Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Konzept und Implementierung einer Patternlandkarte zur Navigation durch Patternsprachen

Ridvan Esin

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. Dr. h.c Frank Leymann
Betreuer/in:	Michael Falkenthal, M.Sc.
Beginn am:	26. Februar 2019
Beendet am:	26. August 2019

Kurzfassung

Patterns beschreiben abstrakte Lösungsansätze zu Problemen in einem speziellen Kontext. Dabei verweisen sie auf weitere Patterns, welche anliegende Probleme behandeln. Das daraus resultierende Netzwerk an Pattern definiert eine Patternsprache. In der Praxis werden Patterns aus verschiedenen Sprachen in Kombination verwendet, um Probleme zu lösen. Patternsprachen werden in der Regel isoliert voneinander entwickelt. Dies hat zur Folge, dass sich Verlinkungen zu Pattern aus anderen Sprachen nur implizit, d. h. über spezielles Vorwissen, finden lassen.

Das Ziel dieser Arbeit gliedert sich in zwei Teile. Im ersten Teil werden exemplarisch Cross Language Relations aus den Patternsprachen *Enterprise Integration*, *Cloud Computing*, *Enterprise Application Architecture* und *Internet of Things* abgeleitet. Cross Language Relations sind Verlinkungen von Pattern aus einer Sprache zu Patterns aus einer anderen Sprache. Diese werden anhand der in der Literatur verfügbaren Daten ausfindig gemacht.

Der zweite Teil der Arbeit behandelt die Navigation durch Patternsprachen. Dabei werden die Daten der Patternsprachen sowie die Cross Language Relations auf Basis des *Semantic Webs* als Tripel gespeichert. Über die Prinzipien des Semantic Webs ist eine dezentrale Datenverwaltung möglich. Die Daten liegen verstreut im *world wide web* und verlinken sich gegenseitig. Ein Client lädt die Daten und stellt sie als Netzwerkgraphen dar. Dieser zeigt eine Patternsprache mitsamt ihren Patterns und deren Relationen zueinander. Diese Visualisierung ermöglicht die Navigation. Filter sorgen für bessere Übersichtlichkeit, indem sie unerwünschte Patterndaten ausblenden.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Nach Alexander et al. [Ale78] beschreibt ein Pattern zum einen ein wiederkehrendes Problem in einem speziellen Kontext und bietet zum anderen eine Lösung an, welche auf das Problem angewandt werden kann. Oftmals gibt es Probleme, welche aus mehreren Teilproblemen bestehen. Zu jedes dieser Teilprobleme kann es Patterns geben, welche eine Lösung dafür beschreibt. Für dieses große Problem besteht die Lösung aus dem Verbund der einzelnen Pattern. Verlinkungen innerhalb der Pattern finden sich in der Literatur meist unter dem Punkt „siehe auch“, „related patterns“, o.Ä.

Patterns und deren Verlinkungen untereinander bilden ein Netzwerk, welches man als eine Patternsprache versteht [Ale78]. Eine Patternsprache ist eine Ansammlung von sich gegenseitig verlinkenden Patterns des gleichen Themengebiets. Die Einordnung der Pattern in eine Patternsprache geschieht i.d.R. über den Autor der Patterns. Formal lässt sich eine Patternsprache als gewichteter Graph darstellen, dessen Knoten die einzelnen Patterns und die Kanten die Relationen der Patterns untereinander abbilden [FBL18].

Da Patternsprachen zumeist unabhängig voneinander erstellt werden, kommt es vor, dass einzelne Patterns in unterschiedlichen Sprachen sehr ähnlich zueinander sind oder aber, dass Referenzen zwischen Patterns verschiedener Patternsprachen notwendig sind, um die in den einzelnen Patternsprachen erfassten Lösungskonzepte ganzeinheitlich anzuwenden. Beides zeigt dementsprechend auf, dass Verlinkungen über Patternsprachen hinweg notwendig sind, diese jedoch durch die voneinander unabhängige Erstellung von Patternsprachen häufig in der Literatur nicht finden lassen.

In dieser Arbeit wird das Konzept einer Patternlandkarte sowohl aus konzeptioneller Sicht, als auch aus Sichtweise der Implementierung nähergebracht. Die Patternlandkarte zeigt die Zusammenhänge von Pattern über die eigene Patternsprache sowie über Patternsprachen hinweg dar und ermöglicht die Navigation. Unter Navigation verstehen wir hier die Fähigkeit, von einem Pattern, Information über ein anderes verlinktes Pattern zu bekommen.

Weiter beschäftigt sich diese Arbeit mit der Erarbeitung der Verlinkungen von Pattern über Patternsprachen hinweg. Die Verlinkungen der Patterns verschiedener Patternsprachen ist nicht trivial. Dies bedeutet, dass die Verlinkung auf Basis der Pattern-Kontexte argumentiert werden muss.

