist Rot immer oben, orange in der Mitte und blau unten. Dabei sei erwähnt, dass die Ansicht zweidimensional ist und diese nicht gedreht werden kann, sodass keine Probleme mit den Bezeichnungen *oben* und *unten* auftreten können. Dies führt zu einer Vermeidung des im vorherigen Abschnitt angesprochenen Problems. In der folgenden Visualisierung wurde die Visualisierung aus 4.5 so verändert, dass Transparenzen bei Schnitten eingefügt wurden. Dies ist in Abbildung 4.6 zu sehen. Der Zugewinn durch Transparenzen ist laut Aussagen von Entwicklern von ExplorViz nicht gegeben, sodass dieser Visualisierungsansatz nicht weiterverfolgt wird. In der letzten Visualisierungsvariante 4.7 wurden

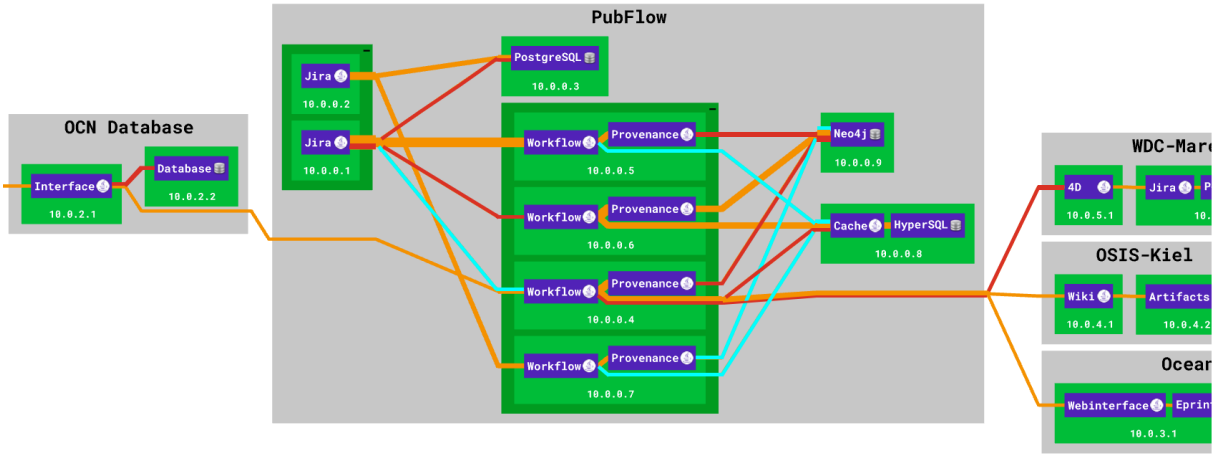

Abbildung 4.5: Die Beispielszene 4.3 wurde eingefärbt, die Lage der Farben innerhalb der Kommunikationslinien ist dabei fest.

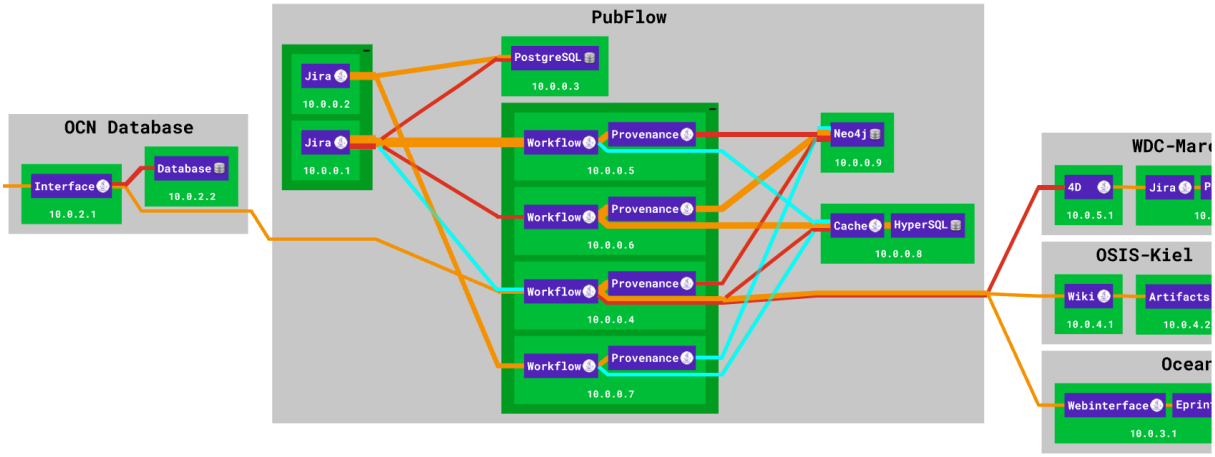

Abbildung 4.6: Eine Veränderung der Szene 4.5 in der die Überschneidungen durch Transparenz gesondert hervorgehoben werden.

zusätzliche Balken unterhalb des Namens der Applikation zur Verdeutlichung der inneren Komponenten hinzugefügt. Aufgrund des erheblichen Mehraufwandes bei kleinem Informationsgewinn wird dieser Ansatz in dieser Arbeit nicht weiterverfolgt.

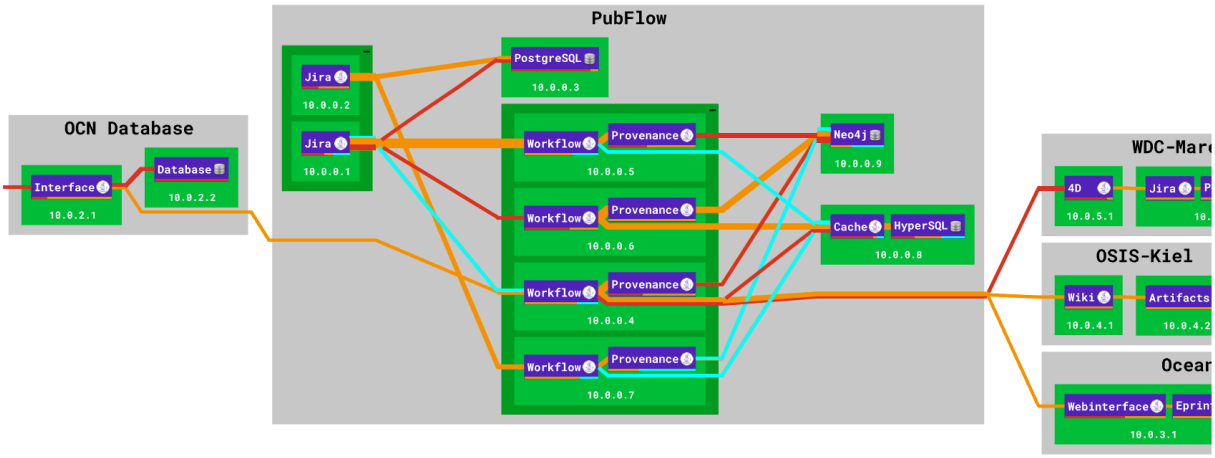

Abbildung 4.7: Eine Veränderung der Szene 4.5 in der die systeminternen Status durch eine unten angebrachte Statusleiste ablesbar sind.

4.3.1 Architekturkonformitätsüberprüfung - Backendansatz

Der Backendansatz stellt die Hauptkomponente der Softwarearchitekturkonformitätsüberprüfung dar. Es muss der Vergleich der beiden Softwarearchitekturen erfolgen. Dabei wird durch den strukturellen Datenmodellaufbau (siehe A.1) ein rekursiver Vergleich von Softwareelementen angestrebt. Das heißt, um eine Landschaft (`Landscape`) mit einer anderen zu vergleichen, müssen alle Syteme (`Systems`), deren Knotengruppen (`NodeGroups`), deren Knoten (`Nodes`), deren Applikationen (`Applications`) und deren Komponenten (`Components`) und Klassen (`Clazzes`) verglichen werden. Zusätzlich müssen die Kommunikationen zwischen Applikationen verglichen werden. Diese wird zum einen als `List` an der jeweiligen ausgehenden Applikation angeheftet. Zum anderen ist eine `List` aller Applikationskommunikationen am `Landscape` angeheftet. Eine der beiden Landschaften sollte als Grundlage für den Vergleich dienen. Für diese Arbeit wird angenommen, dass es sich hierbei um den Ist-Zustand handelt. Es werden alle Elemente des Ist-Zustands in ein neues Landschaftsobjekt übertragen. Nachfolgend wird über die neu kopierten Systeme iteriert. Jedes der Systeme wird mit jedem System im Soll-Zustand verglichen. Dabei werden drei Fälle unterschieden:

1. Das Ist-System hat ein identisches Soll-System
2. Das Ist-System hat kein identisches Soll-System
3. Nach der Iteration aller Ist-System, verbleiben bisher nicht betrachtete Soll-Systeme

Im ersten Fall wird das neu kopierte System als ASMODELLED gekennzeichnet. Die Knotengruppen von ASMODELLED-Systemen müssen wiederum auf ihren Status geprüft werden. Im zweiten Fall wird das neu kopierte System als WARNING gekennzeichnet. Die filialen Elemente (Knotengruppen, Knoten,

Applikationen, Komponenten und Klassen) werden ebenfalls als WARNING gekennzeichnet. Diese Vorgehensweise beruht auf dem Prinzip, dass kein filiales Element eines architekturverletzenden Elementes (WARNING) architekturkonform sein kann. Im dritten und letzten Fall wird das Soll-System als GHOST gekennzeichnet. Dasselbe Prinzip wie im zweiten Fall führt dazu, dass alle filialen Elemente ebenfalls mit GHOST gekennzeichnet werden. Dieses Vorgehen für Systeme kann äquivalent für Knotengruppen, Knoten und Applikationen übernommen werden. Komponenten und Klassen bilden hierbei eine Ausnahme, da Komponenten weitere Komponenten und Klassen enthalten können. Das heißt, Gleichnamigkeit muss innerhalb von Komponentenebenen geprüft werden. Dies wiederum bedeutet, dass es durchaus gleichnamige Komponenten geben kann, die nicht zur Kennzeichnung als ASMODELLED führen. Dies ist genau dann der Fall, wenn sie nicht auf einer Komponentenebene liegen. Klassen auf der anderen Seite haben niemals filiale Elemente. Filiale Elemente bezeichnen im genannten Zusammenhang eine Komposition (Vergleich [Bal99]). Wenn ein Element *A* ein filiales Element *B* aufweist, wird *A* als parentales Element von *B* bezeichnet.

Für Anschauungsbeispiele soll im Backend die Dummylandschaft auf zwei verschiedene Arten erweitert beziehungsweise manipuliert werden. Diese beiden `Landscapes` sollen im zweiten Schritt wie oben beschrieben miteinander verglichen werden, um eine Beispielvisualisierung zu erhalten.

4.3.2 Architekturkonformitätsüberprüfung - Frontendansatz

Im Frontend wird auf die im Backend gesetzten Status der Landschaftselemente eingegangen. Dabei sollen WARNING-Elemente rot überlagert werden, während GHOST-Elemente blau überlagert werden. Dabei kann auf die `Renderingservices` aus dem Frontendkern von ExplorViz zurückgegriffen werden. Die Kommunikation zwischen Back- und Frontend ist eine einmalige Ressourcenabfrage. Das heißt, die Daten, welche visualisiert werden, werden nicht mehr verändert. Ausschließlich wenn ein neuer Zeitpunkt oder ein verändertes Modell verglichen werden sollen, wird diese neue Ressource angefragt.

5 Umsetzung

Das folgende Kapitel beschreibt im Detail, wie das Konzept im Rahmen dieser Arbeit umgesetzt wurde. Hierbei soll darauf eingegangen werden, wie der Modelleditor und Softwarearchitekturkonformitätsüberprüfer jeweils im Back- & Frontend umgesetzt wurde. Das Softwareprojekt der vorliegenden Arbeit wird mittels Git versioniert. [1]

Eine Beschreibung der Umsetzung wird in den nun folgenden Unterkapiteln durchgeführt.

5.1 Überblick

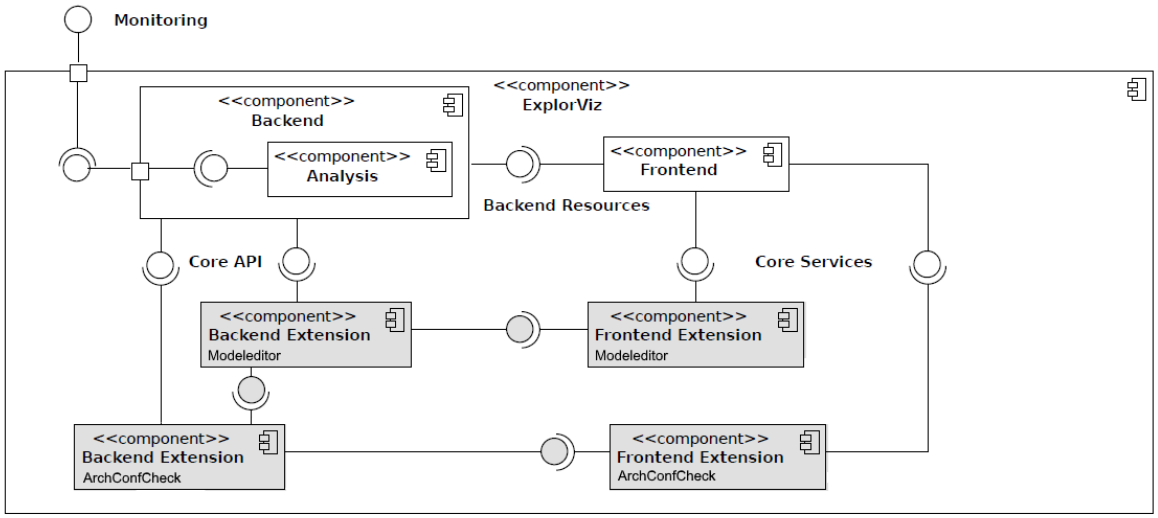

Abbildung 5.1: UML Komponentendiagramm von ExplorViz mit grau gefärbtem Eigenanteil dieser Arbeit

Aus der Abbildung 5.1 lässt sich erkennen, dass die Eigenanteile dieser Umsetzung sich in die vier Aufgaben unterteilen, die in den folgenden Kapiteln behandelt werden. Bei der technischen Umsetzung, welche in den nächsten Abschnitten beschrieben wird, lassen sich Unterschiede zwischen Front- und Backend-Erweiterungen ausmachen.