Gliederung

Diese Arbeit gliedert sich wie folgt:

Kapitel 2 – Grundlagen und Related Work

Diese Kapitel geht auf die Grundlagen und verwandte Arbeiten ein, welche als Basis für diese Arbeit dienen.

Kapitel 3 – Cross Language Relations

Cross Language Relations sind Verlinkungen über Patternsprachengrenzen hinweg. In diesem Kapitel werden diese Verlinkungen anhand von vier Patternsprachen identifiziert.

Kapitel 4 – Implementierung

Die Implementierung kümmert sich um den praktischen Teil der Arbeit. Aufbau und Modellierung der Daten, sowie die Implementierung einer Patternlandkarte zur Navigation durch Patternsprachen finden sich in diesem Kapitel.

Kapitel 5 – Zusammenfassung und Ausblick

Das letzte Kapitel fässt die Arbeit zusammen und gibt mögliche Punkte zur weiteren Arbeit an.

2 Grundlagen und Related Work

In diesem Kapitel werden verwandte Arbeiten beleuchtet, auf welche diese Arbeit aufbaut.

Grundstein sind die Pattern und Patternsprachen, wie sie von Alexander et al. definiert wurden. Ein Pattern, so Alexander et al., beschreibt eine Lösung eines immer wieder auftretenden Problems in einem bestimmten Kontext. Die Lösung widmet sich dabei dem Kern des Problems, sodass die Lösung in allen Fällen des Problems angewendet werden kann [Ale78]. Ursprünglich als Konzept für die Architektur von Städten und Gebäude gedacht, haben sich Patterns auch in anderen Bereichen etabliert. Darunter fällt bspw. auch die Architektur von Softwareanwendungen. Patterns finden sich dort z.B. in den Patternsprachen Enterprise Application Architecture [Fow02] oder den Cloud Computing Patterns [FLR+14].

Pattern verweisen in ihren Dokumenten auf weitere Pattern, welche in Verbindung mit diesem stehen. Das Konzept einer Patternsprache umfasst mehrere dieser Patterns, die miteinander verknüpft sind. Es entstehen Netwerke von Patterns, die typischerweise in Kombination verwendet werden. Zu einem Problem findet sich ein Pattern, welches eine Lösung bereitstellt. Verlinkte Pattern gehen Probleme an, welche mit dem ursprünglichen Problem in Verbindung stehen. Über dieses Netzwerk wird somit eine Navigation durch eine Reihe von Problemen ermöglicht [Ale78].

Dieses Kapitel ist wie folgt aufgebaut: Eine formale Beschreibung von Patternsprachen findet sich in Abschnitt 2.1. Abschnitt 2.2 befasst sich mit einer früheren Version von Patternpedia. Die Konzepte des Semantic Webs werden in Abschnitt 2.3 erläutert.

2.1 Nature of Patternlanguages

Alexander et al. [Ale78] definieren eine Patternsprache als verlinkte Pattern. Die Definition findet sich in natürlicher Sprache, d.h. in Textform. Falkenthal et al. [FBL18] extrahieren diese Charakteristika und überführen sie in eine formale mathematische Definition. Sie geben schrittweise verfeinerte Definitionen an, welche das Model erweitern, dem Original von Alexander et al. aber treu bleiben. Über die Formalität lassen sich Aussagen zur Natur von Patternsprachen machen. Weiter bietet es die Möglichkeit bspw. den Verbund zweier Sprachen mathematisch auszudrücken. Im Folgenden werden die relevanten formalen Definitionen einer Patternsprache erklärt.

Ein Patterngraph $\mathcal{G}$ ist nach [FBL18] definiert als:

$$\mathcal{G} = (N, E)$$

Dabei ist N die Menge der Patterns und E die Menge der Relationen der Patternsprache. Abbildung 2.1 zeigt einen Beispiel-Patterngraphen. Die Patternmenge ist hier $N = \{P_A, P_B, P_C, P_D, P_E, P_F, P_G, P_H, P_I, P_J\}$ und die Kantenmenge entsprechend $E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}\}$.