[1] Die dafür verwendeten Repositories lauten: https://github.com/ExplorViz/explorviz-backend-extension-modeleditor
https://github.com/ExplorViz/explorviz-frontend-extension-modeleditor
https://github.com/ExplorViz/explorviz-backend-extension-archconfcheck
https://github.com/ExplorViz/explorviz-frontend-extension-archconfcheck

5.2 Modelleditor

5.2.1 Modelleditor - Backendumsetzung

In diesem ersten Abschnitt wird das Backend des Modelleditors genauer betrachtet. Bei der Modelleditorerweiterung wurde auf die vorhandene Backend-Erweiterung von ExplorViz zurückgegriffen. Hierbei handelt es sich um die Dummy-Extension. Diese verwendet ein Gradle Projekt, welches mit Hilfe von maven und jitpack ein unabhängiges Backend generiert. Dieses neu generierte Backend erhält als externe Abhängigkeiten die neuste Version des Backend Cores und kann somit immer aktuell gehalten werden. Damit wird auch die Codeintegration und die zukünftige Aktualität gewährleistet.

Die erste Aufgabe bestand darin, eine Speicherung des neu zu erstellenden Landscapes im Backend zu ermöglichen. Da diese Funktionalität bisher noch nicht aus dem Frontend, welches im folgenden Kapitel genauer beschrieben wird, ausgelöst wurde. Die Umsetzung wurde durch die Erweiterung der Route `/landscape` erreicht. Dabei wurde sowohl ein POST als auch ein PATCH in der REST API implementiert. Diese beiden Aufrufe speichern das neu erstellte `Landscape-Objekt` in einen eigens dafür angelegten Unterordner serverseitig. Innerhalb der REST API wird ein GET bereitgestellt, welches eine Liste von vorhandenen Landschaftsobjekten zurückgibt. Die Ressource wird auf dem Pfad `landscapefilldropdown` bereitgestellt. Hierbei wird der angegebene Ordner auf Dateien durchsucht, welche die Endung .expl besitzen. Dieses Dateiformat wurde von ExplorViz bereitgestellt und stellt eine BASE64 kodierte JSON Datei eines Landscape Objektes dar. Ein im Frontend, über die eben genannte Dropdownliste, ausgewähltes Objekt kann durch entsprechende Funktionalität im Backend auch geladen werden. Das aktuelle Landschaftsmodell wird also im Backend bereitgestellt, aber im Frontend gehalten. Wichtig zu erwähnen ist, dass es momentan im Backendcore keine Möglichkeit gibt, IDs beim Up- oder Download neu zu setzen. Das hat zur Folge, dass gerade überwachte Landschaften nicht problemlos exportiert werden können um als Modell für den Modelleditor zu dienen. Wenn man im Modelleditor die importierte Landschaft verändert, verändert sich ebenfalls die überwachte Landschaft. Dies ist durch identische IDs im Emberstore bedingt.

5.2.2 Modelleditor - Frontendumsetzung

Die Umsetzung des Modelleditors im Frontend basiert auf JavaScript und integriert sich in die vorhandene Ember Umgebung. Nachdem das Frontend wie auf github.com[2] beschrieben eingerichtet wurde, wird nun eine Extension erstellt. Da dies auf GitHub detailliert erklärt wird, ist es nicht Fokus dieser Arbeit diesen Prozess genauer zu erläutern. Der Hauptteil der Implementierung des Frontends vom Modelleditor wurde in der Route Modelleditor umgesetzt. Dort wird die Logik für die Erstellung neuer Systeme, Nodegroups, Nodes, Applikations, Components und Clazzes implementiert. Dabei wird wie folgend veranschaulicht immer darauf geachtet, dass es innerhalb einer Landschaft nicht zwei gleichnamige Elemente gibt. Außer der Implementierung des Up- und Downloads, welche schon in Teilen an anderer Stelle vorhanden waren und daher wiederverwendet werden konnten, wurden noch Funktionalitäten zum Befüllen der Dropdownliste im Frontend implementiert.

[2] `https://github.com/ExplorViz/Docs`

```
newSystem(landscape){
  let foundDouble = false;

  for(let i =0 ; i < this.get('controller.model.systems').length;i++){
    if(this.get('controller.model.systems').objectAt(i).name === document.
        getElementById('nSN').value){
      this.showAlertifyMessage("You cannot have two systems with the exact
          same name.");
      foundDouble = true;
      break;
    }
  }
  if(foundDouble === false){
    const system = this.get('store').createRecord('system', {
      "name": document.getElementById("nSN").value,
      "parent": landscape,
      "id": Math.floor(Math.random() * 100000 + 10000)
    });
    this.get('controller.model.systems').addObject(system);
    this.get('modellRepo.modellLandscape').save();
    landscape.get('systems').addObject(system);
    this.set('modellRepo.modellLandscape', landscape);
    this.get('renderingService').reSetupScene();
  }
}
```

Codesnippet 5.1: aus modelleditor/addon/routes/modelleditor.js. Erstellung eines Systems.

Die IDs, die in Codesnippet 5.1 vergeben werden, sind randomisiert erzeugt, da momentan keine Möglichkeit besteht, eine atomare Zählervariable zwischen Front- und Backend zu synchronisieren. Eine mögliche Lösung für dieses Problem wäre eine vorübergehende ID, welche dann im Backend durch eine atomare Zählvariable ersetzt wird. Dies führt aber zur Erzeugung einer großen Anzahl an Elementen, welche im Anschluss wieder durch die *Garbage Collection* aufgegriffen werden müssten. Somit ist die Erzeugung randomisierter IDs, wenn auch nicht ideal, vorerst ausreichend. Weiterhin ist zu sagen, dass für alle einfügenden Operationen jeweils die Namen der parentalen Elemente verglichen werden. Dies wird in den Textfeldern im Editorfenster rechts angezeigt. Diese Elementnamen können auch per Linksklick in der Landschaftsansicht ausgewählt werden. Dies wurde in einer eigenen `Landscapeinteraction` umgesetzt (siehe Codesnippet 5.2).

```
initMyListeners() {
  const landscapeInteraction = LandscapeInteraction.create(getOwner(this).
      ownerInjection());

  this.get('landscapeInteraction').on('singleClick', function(emberModel) {
    if(emberModel){
      switch(emberModel.constructor.modelName){
        case "node":
          document.getElementById('nSN').value = emberModel.get('parent').
              get('parent').get('name');
          document.getElementById('nNgN').value = emberModel.get('parent').
              get('name');
          document.getElementById('nNN').value = emberModel.getDisplayName
              ();
```

```
11            document.getElementById('nAN').value = "to be selected";
12            break;
13          case "nodegroup":
14            document.getElementById('nSN').value = emberModel.get('parent').
                  get('name');
15            document.getElementById('nNgN').value = emberModel.get('name');
16            document.getElementById('nNN').value = "to be selected";
17            document.getElementById('nAN').value = "to be selected";
18            break;
19        }
20      }
21    }
22  }
```

Codesnippet 5.2: aus modelleditor/addon/controller/modelleditor.js. Initialisierung der `LandscapeInteraction`.

Bezugnehmend auf Abbildung 5.2 und Codesnippet 5.3 werden nun die einzelnen Bereiche der Abbildung erläutert. Innerhalb der .hbs Datei wird der strukturelle Aufbau ExplorVizs beibehalten, indem eine Toolleiste oberhalb der Visualisierung eingeblendet wird. Hierbei handelt es sich um den in rot markierten Bereich. Der strukturelle Aufbau wird aber, da es sich um einen Editor handelt, rechts um eine Toolleiste erweitert (blau markierter Bereich), welche zur Generierung und Editierung des Landscapes genutzt wird. Die Toolleiste oben ist, wie ExplorViz-typisch, weiterhin für Kamera- und andere Einstellungsmöglichkeiten verantwortlich.

Ziel ist es, die Benutzung der Oberfläche möglichst transparent und intuitiv zu gestalten. Daher werden alle veränderbaren Elemente textuell ausgegeben und somit vor Augen geführt. Die Bedienung der rechten Toolleiste verändert sich auch dann nicht, wenn von der Landschafts- zur Applikationsdarstellung gewechselt wird (Vgl: Abbildungen 5.2 und 5.3). Beim Wechsel verändern sich jeweils Funktion und Beschriftung der Textfelder, beziehungsweise deren Positionierung, nicht jedoch der Bedienstil.

```
1  <div class="canvas">
2  {{#if showLandscape}}
3    {{visualization/page-setup/visualization-navbar
4      content=(array
5        (component "visualization/page-setup/navbar/reset-visualization")
6        (component "visualization/page-setup/navbar/file-downloader"
7          urlPath=exportLandscapeUrl fileName=exportLandscapeFileName)
8        (component "visualization/page-setup/navbar/file-uploader"
9          backendURL=uploadLandscapeUrl)
10       (component "model-dropdown" replayModels=fillDropdownUrl)  )
11     // beispielhaft die Toolleiste oben wenn die Landschaftsansicht gewählt
           wurde
12   }}
13   {{visualization/rendering/landscape-rendering latestLandscape=modellRepo.
       modellLandscape interaction=landscapeInteraction}}
14 {{else}}
15   // Toolleiste oben wenn die Applikationsansicht gewählt wurde
16   }}
17   {{visualization/rendering/application-rendering latestApplication=
       modellRepo.modellApplication interaction=applicationInteraction}}
18 {{/if}}
19 </div>
```

```
20 <div class="buttonMenu">
21   <div class="buttonContainer" >
22     {{#if showLandscape}}
23       // Buttons wenn die Landschaftsansicht gewählt wurde
24     {{else}}
25   // Buttons wenn die Applikationsansicht gewählt wurde
26    New Component:<br>
27          <input id="nCComponentN" type="text" class="form-control"
           value="newComponentName">
28    <button id="bNComponentN" class="btn btn-default" {{action '
       newComponentInput' modellRepo.modellApplication}}>New Component</
       button><br>
29     {{/if}} </div></div>
```

Codesnippet 5.3: aus modelleditor/addon/templates/modelleditor.hbs. Angliederung der html Dateien an die Umgebung von ExplorViz

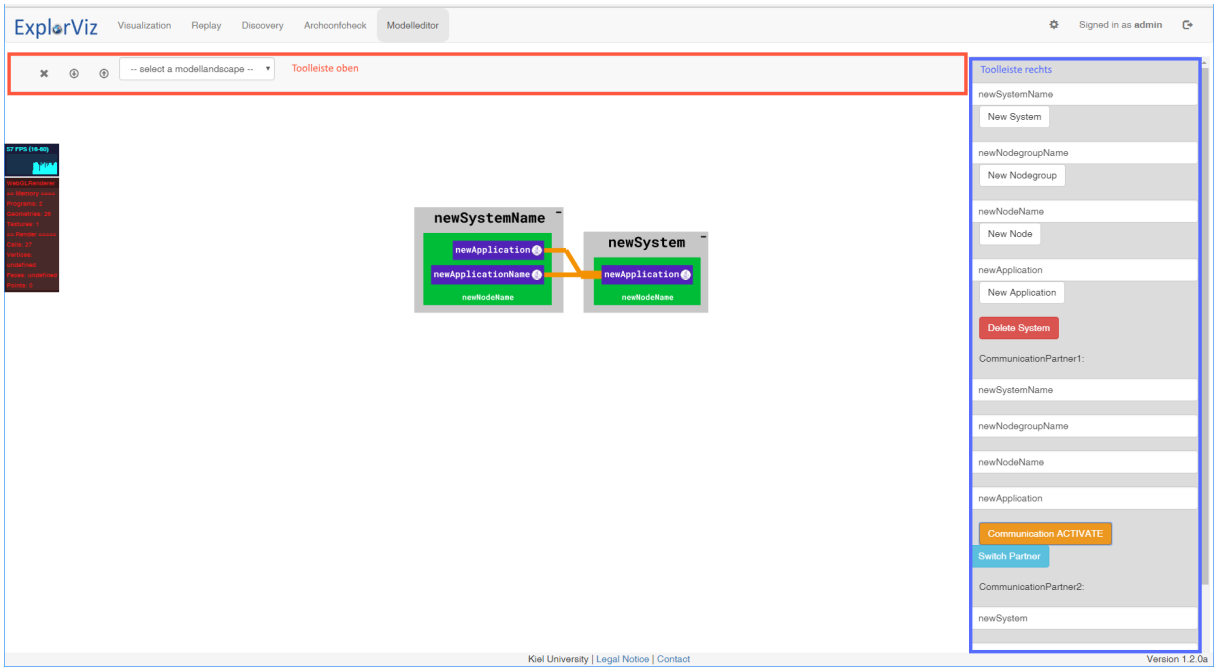

Abbildung 5.2: Die fertige Ausgabe des Frontends des Modelleditors während der Landschaftsansicht mit eingezeichneten Toolleisten

Erfolgt ein Klick auf ein Element in der Landschaftsansicht, wird dessen Name im jeweiligen Textfeld angezeigt. Des Weiteren werden alle parentalen Elemente in die parentalen Felder eingelesen. Dies geschieht hierarchisch und wird auch in der entsprechenden Hierarchie angezeigt. Dabei werden zusätzlich die ausgewählten Informationen in die Felder des ersten Kommunikationspartners eingetragen. Bei beiden Eintragungen werden nicht ausgewählte Kindelemente mit entsprechenden Texten gefüllt, welche anzeigen, dass eine genauere Auswahl zu treffen ist. Wenn man nun auf einen der Buttons klickt, welche sich direkt unter dem Textfeld befinden (New System/ Notegroup/ Note/ Applikation), so wird ein neues Element der obigen Klasse hinzugefügt. Dies kann auch durch einen Druck auf die Enter Taste im Textfeld hervorgerufen werden. Durch das Auswählen per Raycasting wird das Hinzufügen an bestimmten Stellen in der Landschaft erleichtert. All die genannten Auswahlen sind auch in der Applikationsansicht benutzbar und führen auch hier zu einer erleichterten Bedienung. Der Name, welcher bei Komponen-