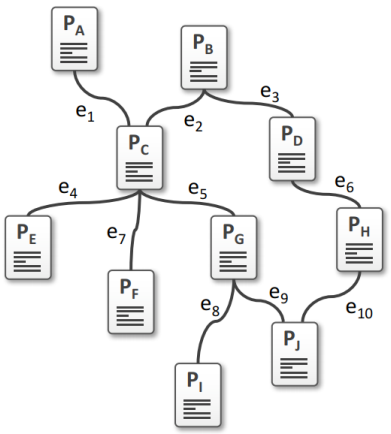

Abbildung 2.1: Einfacher Patterngraph mit Pattern und Relationen aus [FBL18]

Diese Definition zeigt einen wichtigen Aspekt der Modellierung einer Patternsprache auf. In der Literatur finden sich Pattern als einzelne Dokumente. Verlinkungen zu weiteren Pattern finden sich innerhalb der Dokumente als Teil des Patterns und nicht als gesonderte Elemente. Der Patterngraph nach obiger Definition behandelt beides als getrennte Konzepte. Vorteil der Trennung ist, dass Patterns bei Hinzunahme neuer Relationen nicht überarbeitet werden müssen. Die Pattern bleiben unberührt, die Verlinkung kann der Sprache unabhängig davon zugeordnet werden.

Gehen wir von einem Pattern entlang der Kanten zu anderen Pattern, haben wir einen Pfad an relevanten Patterns durch den Graphen. Suchen wir uns bspw. zwei Pattern im Graph, welche Teil der Lösung eines Problems sind, finden sich auf dem Pfad dieser beiden Pattern weitere Pattern, die zur Lösung beitragen.

Handelt es sich bei den Elementen der Kantenmenge E um gerichtete Kanten, hat die Richtung eine Bedeutung. Haben wir bspw. eine Kante $e = (n_1, n_2)$ mit $e \in E \quad n_1, n_2 \in N$, sagt die Richtung folgendes aus: das Pattern n_1 gilt als größeres Pattern und muss vor n_2, dem kleineren Pattern, angewendet werden. Der Patterngraph ist dann ein gerichteter Graph [FBL18].

Durch gerichtete Kanten bekommen Relationen eine implizite Semantik. In Patternsprachen finden sich jedoch oft Verweise wie „siehe auch“ o.Ä. Um solche expliziten semantische Aussagen in Kanten zu ermöglichen, erweitern Falkenthal et al. [FBL18] die Definition wie folgt:

$$\mathcal{G} = (N, E, W)$$

Durch die Hinzunahme einer Gewichtsmenge W, können Kanten einer expliziten Bedeutung zugewiesen werden. Als Gewichte können beliebige Werte verteilt werden, bspw. $w \in W, w = siehe\ auch$. Alternativ lassen sich die Gewichte auf eine domainspezifische Menge beschränken damit bspw. Programme die Graphen automatisch travesieren und verarbeiten können [FBL18].

Durch die Erweiterung um Typen bildet die formale Definition die Natur von Patternsprachen vollständig ab, so Falkenthal et al. Es ergibt sich:

$$\mathcal{G} = (N, E, W, \mathfrak{D}, \alpha, \beta)$$

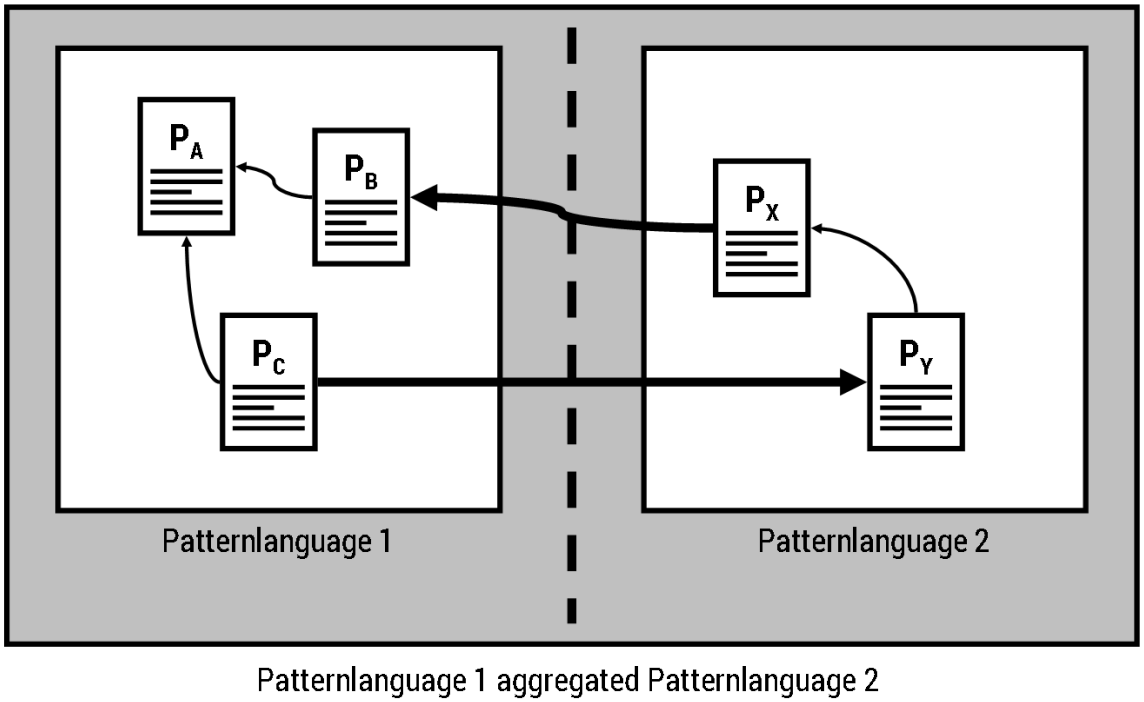

Abbildung 2.2: Zwei Patternsprachen werden über den Aggragator verbunden nach [FBFL15]

α weißt den Gewichten eine Teilmenge aller Domains zu und über β werden Kanten typspezifische Beschreibungen zugewiesen. Die Beschreibungen können willkürliche Werte beinhalten, z.B. eine textuelle Erklärung, inwiefern die beiden umfassten Pattern zusammenhängen [FBL18].

Über die genannten Definitionen lassen sich keine Relationen zu Patterns aus anderen Patternsprachen ziehen. Grund ist die Definition der Kantenmenge. Für einen Link zu einem Pattern aus einer anderen Patternsprache wäre

$$e = (n_{PL1}, n_{PL2}) \quad n_{PL1} \in N_{PL1}, n_{PL2} \in N_{PL2} \quad \lightning$$

Hier liegt ein Widerspruch vor, da eine Kante zwei Knoten aus der Knotenmenge desselben Graphen benötigt. In der obigen Formel ist n_{PL2} ein Pattern aus einer anderen Patternsprache und damit aus einer anderen Knotenmenge. Um diese Verlinkung dennoch zu ermöglichen, führen Falkenthal et al. [FBL18] den Patternsprachen-Aggregator ein. Bei diesem handelt es sich um einen Operator, welcher zwei Patterngraphen unter Zunahme einer Kantenmenge, zu einem neuen Patterngraphen verbindet. Zwei Patterngraphen werden nach [FBL18] wie folgt aggregiert:

$$\mathcal{G}_3 = \mathcal{G}_1 \odot_{\mathcal{E}} \mathcal{G}_2$$

Die Menge $\mathcal{E}$ definiert die Relationen eines Patterns zu Pattern aus anderen Sprachen, d.h.:

$$e = (n_1, n_2) \quad n_1, n_2 \in N_1 \cup N_2$$

Dabei sind N_1 Patterns aus der einen und N_2 Patterns aus der anderen Patternsprache.

Abbildung 2.2 zeigt die Aggragation zweier Patternsprachen. Die gestrichelte Linie zeigt die Patternsprachengrenzen auf. Durch den Aggregator können jedoch Links über die Grenze hinweggezogen werden. Dies wird durch die dickeren Pfeile dargestellt. In dieser Abbildung verweist Pattern P_C aus *Patternlanguage 1* auf Pattern P_Y aus *Patternlanguage 2* sowie P_X aus *Patternlanguage 2* zu P_B aus *Patternlanguage 1*.

2.2 PatternPedia

Patternsprachen sind lebendige Systeme. Das bedeutet neue Patterns und Relationen können im Nachhinein hinzugefügt werden. Die Dokumentation von Patternsprachen findet sich jedoch oft in Printmedien, also Bücher, Artikel und Papers. Bei diesen ist das Problem, dass sie nicht ohne Neupublikation die neuen Inhalte umfassen können. Patternsprachen werden teilweise zusätzlich auch online, meist in reduzierter Form, veröffentlicht. Diese stehen jedoch für sich. Das bedeutet, dass Relationen zu Pattern aus anderen Sprachen keine Erwähnung finden.

Fehling et al. [FBFL15] gehen auf diese Problematik ein und stellen mit *PatternPedia* eine Tool Chain für die kollaborative Dokumentation von Patterns bereit. Die genauen Konzepte von *PatternPedia* werden in diesem Abschnitt näher gebracht.

Die PatternPedia Tool Chain unterstützt die Identifizierung, Abstrahierung und Anwendung von Patterns als eine Reihe von Prozessen. Anwender bekommen, passend zu ihren Tätigkeiten, verschiedene Rollen, um diese auszuführen. Als Beispiel dient die Rolle des *Pattern Authors*. Eine Gruppe *Pattern Authors* schaut sich mögliche Pattern-Kandidaten an und formuliert bzw. verfeinert diese zu vollwertigen Patterns.

Die abstrakte Architektur von PatternPedia findet sich in Abbildung 2.3. *Domain Experts* sammeln alle möglichen Arten an Domain-Informationen, welche für die Extrahierung von Patterns relevant sein könnten. Aus diesen Informationen werden Lösungen extrahiert und auf Pattern-Kandidaten untersucht.