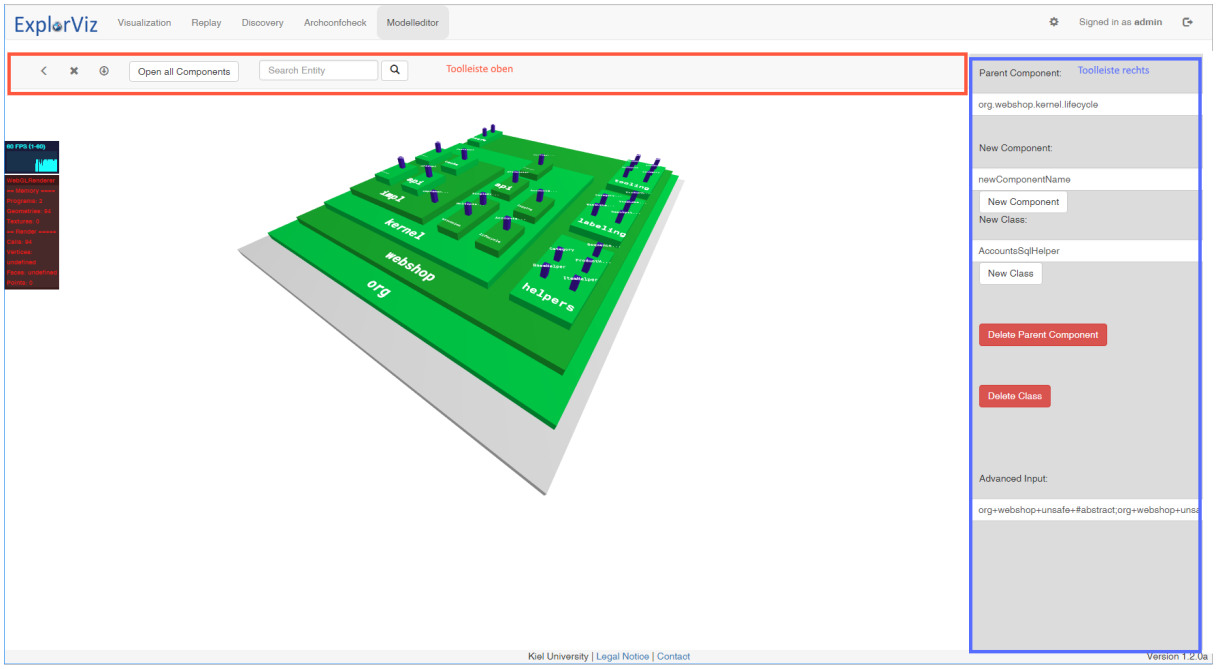

Abbildung 5.3: Die fertige Ausgabe des Frontends des Modelleditors während der Applikationsansicht mit eingezeichneten Toolleisten

ten und Clazzes verschachtelt werden kann, wird dabei immer vollständig angegeben. Die einzelnen Elemente werden durch Punkte hierarchisch voneinander getrennt. Am unteren Rand der Toolleiste in der Applikationsansicht wird außerdem ein Feld mit der Beschriftung advanced angeboten, welches nur durch den Druck auf Enter, nicht aber einen Button ausgeführt werden kann. Die Syntax der hier erstellten
Schnelladdition von verschiedenen Elementen wird im Folgenden erläutert. Man kann diesen advanced Modus nur in der Applikationsansicht verwenden. Dabei wird ein String entgegengenommen, welcher dann nach "`;`" aufgeteilt wird. Jedes durch Semikolon getrennte Element verheißt eine Klasse oder Komponente. Die Ebenen der Komponenten werden durch Pluszeichen angegeben. Wenn ein "`#`" vor einem Element steht, so wird eine Klasse mit dem nachfolgenden Namen erstellt. Diese Methodik wird nicht weiter auf Eingabeschwächen der Nutzer überprüft und sie wirft sehr viele Fehlermeldungen, welche von "Advanced Users" ignoriert werden können. Ein Beispiel für eine advanced Eingabe kann im README auf github.com[3] gefunden werden (Vgl. Abbildungen 5.2 und 5.3).

Die Kommunikation zwischen Front- und Backend findet ausschließlich über die REST API statt. Im speziellen Fall des Modelleditors wird nach jeder Änderung der Landschaft ein PATCH aufgerufen, mit dem das aktuelle Landschaftsmodell übermittelt wird. Das GET der Dropdownliste wird einmalig beim Aufruf der Route ausgeführt, wobei eine Liste von `Strings` übertragen wird. Die Landschaften werden per GET bereitgestellt, wenn auf den entsprechenden Eintrag in der Dropdownliste geklickt wird.

[3] `https://github.com/ExplorViz/explorviz-frontend-extension-modeleditor`

5.3 Architekturkonformitätsüberprüfer

5.3.1 Architekturkonformitätsüberprüfer - Backendumsetzung

Im Backend des Architekturkonformitätsüberprüfers wird die zentrale Logik der vorliegenden Arbeit implementiert. Der GET Request an die REST API erhält als Übergabeparameter zwei Timestamps. Dabei wird als erstes der Ist-Zustand und danach der Soll-Zustand mit ihrem eineindeutigen Identifikator, dem Zeitstempel, an das Backend übermittelt. Hierbei ist der Zeitstempel 0-0 zusätzlich als aktueller Zeitstempel konventioniert. Nachdem die Landschaften aus den jeweiligen Ordnern und Dateien geladen wurden (Vgl. Codesnippet 5.4), werden sukzessiv alle Elemente der Landschaft semirekursiv durchsucht. Semrekursiv bedeutet hierbei, dass die Aufrufe der einzelnen Elemente in getrennten Methoden behandelt werden müssen, da die einzelnen Aufrufe nicht identisch sein können (Vgl. Codesnippet 5.5). Eine Verallgemeinerung der Methoden auf BaseEntities von der alle Elemente des Landscape Objektes erben, verhilft nur mäßig zu besserer Lesbarkeit des Codes, da Switch Blöcke der gleichen Größe die Methodenrümpfe ersetzen. In der vorliegenden Arbeit wurde noch keinerlei Optimierung in Hinsicht auf Aufrufe vorgenommen. Momentan wird jedes Element der Ist-Landschaft mit jedem Element der Soll-Landschaft verglichen. Obwohl bereits ein Überspringen einiger Objekte durch “early-out“ gewährleistet wird, existieren noch weitere Möglichkeiten der Optimierung. “Early-out“ beschreibt hierbei den Abbruch eines Schleifendurchlaufes beim erfolgreichen Finden von gesuchten Elementen. Ein Anlegen einer Map oder Liste, in welcher die verglichenen Objekte nacheinander ein- und wieder ausgetragen werden, würde den Suchaufwand erheblich minimieren. Auch würde der in der momentanen Implementierung vorhandene Rückwärts-Suchlauf dadurch verhindert werden. Nichtsdestotrotz können die Landschaften in der getesteten Größe ohne Probleme in realtime zurück geliefert werden. An jedes Element wird dabei ein eigens dafür implementierter Status angehangen. Diese Erweiterbarkeit wird von ExplorViz in jedem von BaseEntity erbenden Objekt mittels ExtensionAttributes bereitgestellt. Die verwendeten Status sind entweder ASMODELLED, welches für eine Übereinstimmung von Ist- und Soll-Zustand steht, oder GHOST, was für ein im Soll-Zustand definiertes Element steht, welches im Ist-Zustand nicht vorhanden ist. WARNING steht als letztmöglicher Status für ein Element, welches im Ist-Zustand vorhanden ist, aber nicht im Soll-Zustand definiert wurde. Da parentale Elemente, welchen einen GHOST oder WARNING Status aufweisen, keinen anderen Zustand in ihren Kind-Elementen aufweisen können, wird für diese Kind-Elemente keine Überprüfung mehr vorgenommen, sondern lediglich der Status der Elternelemente entsprechend gesetzt. Dies kann allerdings nicht für ASMODELLED Elemente übernommen werden, da sich hier Kind-Elemente als GHOST oder WARNING herausstellen können. Die Umsetzung des Backendes wurde für die Evaluierung optimiert, dabei wird die Dummylandschaft, welche in ExplorViz existiert, um einige Elemente erweitert. Diese Erweiterungen sind dabei so gewählt, dass viele der auftretenden Sonderfälle abgedeckt werden. Für die Evaluation wurden Testdaten erstellt. Diese sind fest kodiert, da die Herausforderungen bezogen auf die identischen IDs (wie in 5.2.1 ausgeführt) im Backendkern nicht gelöst wurden.

```
30 final File modelDirectory = new File(
31   FileSystemHelper.getExplorVizDirectory() + File.separator +
       MODEL_REPOSITORY);
32   final File[] modelFileList = modelDirectory.listFiles();
```

```
33
34    if (modelFileList != null) {
35      for (final File f : modelFileList) {
36        final String filename = f.getName();
37        if (filename.endsWith(".expl") && filename.equals(modelString + ".
              expl")) {
38          // first validation check -> filename
39          modelLandscape = api.getLandscape(Long.parseLong(modelString.split(
                "-")[0]), MODEL_REPOSITORY);
40          break;
41        }
42      }
43    } else {
44    // error modelReplayRepository is empty
45  }
```

Codesnippet 5.4: Aus archconfcheck/ressources/ArchConfCheckRessource.java. Öffnen und Speichern einer Landscape Datei auf ein Landscape Objekt.

```
46  private void setStatusOfNodegroups(final System comparedSystem, final
        System system, final Status status) {
47    if (system != null) {
48      for (final NodeGroup nodeGroup : system.getNodeGroups()) {
49        final NodeGroup comparedNG = new NodeGroup();
50        comparedNG.setName(nodeGroup.getName());
51        comparedNG.setParent(comparedSystem);
52        comparedNG.getExtensionAttributes().put(saveAs, status);
53        setStatusOfNodes(comparedNG, nodeGroup, status);
54        comparedSystem.getNodeGroups().add(comparedNG);
55      }
56    }
57  }
58
59  private void setStatusOfNodes(final NodeGroup comparedNodeGroup, final
        NodeGroup nodeGroup, final Status status) {
60    if (nodeGroup != null) {
61      for (final Node node : nodeGroup.getNodes()) {
62        final Node comparedNode = new Node();
63        comparedNode.setName(node.getName());
64        comparedNode.setParent(comparedNodeGroup);
65        comparedNode.getExtensionAttributes().put(saveAs, status);
66        setStatusOfApplications(comparedNode, node, status);
67        comparedNodeGroup.getNodes().add(comparedNode);
68      }
69    }
70  }
```

Codesnippet 5.5: Aus archconfcheck/ressources/ArchConfCheckRessource.java. Verschiedene Methodenrümpfe aufgrund der unterschiedlichen Namensgebungen, welche Status an Kind-Elemente weitergeben.

5.3.2 Architekturkonformitätsüberprüfer - Frontendumsetzung

Um beim Architekturkonformitätsüberprüfer das Frontend einzubinden, ist es momentan noch von Wichtigkeit im ExplorViz Frontend auf den `Branch` namens `tim_archconfcheck` zu wechseln. Innerhalb dieses `Branches` wurden Veränderungen getroffen, welche zu Fehlermeldungen führten. Dabei wurden vor allem Abfragen auf `null` beziehungsweise `not definied` hinzugefügt.

Der Architekturkonformitätsüberprüfer besitzt im Frontend eine eigene Route. Diese Erweiterung soll dabei stets nur mit der Extension Modelleditor verwendet werden. Im Gegensatz dazu kann der Modelleditor zusätzlich autark verwendet werden. Dies ist aus Anwendersicht sinnvoll, da nicht jeder, der ein Modell des Systems erstellt, dies auch mit einer überwachten Landschaft vergleicht. Sollte jedoch der Architekturkonformitätsüberprüfer zum Vergleich verwendet werden, so muss auch eine Möglichkeit zur Anpassung des Soll-Zustandes unter Nutzung des Modelleditors gegeben sein. Durch diese semantische Verknüpfung der Frontend Erweiterung kann Code, durch Wiederverwendung, eingespart werden. In der Toolleiste oben (rot markierter Bereich) (Vergleich Abbildung 5.2) kann das zu verwendende Sollmodell ausgewählt werden. Durch einen Klick auf eine der beiden Auswahlen wird ein Request an das Backend gesendet, welcher dann zu einer Berechnung der Konformität führt und eine neu erstellte Softwarelandschaft mit erweiterten Status zurück liefert. Auf diese Status der einzelnen Elemente der Softwarelandschaft wird dann im Renderingservice eingegangen. Dieser Renderingservice hat einen vordefinierten von ExplorViz bereitgestellten `Configurationservice`. Dieser ist vor allem für die Farbgebung entscheidend. Dabei wird für jeden gewählten Zustand im Landschaftsmodell eine Farbpalette definiert (Vgl. Codesnippet 5.6).

```
71 export default ConfigService.extend({
72    warningLandscapeColors: {
73     system: "rgb(239,150,150)",
74   nodegroup: "rgb(179, 0, 0)",
75   node: "rgb(214, 29, 29)",
76   application: "rgb(112, 0, 0)",
77   communication: "rgb(255,0,0)",
78   textsystem: "rgb(0,0,0)",
79   textnode: "rgb(255,255,255)",
80   textapp: "rgb(255,255,255)",
81   collapseSymbol: "rgb(0,0,0)",
82   textchanged: false
83   },
84   warningApplicationColors: {
85     foundation: "rgb(239,150,150)",
86     componentOdd: "rgb(214, 29, 29)",
87     componentEven: "rgb(179, 0, 0)",
88     clazz: "rgb(112, 0, 0)",
89     communication: "rgb(255,0,0)",
90     highlightedEntity: "rgb(255,255,0)",
91   },
92   //...
93 }
```

Codesnippet 5.6: Aus archconfcheck/addon/services/color-configuration.js. Beispielhafte Deklaration von Farben, welche vom Renderingservice aufgegriffen werden.

Innerhalb der Architekturkonformitätsüberprüfung wurden die Kamerasteuerung, sowie andere grundlegende Interaktionen (Öffnen und Schließen von Knotengruppen, Systemen beziehungsweise den Wechsel in die Applikationsansicht) von ExplorViz übernommen. Dies hat den Vorteil, dass so ein durchgängiges Bedienkonzept sichergestellt wird.

6 Evaluation

Die vorliegende Evaluation (siehe C) wurde als Studie zur Evaluation des Modelleditors und des Architekturkonformitätsüberprüfers angelegt.

6.1 Ziel

Ziel ist es herauszufinden, in welchem Maß der erstellte Modelleditor und der Architekturkonformitätsüberprüfer den beabsichtigten Zweck erfüllt und inwiefern seine Benutzung als zufriedenstellend eingestuft wird.

6.2 Methodik

Die Evaluation wurde anhand eines Fragebogens durchgeführt. Dieser Fragebogen beinhaltet sowohl Aufgabenstellungen, welche vom Probanden zu bewältigen sind, als auch offen gestellte Fragen. Die Fragen sind entweder mit einem Freitext oder einer Beurteilung anhand einer Skala zu beantworten. Dies dient dazu, um die Rückmeldung der Probanden zu erhalten. Die Methode der Evaluation ist angelehnt an die in [Sal12] beschriebene Methode.