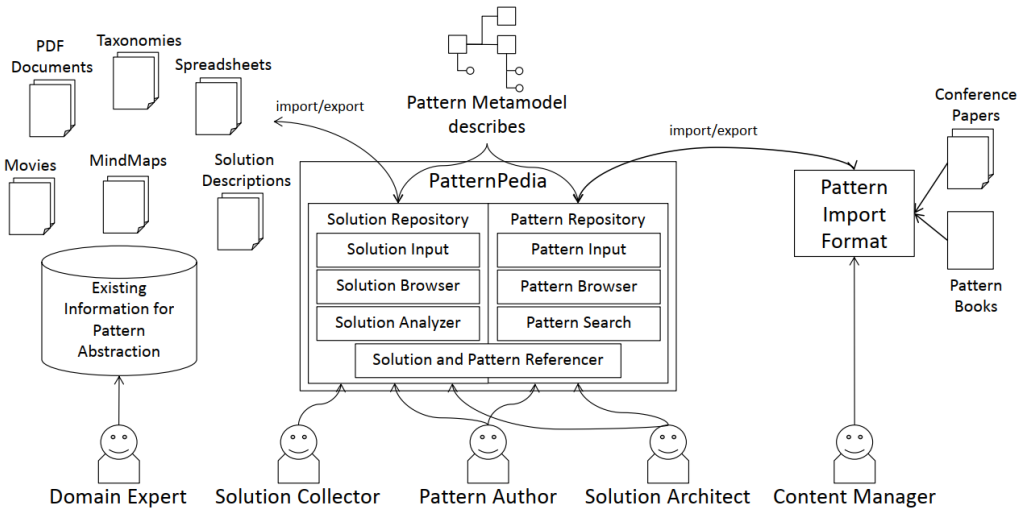

Abbildung 2.3: Abstrakte Architektur und funktionale Übersicht von *PatternPedia* [FBFL15]

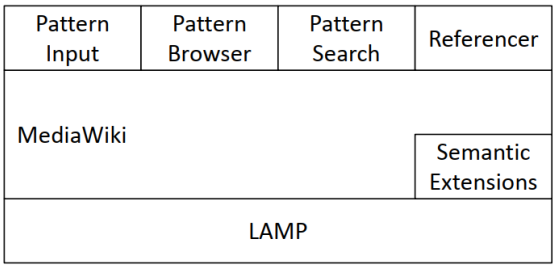

Abbildung 2.4: Software-Stack von *PatternPedia* [FBFL15]

Im Kern beinhaltet PatternPedia die folgenden zwei Dinge: *(1)* ein erweiterbares Metamodell von Patterns und *(2)* eine ebenso erweiterbare Tool Chain.

Das Metamodell dient in erster Linie für die MediaWiki Plattform (https://www.mediawiki.org). Damit die Tool Chain Unterstützung mit sich bringt, müssen die Daten in Form dieses Metamodells vorliegen. Das bedeutet, Pattern und Informationen, die nicht in diesem Format vorhanden sind, müssen erst in ein geeignetes Model tranferiert werden. Die Semantik der Daten wird daher modifiziert und könnte sich von der ursprünglichen Bedeutung unterscheiden.

Abbildung 2.4 zeigt den verwendeten Software-Stack für PatternPedia an. Basis ist die MediaWiki, welche eine Plattform mit kollaborativen Funktionen bspw. für die Bearbeitung von Dokumenten bereitstellt. Weitere Basis ist die Verwendung von Linux, Apache, MySQL und PHP (LAMP) auf der Serverseite. Wichtig bei dieser Umsetzung ist, dass es sich um einen zentralen Ansatz handelt. Die Patterns werden alle in einem zentralen Repository verwaltet. Das bedeutet insbesondere, dass sie nicht verstreut im World Wide Web liegen, sondern sich in diesem Repository befinden. Wird das Repository abgeschaltet bzw. entfernt, sind alle enthaltenen Patterns nicht mehr zugänglich. Dies stellt eine wichtige Motivation für diese Arbeit dar, welche in Kapitel 4 genauer ausgearbeitet wird.

2.3 Semantic Web

Ein Großteil der Daten, welche im World Wide Web verfügbar sind, sind von Menschen lesbar. Eine Website mit HTML, welche eine Liste von bspw. Pattern bereitstellt, kann von einem Menschen erfasst werden. Die Semantik der Daten lässt sich anhand des Kontextes ableiten. Maschinen hingegen können diesen Kontext nicht ableiten. Verarbeitet ein Programm dieselbe Website, kann sie nur Aussagen über deren generellen Aufbau machen. Also bspw. dass sich auf dieser eine Liste mit Inhalten befindet. Über die Bedeutung der Inhalte kann das Programm ohne weiteren Input keine Aussagen treffen.

Die Idee des Semantic Webs nach Berners-Lee et al. [BHL01] ist es, dass Daten in einer Form vorliegen, sodass Software-Agenten diese auch semantisch verarbeiten können. Dies bedeutet, dass das Semantic Web eine Sprache bereitstellen muss, welche alle Daten und Fakten beschreiben kann, unabhängig von deren Struktur und Inhalt. Eine solche Sprache wird über die folgenden drei Komponenten ermöglicht: *(1)* eXtensible Markup Language (XML), *(2)* Resource Description Framework (RDF) sowie *(3)* Ontologien. Über XML lassen sich beliebige Strukturen anlegen. Diese

Strukturen sagen jedoch nichts aus. Die Bedeutung kommt in Kombination mit RDF zustande. RDF erlaubt beliebige Aussagen in Form von Tripeln bestehend aus *Subjekt*, *Prädikat* und *Objekt*. Die Tripel Struktur kann in XML definiert werden [BHL01].

Ein Tripel besteht immmer aus einem Subjekt, einem Prädikat sowie einem Objekt und stellen einen Fakt dar. Darüber lassen sich alle Dinge in folgender Form beschreiben: Bestimmte Dinge haben Eigenschaften mit bestimmten Werten [BHL01].

Subjekte und Objekte, auch als Konzepte bezeichnet, werden über einen Universal Resource Identifier (URI) eindeutig identifiziert. D.h. jedes Konzept benötigt eine eigene URI. Eine URI kann in Form einer Uniform Resource Location (URL) angegben werden, muss sie jedoch nicht. URLs dienen im Kontext von Webseiten zur Verlinkung zu weiteren Webseiten. Für die Konzepte kann die URL zur Lokalisierung verwendet werden. Ein Beispiel: Der Student Max Mustermann wird über `http://purl.org/student/max-mustermann` eindeutig identifiziert. In diesem Beispiel ist der Identifier ist eine URL, welche auf die Ressource im World Wide Web hinweisen kann. URIs identifizieren Ressourcen eindeutig. Dabei ist es egal, wo sich die Ressourcen im einzelnen tatsächlich befinden. Wird an unterschiedlichen Orten dieselbe URI verwendet ist dasselbe Ding damit gemeint [BHL01]. Dieser Sachverhalt zeigt einen Vorteil des Semantic Webs auf: Konzepte können verstreut im Internet liegen. Es gibt kein zentrales Repository, welches alle Konzepte enthält und verwaltet werden muss [BHL01].

Die dritte Komponente des Semantic Webs sind Ontologien. Grundlegend lässt sich sagen, dass eine Ontologie eine Ansammlung von Informationen über Dinge umfasst. Sie sagt aus, mit welchen Arten von Dingen wir es zu tun haben und wie sie miteinander in Verbindung stehen. Wichtig ist auch, dass es sich bei einer Ontologie nicht um ein Datenformat handelt, sondern um die Repräsentation von Wissen. Für die Erstellung und Verwendung von Ontologien gibt es die *OWL Web Ontology Language* des World Wide Web Consortiums (W3C). Diese erweitert RDF um weitere Definitionen, die dazu verwendet werden können, eigene Ontologien zu definieren [W3C04].

Über RDF lassen sich Aussagen über Individuen treffen. Durch Ontologien können jedoch Aussagen über Klassen von Dingen gemacht werden [W3C04]. Darin liegt die Stärke von Ontologien. Ein Beispiel: Wir definieren eine Klasse von Enterprise Integration Patterns und beschreiben, wodurch sich diese Patterns auszeichnen. Bspw. treffen wir die Aussage, dass Enterprise Integration Patterns einen Namen haben können. Eine Instanz dieser Klasse ist ein konkretes Pattern und sagt explizit aus, dass sie zur Klasse der Enterprise Integration Patterns gehört. Ein Software-Agent kann aus dieser Aussage automatisch schließen, dass diese Ressource ein konkretes Enterprise Integration Pattern ist und einen Namen besitzen kann.

3 Cross Language Relations

Patternsprachen werden meist unabhängig voneinander entwickelt. Einzelne Pattern aus mehreren Sprachen können und werden in der Praxis jedoch in Kombination verwendet. Bspw. können die Cloud Computing Patterns zusammen mit den Enterprise Integration Patterns für die Architektur eines verteilten Systems verwendet werden. Formal werden die beiden Patterngraphen unter Zunahme einer Kantenmenge zu einem neuen Patterngraphen aggregiert. Die Kantenmenge umfasst Relationen von Pattern der einen Sprache zu Pattern der anderen. Diese Art von Kanten bezeichnen wir im Rahmen dieser Arbeit als *Cross Language Relations* oder auch *CLRs*.