6.3 Experiment

Zunächst wird die Struktur des Fragebogens detailliert erläutert. Daran schließt sich eine Erklärung des Experimentaufbaus an. Als nächstes folgt eine Schilderung der Durchführung, sowie der Ergebnisse und deren anschließende Diskussion.

6.3.1 Fragebogenstruktur

Der Evaluationsbogen befindet sich im Anhang C. Der erste Teil des Evaluationsbogens besteht aus der Testvorbereitung. Hierbei gibt es zunächst sieben Merkmalsabfragen (Alterskategorie, Geschlecht, Beruf, höchster akademischer Abschluss, ExplorViz-Erfahrung, Programmiererfahrung, Sehvermögen), welche vom Versuchsleiter mit den individuellen Daten des Probanden auszufüllen sind. Dabei bedarf es teilweise eine Entscheidung zwischen Merkmalsabstufungen (z.B. zwischen Altersgruppen), teilweise aber auch einer Freitextantwort. Besonders zu betonen ist dabei die Programmiererfahrenheit und die Erfahrungswerte mit ExplorViz. Diese Werte sind entscheidend, da die Anwender der in dieser Arbeit erstellten Erweiterungen, als ExplorViz erfahren vorausgesetzt werden. Der Grund dafür ist, dass es eines grundsätzlichen Verständnisses von Softwarelandschaften sowie den speziellen Bezeichnungen von ExplorViz bedarf, um die Erweiterung effizient zu nutzen. Ebenfalls zur Testvorbereitung gehört

die Einführung des Probanden in ExplorViz und den Modelleditor durch den Versuchsleiter. Wichtig ist es, dass die Einführung interindividuell einheitlich durchgeführt wird. Aus diesem Grund verfügt der Evaluationsbogen über Stichworte anhand welcher die Einführung in ExplorViz und den Modelleditor stets kongruent erfolgt. Der zweite Teil des Evaluationsbogens besteht aus der Testausführung. Hierbei handelt es sich um den qualitativen Test des Modelleditors, ein schriftliches Interview sowie die Durchführung der Evaluation der Architekturkonformitätsüberprüfungskomponente. Der qualitative Test des Modelleditors enthält 17 schriftlich gestellte Aufgabenstellungen, welche vom Probanden auszuführen sind. Im darauffolgenden Interview werden dem Probanden wiederum schriftlich sechs Fragen gestellt, welche er schriftlich zu beantworten hat. Hierbei wird der Proband ebenfalls dazu aufgefordert, bestimmte Aspekte des Modelleditors zu bewerten, wobei er sich einer Skala, welche von 0 (schlecht) bis 10 (optimal) reicht, bedient. Anschließend folgt eine kurze Einführung des Probanden in die Architekturkonformitätsüberprüfungskomponente durch den Versuchsleiter. Diese erfolgt wie auch die vorangegangenen Einführungen anhand von Stichpunkten. Im letzten Teil der Testdurchführung wird nun die Architekturkonformitätsüberprüfungskomponente evaluiert. Dabei werden insgesamt zehn Fragen zu Merkmalen gestellt und der Proband soll ein jeweils zutreffendes Element nennen. Hierbei kann neben der Nennung der Komponente ebenfalls ein Worturteil durch den Probanden notiert werden.

6.3.2 Experimentaufbau

Alle Probanden führten die Evaluation am selben Gerät durch. Dieses Gerät verfügte über folgende Hardwaremerkmale:

Modell	Dell Optiplex 7010
CPU	Intel Core i5 3470 4x 3,20 GHz
RAM	8 GB
Displaygröße	24 Zoll (Widescreen)
Displayauflösung	1920 x 1080 Pixel

Tabelle 6.1: Hardwaremerkmale des für die Evaluation genutzten Gerätes

Das Betriebssystem des Geräts war Windows 10 64 Bit. Außerdem lag das Java Software Development Kit in Version 8 vor.

6.3.3 Experimentdurchführung

Vor dem Experiment wurde einmalig ein Pretest durchgeführt. Die Rückmeldung aus dem Pretest wurde benutzt um den Evaluationsbogen zu verbessern. So wurden einige Aufforderungen und Fragen umfor-

muliert und so verständlicher gemacht. Die fünf Probanden wurden innerhalb des Institutes für Softwaretechnologie der Christian-Albrecht-Universität zu Kiel durch persönliche Ansprache rekrutiert. Die Probanden führten die Evaluation ohne Zeitlimit durch. Es nahm stets nur ein Proband zur selben Zeit am Experiment teil. Neben dem Probanden war nur der Testleiter anwesend. Die Absicht war es, die Bedingungen der Evaluation kongruent zu gestalten, um mögliche Störfaktoren zu minimieren. Die Evaluation erfolgte für alle Probanden am selben Arbeitsplatz, welcher sich im Büro des Autors an der Christian-Albrecht-Universität zu Kiel befindet. Weiterhin wurde die Evaluation stets durch denselben Versuchsleiter durchgeführt. Es wurde darauf geachtet, dass die Positionierung des Probanden zum Versuchsleiter stets die selbige war. Keiner der Probanden erhielt eine Entschädigung finanzieller oder anderer Art für seine Teilnahme. Zu Beginn des Experiments wurde dem Probanden der Arbeitsplatz mit vorbereitetem Versuchsaufbau gezeigt. Darnach folgte die Aufnahme der persönlichen Daten und des relevanten Vorwissens des Probanden durch den Testleiter. Anschließend erfolgte eine kurze Einweisung des Probanden in ExplorViz und den Modelleditor. Hierbei wurde darauf geachtet, dass für das Experiment wichtige Termini deutlich erklärt wurden. Danach wurde dem Probanden ein leerer Evaluationsbogen ausgehändigt und er wurde dazu aufgefordert die Aufgaben der Testausführung auszuführen. Hier wurde auch darauf hingewiesen, dass der Proband zu jeder Zeit Fragen an den Versuchsleiter stellen könne. Parallel zur Bearbeitung der Aufgaben machte sich der Versuchsleiter Notizen zum Vorgehen der Probanden. Nach Abschluss der Bearbeitung der Aufgaben erfolge die schriftliche Beantwortung der Interviewfragen durch den Probanden. Auch hier wurde darauf hingewiesen, dass der Testleiter bei eventuellen Rückfragen zur Verfügung stehe. Diesen Abschnitt abschließend wurde nochmal mündlich nach Rückmeldung gefragt.

Zu Beginn des zweiten des Experiments wurde der Proband kurz in die Aufgabe und die Funktionen des Softwarearchitekturkonformitätsüberprüfers eingewiesen. Daran anschließend bearbeitete der Proband die dazu gehörigen Aufgaben. Nach deren Abschluss wurde erneut mündlich nach eventuellen zusätzlichen Rückmeldungen gefragt. Durch eine Verabschiedung endete anschließend das Experiment.

6.3.4 Ergebnisse des Experiments

Das Alter von 4 der Probanden lag in der Alterskategorie zwischen 21 und 30 Jahren. Ein Proband war zwischen 31 und 40 Jahren alt. Alle Probanden waren männlichen Geschlechts. Drei der fünf Probanden waren wissenschaftliche Mitarbeiter des Lehrstuhls für Softwaretechnologie der Christian-Albrechts-Universität zu Kiel. Die anderen beiden gaben an, Studenten zu sein. Alle Probanden machten die Angabe, Erfahrungen im Programmieren und in ExplorViz zu besitzen. Ein Proband gab an, unter einer Deuteranomalie zu leiden.

6.3.4.1 Ergebnisse des qualitativen Tests des Modelleditors

Die ersten vier Aufgaben des Evaluationsbogens sind inhaltlich als eine Einheit zu verstehen:

1. Erstelle ein `System` mit dem Namen ***"Grundsystem"***
2. Erstelle innerhalb des Grundsystems eine `NodeGroup` mit dem Namen ***"NodeGroup1"***
3. Erstelle innerhalb der Nodegroup1 einen `Node` mit dem Namen ***"Node1"***

4. Erstelle innerhalb von Node1 eine `Applikation` mit dem Namen ***"TestApp"***

Zweck der ersten vier Aufgaben, war es herauszufinden, ob die Probanden die geforderten Schritte einzeln nacheinander durchführen, oder bereits vorzeitig die in der Einführung angesprochene Transitivität der Button wiedererkannten und nutzten. Drei der fünf Probanden entschieden sich aufgrund der Transitivität dazu, die Schritte nicht einzeln auszuführen. Die anderen beiden Probanden beendeten einen Schritt zunächst vollkommen, ehe sie mit der Bearbeitung des nächsten Schrittes begannen. Hierbei äußerten sie zeitweise Zweifel, da leere Knotengruppen nicht angezeigt werden. Ferner werden auch die Namen von Knoten nur dann angezeigt, wenn sie mindestens eine Applikation enthalten. Auch dieser Umstand hat bei den prozedural arbeitenden Probanden Unsicherheit ausgelöst. Die darauffolgende Aufgabe umfasste dieselben Teilschritte wie die Aufgaben 1-4:

5. Erstelle ein zweites System mit dem Namen ***"FortgeschrittenesSystem"***, dieses soll auch eine ***"NodeGroup1"*** mit einem ***"Node1"*** und einer ***"TestApp"*** enthalten

Auch hierbei wurde nicht explizit auf die Transitivität hingewiesen, sondern der Evaluationsleiter beobachtete die Vorgehensweise der Probanden in Bezug auf die Nutzung der Transitivität. Hierbei stellte sich heraus, dass Probanden, die die Transitivität bereits in den 1. Aufgaben verwendet hatten, sie auch in der 5. Aufgabe nutzten. Jedoch traten hierbei zeitweise Unsicherheiten auf. Probanden die die Transitivität bei den Aufgaben 1-4 nicht nutzten, nutzten sie auch in Aufgabe 5 nicht.

6. Verbinde die beiden erstellten `Applikationen` mithilfe einer `Kommunikation`

Die 6. Aufgabe bewältigten alle Probanden ohne Probleme.

7. Erweitere das ***"Grundsystem"*** um eine weitere `Nodegroup` mit dem Namen ***"NodeGroup2"***

Auf die 7. Aufgabe reagierten alle 5 Probanden mit Kommentaren, welche ihre Unsicherheit beziehungsweise Verwirrtheit aufzeigten. Während ein Proband direkt verbal seine Zweifel an der Sinnhaftigkeit der Aufgabe äußerte, versuchten 2 Probanden auch noch in den folgenden Aufgaben die Notwendigkeit der Durchführung der 7. Aufgabe zu erkennen. Die verbleibenden 2 Probanden reagierten mit einer kurzen Pause in ihrem Arbeiten, gaben aber keinen verbalen Kommentar.

8. Benutze die Transitivität der Button um ein weiteres `System` Namens ***"System1"*** zu erstellen, welches wiederum eine ***"NodeGroup1"*** mit einem ***"Node1"*** und einer ***"TestApp"*** enthalten soll

In der 8. Aufgabe wird explizit auf die Transitivität der Button hingewiesen, da ihre Nutzung die Voraussetzung für die Bewältigung der Aufgabe darstellt. Alle Probanden lösten die 8. Aufgabe ohne Probleme. 2 der Probanden, die die Transitivität bereits ab der 1. Aufgabe nutzten, erleichterten sich in der 8. Aufgabe die Arbeit, indem sie Elemente in der Landschaft auswählten, anstatt deren Namen in die entsprechenden Felder manuell eintragen zu müssen.

9. Füge dem ***"Grundsystem"*** in der ***"NodeGroup2"*** zwei `Nodes` Namens ***"Node42"*** und ***"Node1337"*** hinzu

Die 9. Aufgabe lösten 4 von 5 Probanden ohne Probleme. Einem Probanden gelang das Lösen der Aufgabe zunächst nicht, was aber vermutlich an einem simplen Lesefehler lag. Infolge der Lösung der Aufgabe kam es bei einem Probanden zu einem Fehler. Dieser entstand durch das Versagen der Transitivität der

Button. Der Proband füllte das Feld “newNodeName” mit “Node42” aus und das Feld “newApplicationName” mit “LatestApp” aus, klickte dann auf den Button “New Application”. Bis hierhin verhielt sich der Modelleditor wie vorgesehen. Danach änderte er den Inhalt des Feldes “newNodeName” auf “Node1337” und klickte wiederum auf den Button “New Application”. Es kam zu keiner Fehlermeldung und auch zu keinem Erzeugen des Nodes. Erst durch den Druck auf “New Node” (durch den Evaluationsleiter) konnte das Problem behoben werden.

10. Beide gerade erstellten Nodes (***”Node42”*** und ***”Node1337”***) sollen um eine `Applikation` Namens ***”LatestApp”*** erweitert werden

Durch die Ausführung der 10. Aufgabe kam es zur Anzeige der Namen der in Aufgabe 9 erstellten Knoten. Dadurch, dass Knoten in Knotengruppen applikationsgleich sind, war es nicht nötig jedem Knoten einzeln eine Applikation hinzuzufügen, da dies automatisch von der Software gelöst wird. Zu diesem Umstand äußerten sich alle Probanden mit einem positiven Kommentar.

11. Füge einer der ***”TestApp”***s eine Komponente Namens ***”Root”*** hinzu

Alle Probanden lösten die 11. Aufgabe grundsätzlich ohne Probleme, jedoch kam es zu Verunsicherungen in Bezug auf das auszufüllende Feld.

12. Diese Komponente soll zwei beliebig benannte Unterkomponenten enthalten
13. Füge einer der gerade erstellten Unterkomponenten eine Klasse Namens ***”Implementation”*** hinzu

Die 12. und 13. Aufgabe lösten alle 5 Probanden ohne Probleme.

14. Verknüpfe die beiden ***“LatestApp”***s miteinander mit Hilfe einer *Kommunikation*

Die 14. Aufgabe zielte darauf ab, den selbstständigen Wechsel der Ansichten zu beurteilen. Dies wurde bei allen Probanden über den, innerhalb dieser Arbeit hinzugefügten, “Zurück”-Button gelöst. Die eigentliche Aufgabe ist identisch mit Aufgabe 6 und wurde auch hier von allen Probanden ohne Probleme gelöst.

15. Füge insgesamt 6 `Node`s oder `Applikation`en hinzu, sodass neue Applikationen erstellt werden.

Die 15. Aufgabe löste bei 4 der 5 Probanden Verwirrtheit aus. Nach einer Erklärung durch den Versuchsleiter, gelang es diesen 4 Probanden die Aufgabe zu lösen. Der 5. Proband interpretierte die Aufgabe anders, als vom Versuchsleiter gewünscht, wodurch seine Lösung nicht der Vorgesehenen entsprach. Der Proband fügte eine Applikation hinzu und füllte auf insgesamt 6 Applikationen auf, anstelle 6 neue Applikationen hinzuzufügen.