Formal lassen sich CLRs erstellen, in der Praxis werden sie jedoch nicht immer explizit hervorgehoben. Durch die unabhängige Entwicklung von Patternsprachen werden andere Patternsprachen teilweise, jedoch nicht vollständig berücksichtigt. Im günstigsten Fall finden sich minimale Erwähnungen zu anderen Sprachen, welche nicht immer genauer ausgeführt werden. Das bedeutet durch den Aggregator gibt es die Möglichkeit Sprachen zu verbinden, die konkreten Kanten fehlen jedoch und müssen ergänzt werden.

Dieses Kapitel beschäftigt sich somit um das exemplarische Herausfinden und Dokumentieren der Relationen zwischen Patternsprachen. Der Abschnitt 3.1 beschreibt das allgemeine Vorgehen für die Erstellung der CLRs. Weiter listet der Abschnitt 3.2 die Pattern der untersuchten Patternsprachen auf und geht auf die CLRs im Detail ein.

3.1 Vorgehen

Dieser Abschnitt geht darauf ein, wie man Cross Language Relations zu den einzelnen Pattern findet. Verweise auf Patterns von anderen Sprachen finden sich in der Literatur entweder durch genaues Hinsehen und Kombinieren oder die Autoren selbst verweisen auf Ähnlichkeiten. Für Ersteres werden verschiedene Patternsprachen durchforstet und auf Gemeinsamkeiten oder Kombinationsmöglichkeiten geachtet. Bei der zweiten Variante werden die gegebenen Pattern genauer untersucht und die Verlinkung genauer ausgeführt. Meist finden sich nur Kommentare wie *„Pattern A eignet sich hierfür“* jedoch nicht, warum gerade dieses Pattern für das aktuelle Pattern relevant ist. Die Erklärung bzw. genauere Ausführung wurde für diese Fälle entsprechend ausgearbeitet.

Patterns werden nicht erfunden, sondern entdeckt [Fow02]. Dies liegt daran, dass Patterns bewiesene Lösungen aus der Praxis darstellen. Das hat zur Folge, dass es gleiche Pattern, möglicherweise unter anderen Namen oder Gesichtspunkten, in unterschiedlichen Sprachen gibt. Unter verschiedenem Gesichtspunkt verstehen wir bspw. das genauere Awendungsfeld des Patterns. Befinden wir uns bei einem Pattern bspw. im Kontext von Messaging, werden die Messaging-Aspekte des Patterns genauer betrachtet. Die Verwendung in anderen Gebieten wäre evtl. möglich, ist für das konkrete