16. Von einer Applikation ausgehend, verbinde die restlichen gerade erzeugten Applikationen mit dieser.

Die letzte Aufgabe wurde von den 4 Probanden, die die vorangegangene Aufgabe wie vorgesehen gelöst hatten, problemlos gelöst. Der 5. Proband löste die 16. Aufgabe wiederum anders als vorgesehen, da er durch die vorangegangene Aufgabe nicht dieselbe Voraussetzung besaß.

Auf der beiliegenden CD befinden sich alle Screenshots, der Lösungen, welche von den jeweiligen Probanden erstellt wurden. Außerdem befinden sich auf der CD alle ausgefüllten Evaluationsbögen.

6.3.4.2 Ergebnisse des schriftlichen Interviews den Modelleditor betreffend

1. Wie gut kamst Du mit dem Erstellen der verschiedenen Elemente klar? [0*(schlecht)* - 10*(optimal)*]

Die 5 Probanden bewerteten ihren Umgang mit dem Erstellen von Elementen durchschnittlich mit 8,4/10 Punkten. Der Median beträgt 8/10 Punkten. Der Modalwert beträgt 8/10 Punkten.

2. Wie gut konntest Du innerhalb des Landscapeviews navigieren? [0*(schlecht)* - 10*(optimal)*]

Die 5 Probanden bewerteten ihre Navigation innerhalb der Landschaftsansicht durchschnittlich mit 9,5/10 Punkten (eine Enthaltung). Der Median beträgt 9,5/10 Punkten. Der Modalwert beträgt 9,5/10 Punkten.

3. Was würdest Du an der Bedienung ändern?

Die Probanden wünschten sich insgesamt eine modernere Bedienung (z.B. Drag&Drop und eine Aufhebung der Trennung von Modell und Editor), Explizit wurde der “Switch Partner”-Button als verwirrend und umständlich eingestuft. Ein Proband wünschte sich die Anzeige von leeren Knotengruppen und von Namen von Knoten ohne Applikation.

4. Wie gut war die Bedienung der transitiven Buttons? [0*(schlecht)* - 10*(optimal)*]

Die 5 Probanden bewerteten ihre Bedienung der transitiven Button durchschnittlich mit 9,2/10 Punkten. Der Median beträgt 10/10 Punkten. Der Modalwert beträgt 10/10 Punkten.

5. Wie gut war die Bedienung der Kommunikationslinien? [0*(schlecht)* - 10*(optimal)*]

Die 5 Probanden bewerteten ihre Bedienung der Kommunikationslinien durchschnittlich mit 7,8/10 Punkten (eine Enthaltung). Der Median beträgt 8/10 Punkten. Der Modalwert beträgt 8/10 Punkten. Dabei merkten Probanden zusätzlich an, dass Drag&Drop eine geeignetere Alternative zum “Switch Partner”-Button darstellen würde.

6. Was für Anregungen gibt es weiterhin?

Mehrere Probanden merkten an, dass es sinnvoll wäre leere Knotengruppen anzuzeigen. Editorübliche Schaltflächen wie “Rückgängig” und “neue leere Landschaft” wurden als fehlend eingestuft. Weiterhin wurde von mehreren Probanden geäußert, dass es besser wäre, neu hinzugefügte Komponenten immer geöffnet darzustellen. Der Zoomfaktor in der Applikationsansicht wurde als zu gering angemerkt. Als letztes wurde erneut auf die bessere Eignung von Drag&Drop hingewiesen.

6.3.4.3 Ergebnisse der ArchConfCheckEvaluation

Die Landschaft, die den Probanden präsentiert wurde um die Aufgaben zu bewältigen, lässt sich dem Anhang entnehmen (D.1)

Alle Aufgaben wurden von allen Probanden zu 100% korrekt gelöst. Es wurde verschiedenes Vokabular im Zusammenhang mit der Architekturkonformitätsüberprüfung verwendet, um zu prüfen, ob die in die-

ser Arbeit eingeführten Begriffe “GHOST” und “WARNING” leicht verstanden werden und anwendbar sind. Aufgaben in denen Umschreibungen genutzt wurden, wurden genauso gut gelöst, wie Aufgaben, in denen die Begriffe “GHOST” und “WARNING” genutzt wurden. In Einzelfällen ersetzten die Probanden die Umschreibungen durch die neu eingeführten Begriffe “GHOST” und “WARNING” (siehe Worturteil).

Einige Probanden schätzten die Farbwahl im Worturteil mit gut ein. Vor allem die Rottöne für “WARNING”-Elemente wurden positiv bewertet. Andererseits wurde darauf hingewiesen, dass der Unterschied zwischen den Farben der Applikationen in der Landschaftsansicht (blau und violett) nicht in allen Umgebungen deutlich erkannt wird. Dies setzt sich in der Applikationsansicht fort, da die Farben der Klassen den Farben der Applikationen in der Landschaftsansicht gleichen. Ein Proband gab in seinem Urteil die Farbhelligkeit als unruhig in ihrem Verlauf an. Er wünschte sich eine lineare Helligkeitszu- oder -abnahme vom Zentrum in die Peripherie. Probanden wünschten sich ebenfalls eine randständige Legende, welche die Farben und ihre Bedeutung innehält. Die Mehrheit der Probanden merkte außerdem an, dass die Ladezeiten der Landschaftsansicht zu lang seien. Bei der Evaluation ist außerdem aufgefallen, dass die Orientierung innerhalb von Landschaften besonders dann schwerfällt, wenn diese komplex und flächengroß sind.

6.3.5 Diskussion der Ergebnisse des Experiments

Im folgenden Abschnitt werden die soeben vorgestellten Ergebnisse diskutiert.

6.3.5.1 Diskussion des qualitativen Tests des Modelleditors

Die Struktur dieser Diskussion entspricht der Struktur des Evaluationsbogens. Es werden die Ergebnisse chronologisch diskutiert.

Zu 1.-4.: Allein durch die Auswertung der Ausführung der ersten vier Aufgaben kann nicht zweifelsfrei bestimmt werden, ob die Transitivität als intuitiv wahrgenommen wird. Das nacheinander Abarbeiten der Aufgaben durch zwei der Probanden kann auch durch ihre individuelle Arbeitsweise und Heuristik begründet sein.

Zu 5.: Die hier auftretenden Probleme, könnten mit der Unsicherheit in der Auswahl von willkürlichen Elementen in der Landschaft begründet werden.

Zu 6.: Dass alle Probanden die 6. Aufgabe problemlos lösten, zeugt davon, dass die Aufgabe entweder einen sehr geringen Schwierigkeitsgrad aufweist, oder das Erstellen einer Kommunikation intuitiv implementiert wurde.

Zu 7.: Zweck der 7. Aufgabe war es, das Maß der Unsicherheit des Probanden zu testen, da es bei dieser Aufgabe kein direktes visuelles Feedback gab. Aus den Reaktionen der Probanden lässt sich schließen, dass das Rendering oder Layouting innerhalb des Modelleditors so angepasst werden sollte, dass leere Knotengruppen angezeigt werden. Durch das direkte visuelle Feedback, könnte man Unsicherheiten des Nutzers minimieren. Weitere Lösungsvorschläge beinhalten die positive Bestätigung der Ausführung eines Befehls oder die Fehlermeldung beim Erstellen leerer Knotengruppen. Insgesamt könnte auch auf den Button “New Nodegroup” verzichtet werden, da er lediglich dazu benutzt wird, um leere Knoten-

gruppen zu erstellen. Er wurde innerhalb dieser Arbeit aus Gründen der Vollständigkeit implementiert.

Zu 8.: Aus dem durchweg reibungslosen Lösung der Aufgabe lässt sich schließen, dass die Transitivität nach einer Einführung durchaus verstanden und umgesetzt werden kann.

Zu 9.: Aus der erneuten fast reibungslosen Lösung der Aufgabe lässt sich schließen, dass es sich bei der 9. Aufgabe um eine unkomplizierte Aufgabe handelte. Aus dem Auftreten des Fehlers kann geschlossen werden, dass es einen speziellen Fall gibt, der bis zum jetzigen Zeitpunkt noch nicht in der Transitivität der Button berücksichtigt wird.

Zu 10.: Die positiven Kommentare der Probanden lassen erahnen, dass es sich beim automatischen Hinzufügen von Applikationen in Knoten derselben Knotengruppe um eine gewünschte Funktionalität handelt.

Zu 11.: Das Zögern der Probanden in Bezug auf das auszufüllende Feld lässt sich dadurch erklären, dass sich die Probanden vermutlich zunächst zur selben Stelle, an der sie ähnliche Aufgaben in der Landschaftsansicht gelöst hatten, orientierten. An dieser Stelle befindet sich in der Applikationsansicht jedoch das Feld “Parent Component”. Durch das Fehlen eines Buttons unter dem Feld “Parent Component” suchten die Probanden aktiv das richtige Feld (“ newComponentName”) und fanden es schließlich auch. Eine solche Verunsicherung ließe sich durch gleiche Anordnung der Felder in beiden Ansichten vermeiden. In dieser Arbeit wurde sich gegen die kongruente Anordnung entschieden, da “Parent Component” aus Gründen der Übersichtlichkeit an oberster Stelle stehen sollte.

Zu 12. und 13.: Auch diese Aufgaben wurden ohne Probleme gelöst. Dies lässt darauf schließen, dass der Umgang mit der Applikationsansicht des Modelleditors nach einer kurzen Einführungszeit unkompliziert ist.

Zu 14.: Aus der selbstständigen und prompten Nutzung des “Zurück”-Buttons, lässt sich schließen, dass dieser Button sich gut in den Workflow einpasst.

Zu 15.: Dass die Probanden die Aufgabe erst nach einer Erklärung fehlerfrei lösten, kann darin begründet sein, dass die Formulierung der Aufgabe nicht optimal war. Dies würde auch erklären, warum der 5. Proband sie anders als gewünscht interpretierte.

Zu 16.: Die Aufgabe bereitetet den Probanden keine Probleme. Daraus lässt sich schließen, dass der beabsichtigte Workflow durchaus nutzbar ist, auch wenn er veraltet erscheint. Die Umsetzung wurde als veraltet wahrgenommen, da die Interaktion nicht direkt am Landschaftsmodell erfolgte und der Fokus beim Bearbeiten immer zwischen Modell und der Toolleiste rechts wechselte.

6.3.5.2 Diskussion des schriftlichen Interviews den Modelleditor betreffend

Es besteht laut den Probanden noch Bedarf für Verbesserungen, besonders bei der Umsetzung von Drag&Drop, welche sowohl zur Kommunikationserstellung als auch zur Editierung von Elementen verwendet werden kann. Aus den Antworten der Probanden lässt sich schlussfolgern, dass bei der Anzeige von leeren Knotengruppen beziehungsweise bei der Anzeige von applikationslosen Knoten Verbesserungsbedarf besteht. Das Erstellen leerer Elemente sollte entweder unterbunden werden, oder ein direktes visuelles Feedback sollte erfolgen. Die Fragen nach der Zufriedenheit mit der Software wurden im Durchschnitt mit 8,7 auf einer Skala von 0 bis 10 bewertet. Insgesamt zeigte die Auswertung des

schriftlichen Interviews also, dass die Probanden ihre Erfahrungen mit dem Modelleditor in den abgefragten Bereichen als eher *gut* bis *sehr gut* einschätzten.

6.3.5.3 Diskussion der ArchConfCheckEvaluation

Die korrekte Lösung aller Fragen kann darauf zurückzuführen sein, dass der Schwierigkeitsgrad der Aufgaben zu gering war. Es könnte auf der anderen Seite aber auch dafür sprechen, dass die Umsetzung der Visualisierung gelungen ist. Bei den Aufgaben wurde darauf geachtet, dass Sonderfälle wie Überlagerung und Applikationsfarbwechsel getestet werden. Weiterhin wurden alle praxisrelevanten Aufgaben im Test abgedeckt. Daher ist davon auszugehen, dass nicht die Aufgaben zu simpel waren, sondern die Umsetzung als gelungen angesehen werden kann. Der Fakt, dass Aufgaben mit neu eingeführten Begriffen genauso gut gelöst wurden wie Aufgaben mit Umschreibungen, lässt vermuten, dass die neuen Begriffe angenommen und verstanden wurden. Die selbstständige Ersetzung der Umschreibung durch die neu eingeführten Bezeichnungen beweist ebenfalls die gelungene Annahme der Begriffe. Eine Veränderung der von einem Probanden kritisierten Farbgebung wurde innerhalb dieser Arbeit nicht umgesetzt, da sich am Farbschema von ExplorViz orientiert wurde, um die Benutzung innerhalb von ExplorViz schwellenarm zu halten. Durch die innerhalb der Evaluation erstellten flächengroßen Landschaften und die daraus resultierenden Navigationsprobleme wurde ersichtlich, dass die Umsetzung einer Minimap oder einer anderen Orientierungshilfe in Betracht gezogen werden sollte.

Abschließend lässt sich konkludieren, dass der Hauptzweck der vorliegenden Arbeit, also die Erstellung einer Visualisierung, welche das Ergebnis eines Architekturkonformitätsüberprüfers darstellt, auf anwendbare Weise umgesetzt wurde. Die Anmerkungen der Nutzer beziehen sich weitestgehend auf Aspekte die den Umgang mit der Software angenehmer gestalten würden. Diese sind dabei nicht für die grundsätzliche Nutzung essentiell.

6.3.6 Einschränkungen der Validität

Das Experiment wurde von allen Probanden am selben Gerät durchgeführt, dadurch können die Ergebnisse nicht ohne weiteres für andere Geräte und/oder Systeme angenommen werden. Um in diesem Sachverhalt Validität zu erreichen, müsste das Experiment an anderen Geräten mit anderen Hard- und Softwarevorraussetzungen wiederholt werden. Die geringe Stichprobengröße sowie deren große Homogenität bezüglich Geschlecht, Bildungslevel und Erfahrungsraum könnte die Validität weiterhin einschränken. Auch in diesem Fall trüge eine erneute Evaluation zur Erkenntnisgewinnung bei.

6.4 Zusammenfassung der Evaluation

Insgesamt lässt sich eine positive Rückmeldung der Probanden, sowie eine generelle gute Lösbarkeit der gestellten Aufgaben verzeichnen. Dementsprechend lässt sich vermuten, dass sich die vorliegende Software für die Architekturkonformitätsüberprüfung eignet und es sich lohnen würde diese auszubauen. Die Rohdaten der ausgefüllten Fragebögen befinden sich im Anhang E. Dort befindet sich auch der Evaluationsbogen (siehe C.)

7 Fazit

Im Gegensatz zu allen vorhandenen Softwarearchitekturkonformitätsüberprüfungssoftwares, welche auf Basis von statischer Codeanalyse erfolgen, bedient sich diese Arbeit dynamischer Daten. Ein weiterer Unterschied zur Mehrzahl der bereits bestehenden Softwarearchitekturkonformitätsüberprüfungssoftwares besteht in der visuellen Bereitstellung der Daten anstelle von textuellen beziehungsweise strukturiert-textuellen Datenausgaben. Durch das visuelle Aufarbeiten zweier Softwarezustände lassen sich architekturverletzende Elemente schnell ausfindig machen und gezielt kommunizieren. Die visuelle Aufarbeitung ist vorteilhaft, da so Sprachbarrieren (z.B. im Sinne von unterschiedlichem Vokabular) ihren negativen Einfluss auf die Kommunikation vermindern. Die visuelle Aufarbeitung lässt vermuten, dass eine schnellere, präzisere Arbeitsweise aus der Verwendung des in dieser Arbeit erstellten Softwarearchitekturkonformitätsüberprüfungstools hervorgeht. Das Verwenden von dynamischen Daten hat eine aktuelle, auf den aktiven Softwarearchitekturbereich begrenzte Analyse zur Folge. Hierbei bedeutet aktiv, dass Aufrufe von Klassen aufgezeichnet werden können. Die angesprochenen Vorteile können jedoch bei einer Evaluationsgruppenstärke von 5 Probanden nicht statistisch bewiesen werden.

Im Verlauf der Erstellung dieser Arbeit zeigte sich, dass sich die Softwarearchitekturkonformitätsüberprüfung gut in die vorhandene ExplorViz-Umgebung einfügt. Die Hauptaufgabe ExplorViz, nämlich die Kommunikation innerhalb einer Gruppe von Bearbeitern und die Erweiterung oder Erstellung einer Software zu ermöglichen, wird so um einen Aspekt erweitert. Bei diesem Aspekt handelt es sich um die Kommunikation zwischen Softwarearchitekten und implementierenden Softwareentwicklern.

Es stellte sich heraus, dass neue Erweiterungen problemlos in ExplorViz eingegliedert werden können. Dieser Umstand wird dadurch begünstigt, dass sich im Vergleich zu vorangegangenen Arbeiten die zugrundeliegende technische Basis verändert hat (Vergleich [Sim15]). Durch die Umsetzung der vorliegenden Arbeit wurde weiterhin verdeutlicht, dass Backenderweiterungen, welche momentan noch keinem bestimmten Schema folgen, ohne Probleme transitiv gestaltet werden können. Bei den Frontenderweiterungen gibt es dank Ember bereits eine sehr strukturierte und genaue Vorgehensweise.

Die Evaluation des Modelleditors zeigte, dass der in dieser Arbeit erstellte Modelleditor zur Aufgabenlösung geeignet ist. Jedoch gab es Verbesserungsvorschläge, die sich vor allem auf den Wunsch nach einem moderneren Workflow bezogen. Damit ist die noch vorhandene Trennung von Modell und Editortoolleiste, sowie das Fehlen von Drag&Drop gemeint. Verbesserungspotenzial zeigte sich ebenfalls in der Evaluation der Softwarearchitekturkonformitätsüberprüfung. Dies betrifft vorrangig die Farbgebung sowie die Ladezeiten. Die Evaluation zeigte aber auch, dass alle Aufgaben, die im Rahmen einer Softwarearchitekturkonformitätsüberprüfung auftreten, durch die Umsetzung der vorliegenden Arbeit abgedeckt sind.

8 Ausblick

Um die Güte des in dieser Arbeit erstellten Softwarearchitekturkonformitätsüberprüfers besser einschätzen zu können, ist es von Wichtigkeit eine erneute Evaluation mit einer größeren Stichprobe durchzuführen. Dabei könnten auch Vergleiche zu bisher bestehenden Softwarearchitekturkonformitätsüberprüfern erlangt werden. In der im Rahmen der vorliegenden Arbeit bereits durchgeführten Evaluation wurde jedoch schon deutlich, dass es Punkte gibt, welche Verbesserungspotenzial beinhalten. Hierzu gehört unter anderem die Farbwahl innerhalb der Softwarelandschaft. Es besteht bereits ein Ansatz zur individuellen Farbgestaltung innerhalb von ExplorViz. Dieser sollte jedoch um die Möglichkeit der Änderung aller Farben zur Architekturkonformitätsüberprüfung erweitert werden. Dabei könnten vorgefertigte Farbpaletten angeboten werden, zwischen welchen dann gewählt werden kann.

Auch im Modelleditor besteht Bedarf zu Optimierung. Aufgabe zukünftiger Arbeiten ist es, eine verbesserte Bedienung zu gewährleisten. Hierzu zählt beispielsweise die Einführung von Drag&Drop. Dieses könnte unter anderem für die Erstellung von Elementen oder Applikationskommunikationslinien verwendet werden. Eine weitere Verbesserung in der Nutzbarkeit könnte durch das Ermöglichen vom Markieren, Kopieren und Einfügen von Elementen erreicht werden. Momentan lassen sich bereits erstellte Elemente in der Softwarelandschaft nicht nachträglich manipulieren. Dies sollte zukünftig idealerweise ermöglicht werden. Um die Übersichtlichkeit innerhalb einer Softwarelandschaft zu ermöglichen, könnten zukünftige Arbeiten die Einführung von Minimaps und Farblegenden als Ziel haben.

Zukünftig ist auch die Verwendung von Virtual Reality Brillen denkbar. Dies könnte eine große Bereicherung für den Modelleditor und die Softwarearchitekturkonformitätsüberprüfung darstellen, da so intuitiver auf Funktionen zugegriffen werden könnte. Außerdem könnte dies eine kollaborative Arbeitsweise ermöglichen.

Abschließend lässt sich sagen, dass mit dieser Arbeit ein solider Grundstein gelegt werden konnte, welcher unter anderem die oben genannten Möglichkeiten des Ausbaus bietet.

Literaturverzeichnis

[AD07] ALAM, Sazzadul ; DUGERDIL, Philippe: EvoSpaces: 3D Visualization of Software Architecture. (2007), S. 500–505

[Bal99] BALZERT, Heide: Lehrbuch der Objektmodellierung: Analyse und Entwurf. (1999)

[BH09] BREWER, Cynthia ; HARROWER, Mark: *colorbrewer: color advice for carthography*. `http://colorbrewer2.org/#`. Version: 2009

[CZB11] CASERTA, Pierre ; ZENDRA, Olivier ; BODÉNES, Damien: 3D hierarchical edge bundles to visualize relations in a software city metaphor. (2011), S. 1–8

[FKH15] FITTKAU, Florian ; KRAUSE, Alexander ; HASSELBRING, Wilhelm: Exploring software cities in virtual reality. (2015), S. 130–134

[FKH17] FITTKAU, Florian ; KRAUSE, Alexander ; HASSELBRING, Wilhelm: Software landscape and application visualization for system comprehension with ExplorViz. In: *Information and software technology* 87 (2017), S. 259–277

[FRH15] FITTKAU, Florian ; ROTH, Sascha ; HASSELBRING, Wilhelm: ExplorViz: Visual runtime behavior analysis of enterprise application landscapes. (2015)

[FVWSN08] FEKETE, Jean-Daniel ; VAN WIJK, Jarke J. ; STASKO, John T. ; NORTH, Chris: The value of information visualization. (2008), S. 1–18

[FWWH13] FITTKAU, Florian ; WALLER, Jan ; WULF, Christian ; HASSELBRING, Wilhelm: Live trace visualization for comprehending large software landscapes: The ExplorViz approach. (2013), S. 1–4

[HJZ+17] HEINRICH, Robert ; JUNG, Reiner ; ZIRKELBACH, Christian ; HASSELBRING, Wilhelm ; REUSSNER, Ralf: An Architectural Model-Based Approach to Quality-aware DevOps in Cloud Applications. (2017), Juni, S. 69–89

[KP07] KNODEL, Jens ; POPESCU, Daniel: A comparison of static architecture compliance checking approaches. (2007), S. 12–12

[Lew14] LEWIS, James R.: Usability: lessons learned... and yet to be learned. In: *International Journal of Human-Computer Interaction* 30 (2014), Nr. 9, S. 663–684

[PKWB17] PRUIJT, Leo ; KÖPPE, Christian ; WERF, Jan M. d. ; BRINKKEMPER, Sjaak: The accuracy of dependency analysis in static architecture compliance checking. In: *Software: practice and Experience* 47 (2017), Nr. 2, S. 273–309

[Pla04] PLAISANT, Catherine: The challenge of information visualization evaluation. (2004), S. 109–116

[PTV+10] PASSOS, Leonardo ; TERRA, Ricardo ; VALENTE, Marco T. ; DINIZ, Renato ; CHAGAS MENDONCA, Nabor das: Static architecture-conformance checking: An illustrative overview. In: *IEEE software* 27 (2010), Nr. 5, S. 82–89

[PW92] PERRY, Dewayne E. ; WOLF, Alexander L.: Foundations for the study of software architecture. In: *ACM SIGSOFT Software engineering notes* 17 (1992), Nr. 4, S. 40–52

[Sal12] SALVENDY, Gavriel: *Handbook of human factors and ergonomics*. John Wiley & Sons, 2012. – 1267–1312 S.

[Sim15] SIMOLKA, Till: Live architecture conformance checking in ExplorViz. (2015)

[Sol13] SOLUTIONS, Ryobi S.: *visolve: the assistive software for people with color blindness*. `https://www.ryobi-sol.co.jp/visolve/en/`. Version: 2013

[VEBHH11] VAN EYCK, Jo ; BOUCKÉ, Nelis ; HELLEBOOGH, Alexander ; HOLVOET, Tom: Using code analysis tools for architectural conformance checking. (2011), S. 53–54

[ZKH18] ZIRKELBACH, Christian ; KRAUSE, Alexander ; HASSELBRING, Wilhelm: On the Modernization of ExplorViz towards a Microservice Architecture. Online Proceedings for Scientific Conferences and Workshops (2018), Februar

Abbildungsverzeichnis

1.1 Venn-Diagramm der betrachteten Softwarezustände 4

2.1 Ergebnisse der Detektierung von direkten Abhängigkeiten (Vgl:[PKWB17]) 5
2.2 Ergebnisse der Detektierung von indirekten Abhängigkeiten(Vgl:[PKWB17]) 6

3.1 Applikationsansicht innerhalb von ExplorViz 10

4.1 UML Komponentendiagram von ExplorViz (Vgl:[FKH17]) 13
4.2 Mockup des Modelleditors 15
4.3 Beispielhafte Visualisierung einer Softwarelandschaft ohne Unterschiede zwischen Soll- und Ist-Zustand 17
4.4 Eine Veränderung der Beispielszene 4.3 in der die Kommunikationen randomisiert eingefärbt wurden, und deren relative Lage nicht verändert wurde. 17
4.5 Die Beispielszene 4.3 wurde eingefärbt, die Lage der Farben innerhalb der Kommunikationslinien ist dabei fest. 18
4.6 Eine Veränderung der Szene 4.5 in der die Überschneidungen durch Transparenz gesondert hervorgehoben werden. 18
4.7 Eine Veränderung der Szene 4.5 in der die systeminternen Status durch eine unten angebrachte Statusleiste ablesbar sind. 19

5.1 UML Komponentendiagramm von ExplorViz mit grau gefärbtem Eigenanteil dieser Arbeit 23
5.2 Die fertige Ausgabe des Frontends des Modelleditors während der Landschaftsansicht mit eingezeichneten Toolleisten 27
5.3 Die fertige Ausgabe des Frontends des Modelleditors während der Applikationsansicht mit eingezeichneten Toolleisten 28

A.1 UML des Backend Datenmodells 54

B.1 Abbildung 4.3 im Großformat: Beispielhafte Visualisierung einer Softwarelandschaft ohne Unterschiede zwischen Soll- und Ist-Zustand 56
B.2 Abbildung 4.4 im Großformat: Eine Veränderung der Beispielszene 4.3 in der die Kommunikationen randomisiert eingefärbt wurden, und deren relative Lage nicht verändert wurde. 57
B.3 Abbildung 4.5 im Großformat: Die Beispielszene 4.3 wurde eingefärbt, die Lage der Farben innerhalb der Kommunikationslinien ist dabei fest. 58
B.4 Abbildung 4.6 im Großformat: Eine Veränderung der Szene 4.5 in der die Überschneidungen durch Transparenz gesondert hervorgehoben werden. 59

B.5 Abbildung 4.7 im Großformat: Eine Veränderung der Szene 4.5 in der die systeminternen Status durch eine unten angebrachte Statusleiste ablesbar sind. 60

D.1 Visualisierung des Architekturkonsormitätsüberprüfers im Rahmen der Evaluation . . . 68

Tabellenverzeichnis

6.1 Hardwaremerkmale des für die Evaluation genutzten Gerätes 34

E.1 Rohdaten der Evaluation des Architekturkonformitätsüberprüfers; $\surd$ steht für eine richtige Lösung 69

A Backend Datenmodell

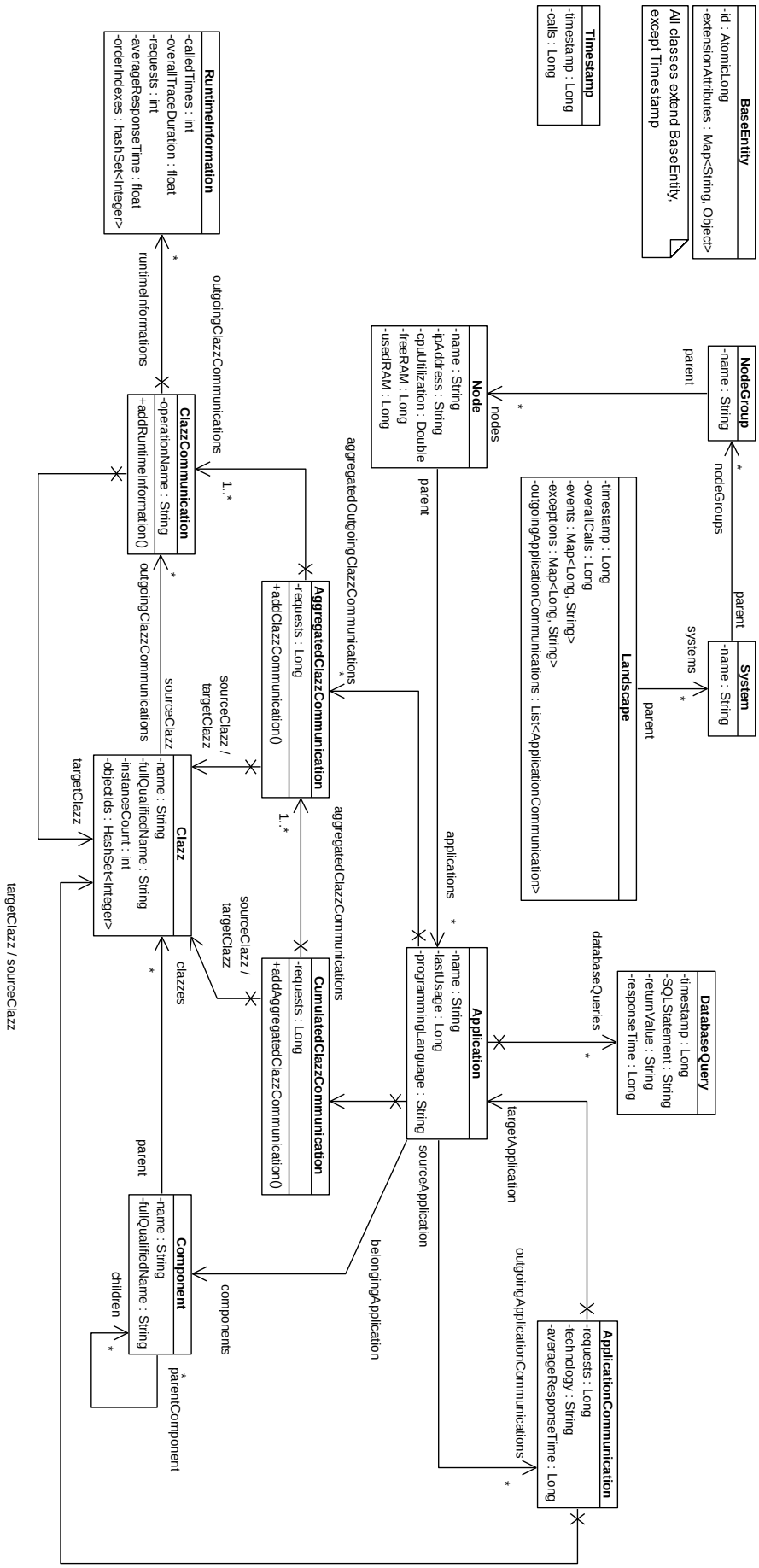

Abbildung A.1: UML des Backend Datenmodells

B Bilder des Ansatzes im Großformat

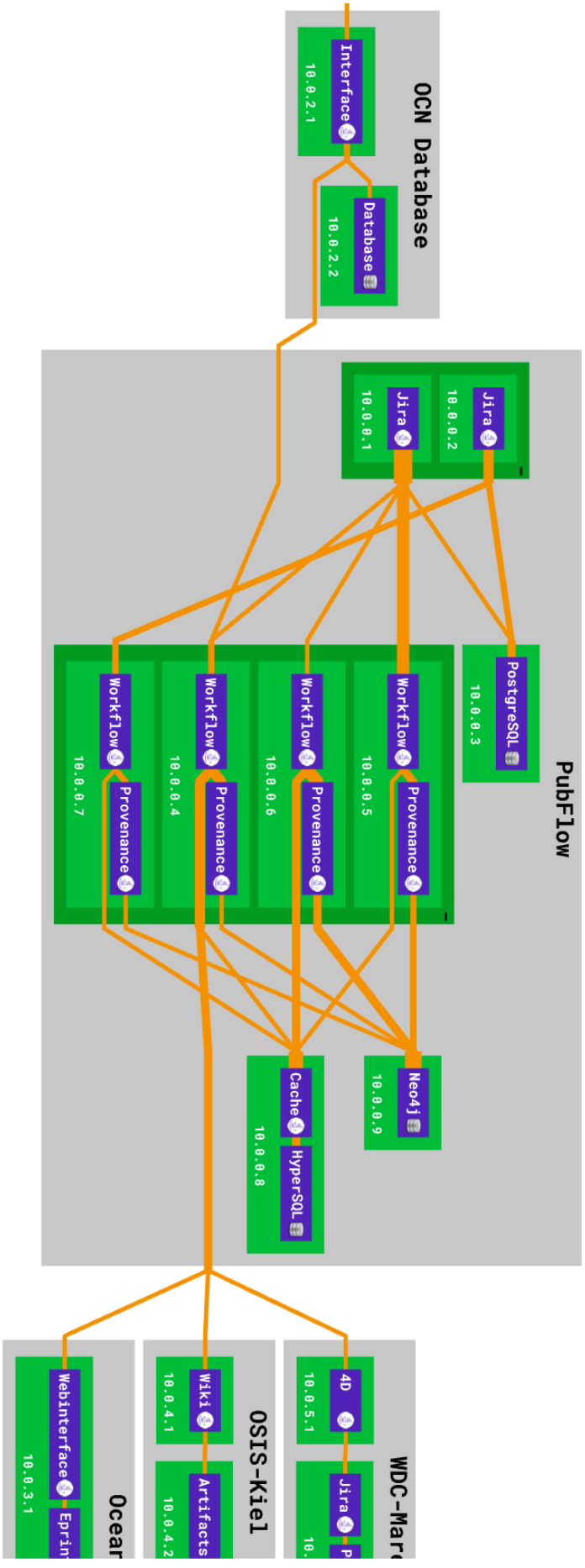

Abbildung B.1: Abbildung 4.3 im Großformat: Beispielhafte Visualisierung einer Softwarelandschaft ohne Unterschiede zwischen Soll- und Ist-Zustand

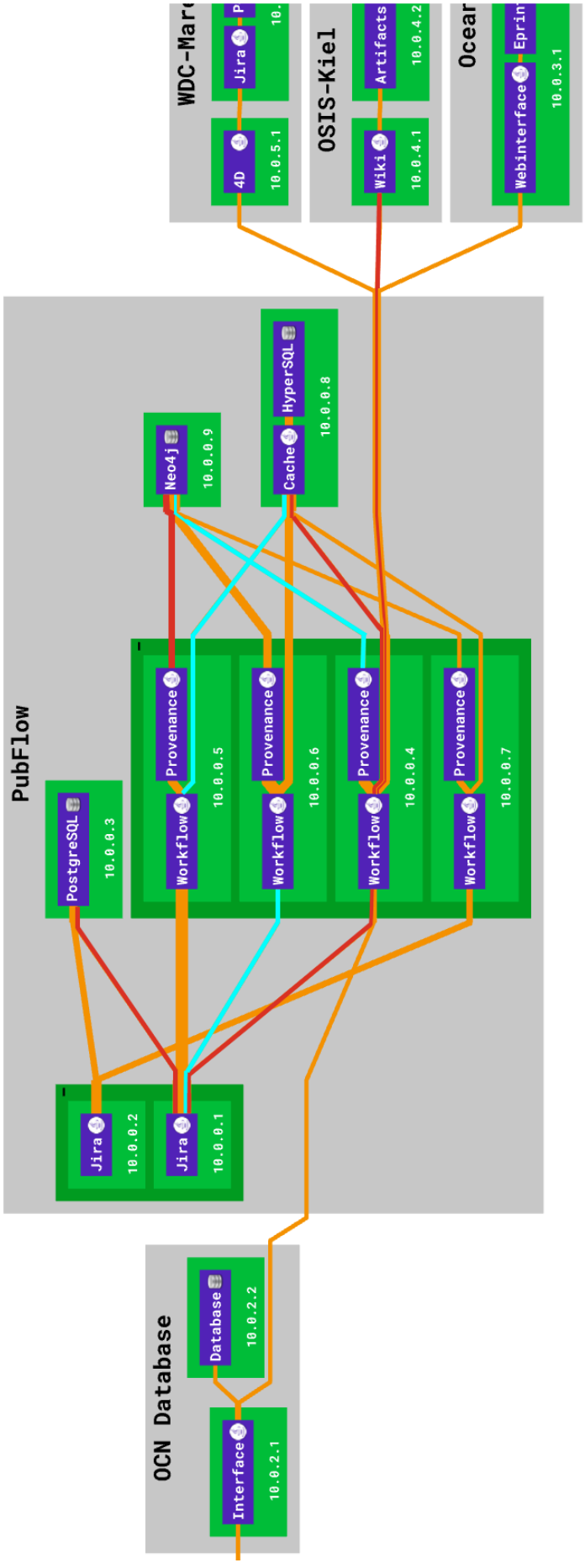

Abbildung B.2: Abbildung 4.4 im Großformat: Eine Veränderung der Beispielszene 4.3 in der die Kommunikationen randomisiert eingefärbt wurden, und deren relative Lage nicht verändert wurde.

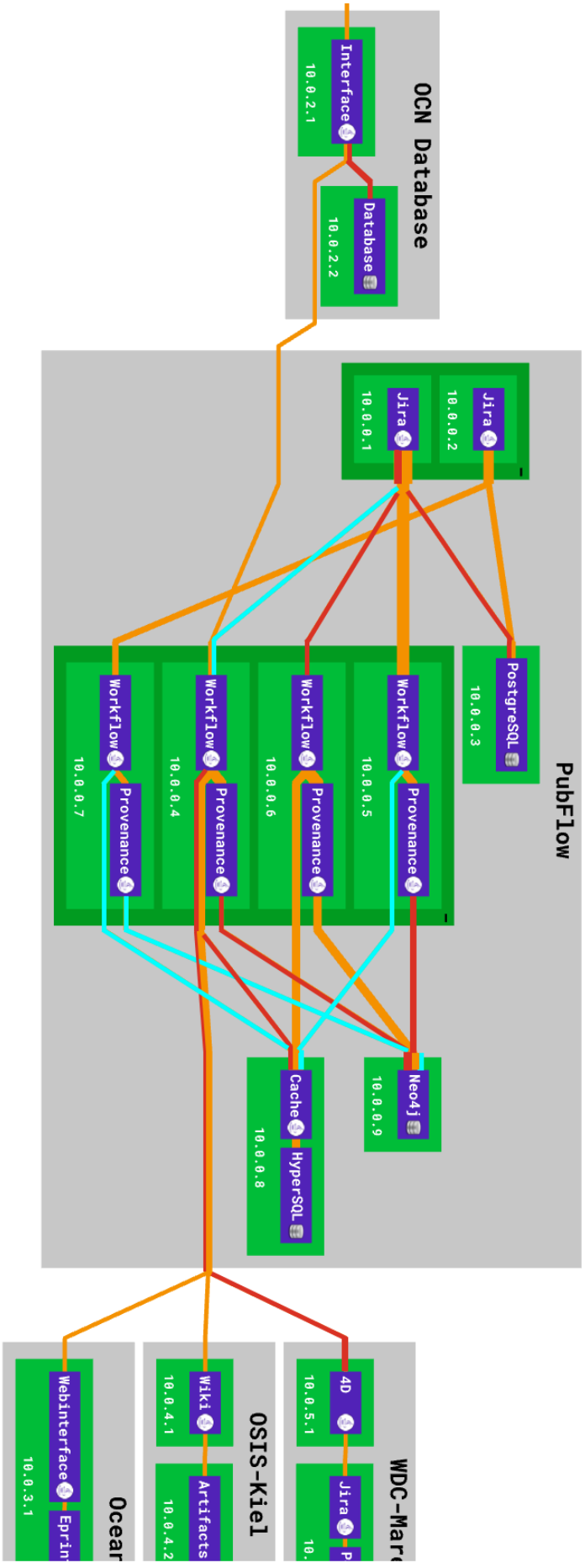

Abbildung B.3: Abbildung 4.5 im Großformat: Die Beispielszene 4.3 wurde eingefärbt, die Lage der Farben innerhalb der Kommunikationslinien ist dabei fest.

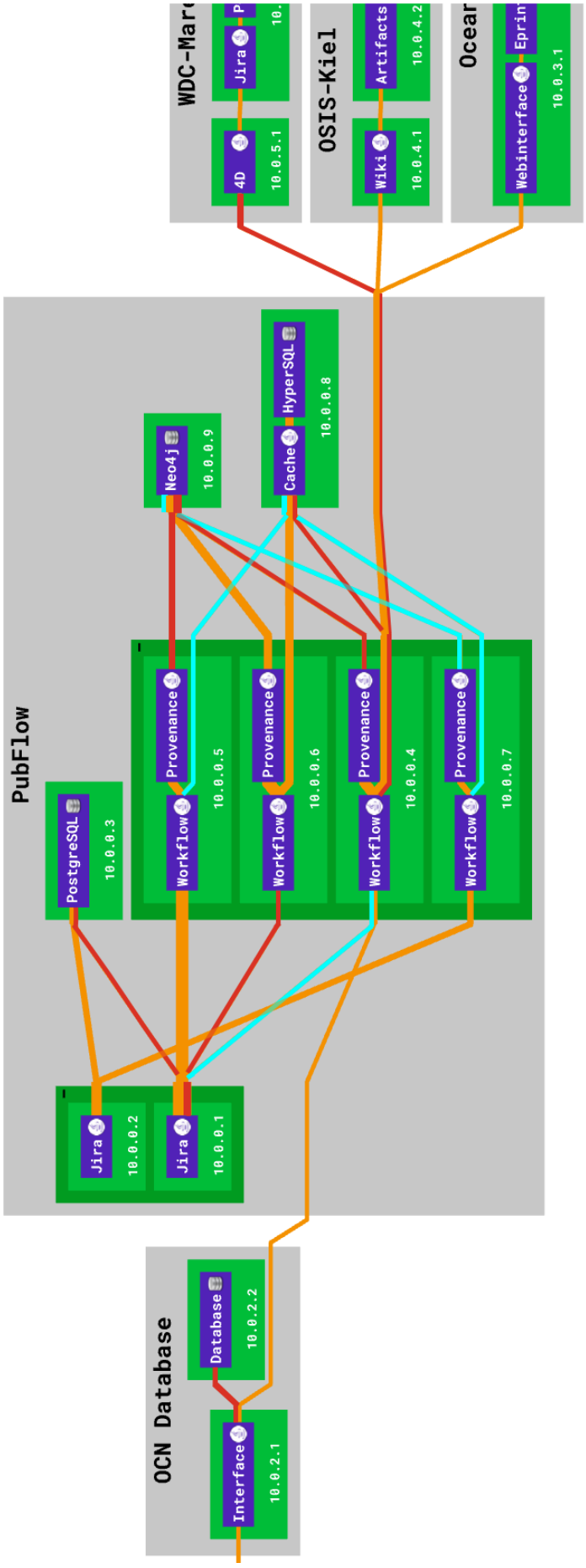

Abbildung B.4: Abbildung 4.6 im Großformat: Eine Veränderung der Szene 4.5 in der die Überschneidungen durch Transparenz gesondert hervorgehoben werden.

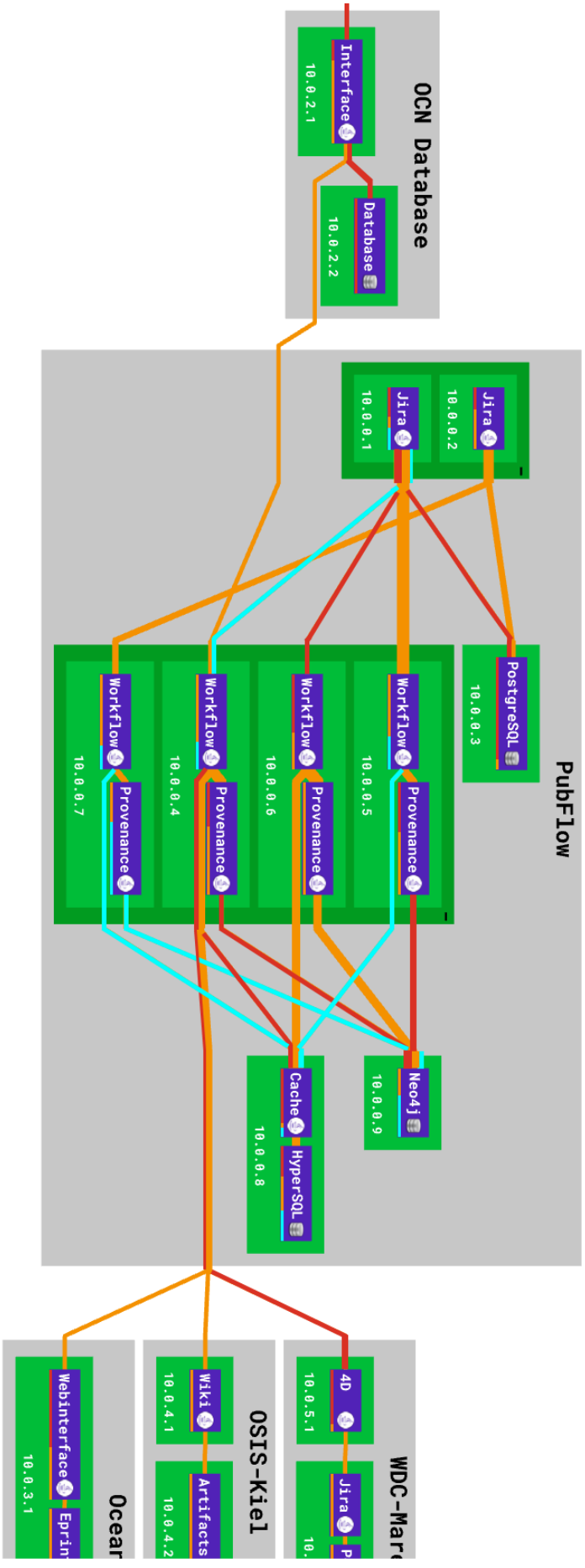

Abbildung B.5: Abbildung 4.7 im Großformat: Eine Veränderung der Szene 4.5 in der die systeminternen Status durch eine unten angebrachte Statusleiste ablesbar sind.

C Evaluationsbogen

C.1 Testvorbereitung:

C.1.1 Proband:

- Alterskategorie [0-10][11-20][21-30][31-40][41-50][51-60][61-70][71-80][81-90][91+]
- Geschlecht: [m / w]
- Beruf:
- höchster akademischer Abschluss:
- ExplorViz erfahren: [j / n]
- Programmiererfahren: [j / n]

C.1.2 ExplorViz Vorführung:

- Was sind Systems, Nodegroups, Nodes, Applications?

- Nodes sind Applikations-gleich

- beide Ansichten berücksichtigen

- auf der Ersten Ebene kann es keine Clazzes geben! Die Foundation ist ein Übertrag der Application

- Was sind components, clazzes?

C.1.3 Einführung in den ModelEditor:

- allgemeine Aufgabe:

- Funktionen der Felder

- Forward Possibility zeigen! = Transitivität der Button

- Kommunikation zeigen und workflow erklären!

- Applicationview erklären, was sind die Felder wie kann ich genau hinzufügen?

C.2 Testausführung:

C.2.1 Qualitativer Test Modelleditor:

Aufgaben:

- Erstelle ein `System` mit dem Namen ***"Grundsystem"***
- Erstelle innerhalb des Grundsystems eine `NodeGroup` mit dem Namen ***"NodeGroup1"***
- Erstelle innerhalb der Nodegroup1 einen `Node` mit dem Namen ***"Node1"***
- Erstelle innerhalb von Node1 eine `Applikation` mit dem Namen ***"TestApp"***
- Erstelle ein zweites System mit dem Namen ***"FortgeschrittenesSystem"***, dieses soll auch eine ***"NodeGroup1"*** mit einem ***"Node1"*** und einer ***"TestApp"*** enthalten
- Verbinde die beiden erstellten `Applikationen` mithilfe einer `Kommunikation`
- Erweitere das ***"Grundsystem"*** um eine weitere `Nodegroup` mit dem Namen ***"NodeGroup2"***
- Benutze die Transitivität der Button um ein weiteres `System` Namens ***"System1"*** zu erstellen, welches wiederum eine ***"NodeGroup1"*** mit einem ***"Node1"*** und einer ***"TestApp"*** enthalten soll
- Füge dem ***"Grundsystem"*** in der ***"NodeGroup2"*** zwei `Nodes` Namens ***"Node42"*** und ***"Node1337"*** hinzu
- Beide gerade erstellten Nodes (***"Node42"*** und ***"Node1337"***) sollen um eine `Applikation` Namens ***"LatestApp"*** erweitert werden
- Füge einer der ***"TestApp"***s eine Komponente Namens ***"Root"*** hinzu
- Diese Komponente soll zwei beliebig benannte Unterkomponenten enthalten
- Füge einer der gerade erstellten Unterkomponenten eine Klasse Namens ***"Implementation"*** hinzu
- Verknüpfe die beiden ***"LatestApp"***s miteinander mit Hilfe einer *Kommunikation*
- Füge insgesammt 6 `Nodes` oder `Applikation`en hinzu, sodass neue Applikationen erstellt werden.
- Von einer Applikation ausgehend, verbinde die restlichen gerade erzeugten Applikationen mit dieser.
- Speichere ein **Screenshot** deines Landscape- und Applicationviews der gerade erstellten Softwarearchitektur in einem Ordner und gib diesen hier an
- :

C.2.2 Interview:

- Wie gut kamst Du mit dem Erstellen der verschiedenen Elemente klar? [0*(schlecht)* - 10*(optimal)*]

- Wie gut konntest Du innerhalb des Landscapeviews navigieren? [0*(schlecht)* - 10*(optimal)*]

- Was würdest Du an der Bedienung ändern?

- Wie gut war die Bedienung der Transitiven Buttons? [0*(schlecht)* - 10*(optimal)*]

- Wie gut war die Bedienung der Kommunikationslinien? [0*(schlecht)* - 10*(optimal)*]

- Was für Anregungen gibt es weiterhin?

C.2.3 Erklärung des ArchConfCheckers

- Was sind Ist- und Soll-Zustand?

- Was sind WARNING und GHOST und ASMODELLED?

- Welche Farbigkeit sind WARNING GHOST und ASMODELLED zugeordnet

- Farbigkeitsbesonderheiten bei Applications erklären

C.2.4 Durchführung der ArchConfCheckEvaluation:

Aufgabe	Antwort	Worturteil
Notiere ein ”AS-MODELLED” System		
Notiere eine GHOST Application		
Notiere eine WARNING Communication		
Notiere eine AS-MODELLED Communication		
Notiere eine WARNING Communication, welche andere überlagert		
Notiere eine GHOST Clazz		
Notiere ein Node der im Ist- und Soll-Zustand gleich ist		
Notiere eine Application die im nicht so konzipiert wurde, wie sie nun vorhanden ist		

Aufgabe	Antwort	Worturteil
Notiere eine Klasse deren Konzeption mit der Umsetzung übereinstimmt		
Notiere eine Componente welche als WARNING eingestut wurde		

D Evaluationsvisualisierung

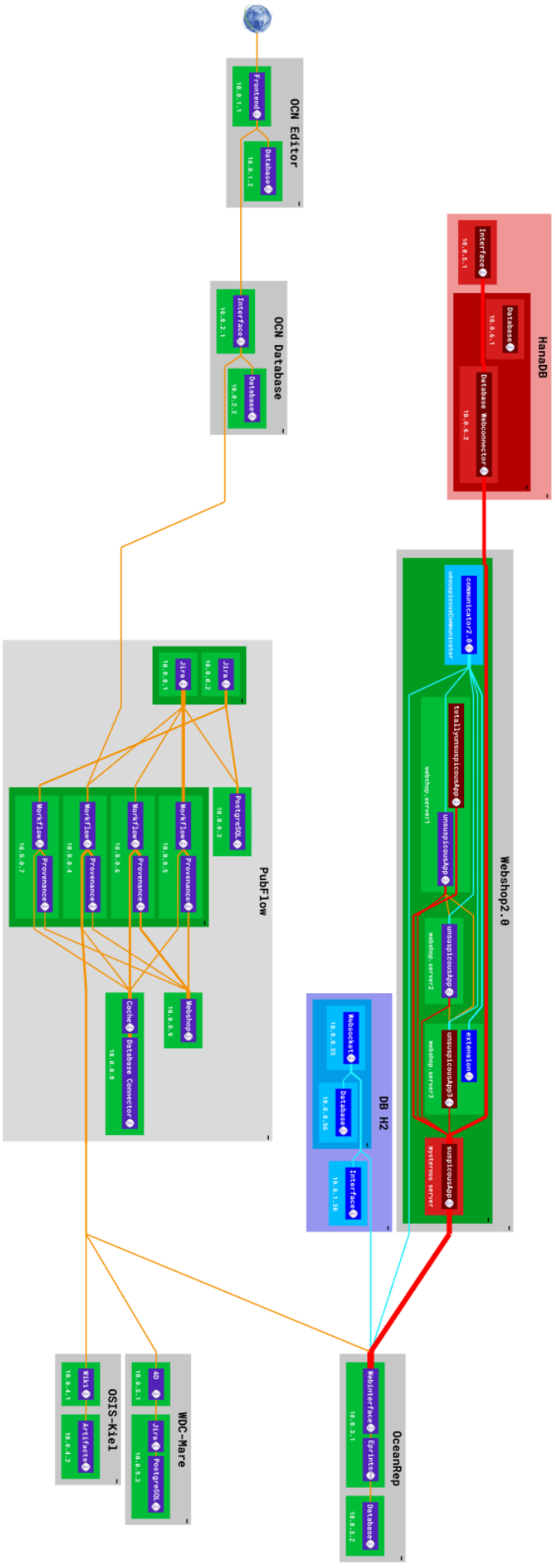

Abbildung D.1: Visualisierung des Architekturkonsormitätsüberprüfers im Rahmen der Evaluation

E Rohdaten der Evaluation

	Proband 1	Proband 2	Proband 3	Proband 4	Proband 5
Aufgabe 01	√	√	√	√	√
Aufgabe 02	√	√	√	√	√
Aufgabe 03	√	√	√	√	√
Aufgabe 04	√	√	√	√	√
Aufgabe 05	√	√	√	√	√
Aufgabe 06	√	√	√	√	√
Aufgabe 07	√	√	√	√	√
Aufgabe 08	√	√	√	√	√
Aufgabe 09	√	√	√	√	√
Aufgabe 10	√	√	√	√	√

Tabelle E.1: Rohdaten der Evaluation des Architekturkonformitätsüberprüfers; √ steht für eine richtige Lösung

Danksagung

Mein **Dank** geht an alle Leute die aktiv oder inaktiv dazu beigetragen haben, diese Arbeit nach solch langer Zeit zu einem Ende zu bringen. Ich weiß dabei gar nicht wo ich anfangen soll.

Ich danke meinen *Eltern*, ohne welche diese lange Odysee nicht finanzierbar oder emotional durchhaltbar gewesen wäre. **Vielen Dank!** Ich danke auch meiner besten und Freundin, *Victoria Zeeb*, welche sich durch herausragende Leistungen im Umgang mit LaTeX und meiner Gefühlslage hervorgetan hat, denn es braucht Mut sich einem Gegner oder Feind in den Weg zu stellen, aber es braucht besonderen Mut sich dem eigenen Freund in den Weg zu stellen und ihm in den Allerwertesten zu treten.

Vielen Dank!

Ich möchte mich weiterhin bei *Anika* und *Tom Geißendörfer* bedanken, ohne jene diese Arbeit ein jähes Ende gefunden hätte. Die letzten Motivationsmeter wurden zum Glück an meinen Elan Empfang weitergeleitet. Ich möchte an dieser Stelle *Tom* auch nochmal sehr herzlich danken für die langen Stunden beim Einrichten und Erklären von Kleinigkeiten. **Vielen Dank.** Dank gebührt auch meinen Betreuern in Kiel, *Christian Zirkelbach* und *Alexander Krause*, vielen Dank für die gute Betreuung und das offene Ohr für Technik-Fragen am Wochenende oder lange nach der Arbeitszeit! **Vielen Dank.**

Dank geht auch an *Hendrik Schön*, der mich immer in seinen 4 Wänden aufnahm, um die beschwerliche Reise von Kiel nach Dresden nicht an einem Stück in ihrer Hin- und Herrichtung bewältigen zu müssen. **Vielen Dank.** Weiterhin seien hier noch *Loki* und *Mino* sowie *Bubi* erwähnt, welche sich immer eines offenen Ohres für meine Probleme rühmen konnten und hoffentlich auch in Zukunft rühmen werden können. **Vielen Dank.** Auch an den *Martin Kühnl*, welcher in der schwersten Zeit für mich und meine Familie da war! **Danke!**

Vielen Dank auch an alle die sich dazu berufen fühlen an dieser Stelle erwähnt zu werden. Vielen Dank für die tolle Zeit @*Emily*, *Konsti*, *Linn*, *Lisa*, *Pauli*, *Timur*, *Sven* und all die anderen Murmler die da noch kommen mögen.**Vielen Dank.**