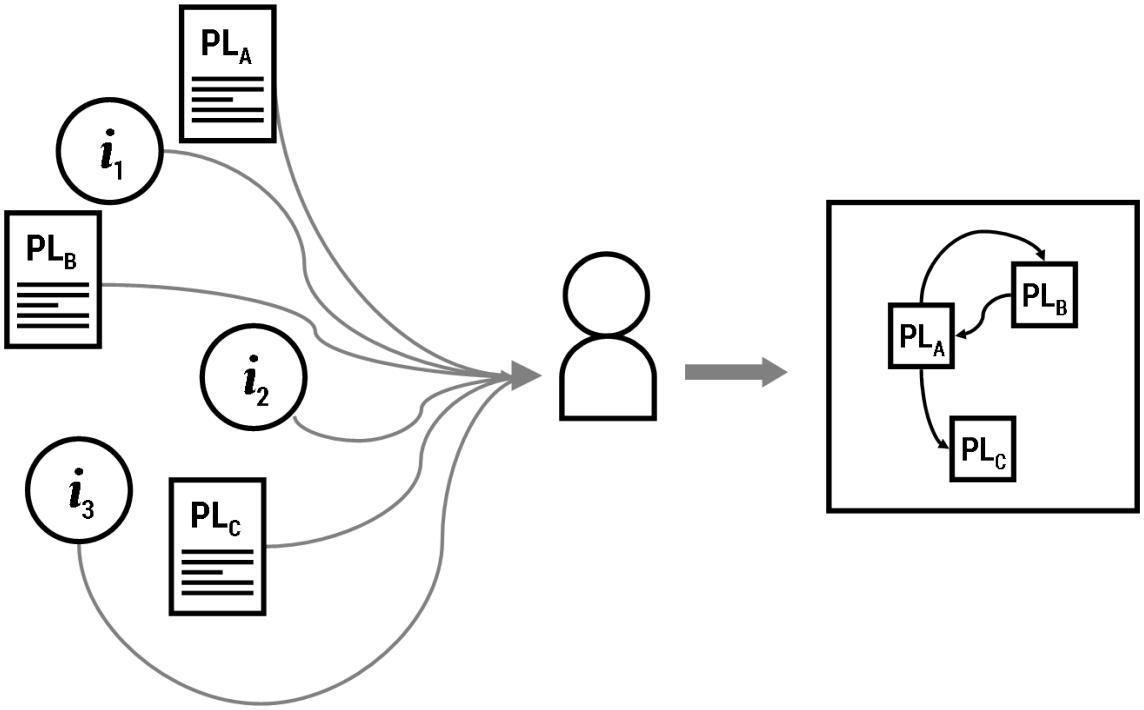

Abbildung 3.1: Abstrakte Vorgehensweise beim Identifizieren der *CLRs*

Pattern aber nicht von Relevanz. Aus diesem Grund kann ein Pattern eine ähnliche Lösung für das andere Gebiet beschreiben. Eine Verlinkung von diesen beiden genannten Pattern wird nur dann erarbeitet, wenn die Gebiete zueinander passen.

Es gibt Fälle, bei denen sich die Pattern in ihrerer Granularität unterscheiden. Zum Beispiel beschreibt ein Pattern eine allgemeinere Lösung als ein anderes ähnliches Pattern. In den Cloud Computing Patterns findet sich das *Idempotent Processor* Pattern, während sich bei den Enterprise Integration Patterns das *Idempotent Receiver* Pattern findet. Beide Pattern behandeln eine Lösung, wie man eine Komponente entwickelt, welche mehrere gleiche Nachrichten, also Duplikate, ohne Probleme empfangen kann. Die Pattern haben dieselbe Aufgabe und befinden sich beide im Messaging Kontext. Der Unterschied ist jedoch, dass sich der *Idempotent Processor* auch der Datenkonsistenz bei *Eventual Consistency* widmet. Formal hat eine Relation immer eine Richtung, weswegen diese Unterscheidung berücksichtigt werden muss. Die Richtung ist für die Identifizierung von Patternpfaden wichtig, welche in Kapitel 2 behandelt wurden.

Feingranulare Pattern verweisen auf grobe Pattern, jedoch nicht zwingend andersherum. Bspw. können Enterprise Integration Patterns Relationen zu den Gang of Four Pattern haben, andersherum gibt es keine Notwendigkeit dafür. Das liegt daran, dass die EIP Lösungsansätze speziell für verteilte Anwendungen im Kontext von Messaging bereitstellen, während die GoF Patterns für objektorientierte Systeme ausgelegt sind. Aus Sicht eines GoF Patterns spielt die Verwendung der Lösung im Messaging Kontext keine Rolle. Diese Relationen verweisen in dieser Arbeit deshalb von den EIP zu den Pattern von Gamma et al.

Eine Cross Language Relation wird, sofern ein Autor der Patternsprache nichts genaueres aussagt, nur dann hinzugefügt, wenn die Patterns semantische Ähnlichkeiten haben. Die Ähnlichkeit wird aus eigener Sicht beurteilt. Haben bspw. Patterns irgendetwas mit Datenbanken zu tun, müssen sie noch keine Relation zueinander haben, nur weil sie sich diesen Aspekt teilen. Eine andere Sicht könnte dies anders begründen und eine Relation hinzufügen. Hinzu kommt, dass eine Patternsprache ein lebendiges System ist, d.h. weitere Patterns und Relationen können hinzugenommen werden [FBL18]. Die CLRs vollständig zu identifizieren ist bei einem wachsenden System nicht möglich. Aus diesem Grund befasst sich diese Arbeit nur exemplarisch mit Cross Language Relations.

Die Abbildung 3.1 stellt dar, wie die Cross Language Relationen gewonnen werden. Verschidene Patternsprachen PL_X und verschiedene sonstige Informationen i_X dienen als Input. Ein Mensch geht diese Daten durch und schließt auf verschiedene Relationen. Der Mensch steht hier in der Rolle des *Experten*, welcher die notwendige Fähigkeiten für das Ableiten der CLRs verfügt. Als Output verbinden die Relationen einzelne Pattern aus verschiedenen Sprachen und somit die Sprachen selbst.

Der Fokus dieser Arbeit liegt daher auf Relationen, welche von den Autoren der Patternsprachen gezogen wurden und auf jene Relationen, bei denen die Pattern eine ausreichende semantische Ähnlichkeit haben.

3.2 Patternsprachen

Im Rahmen dieser Arbeit wurden die folgenden Patternsprachen auf CLRs untersucht:

1. *Enterprise Integration Patterns* von Hohpe und Woolf [Gre04]
2. *Cloud Computing Patterns* von Fehling et al. [FLR+14]
3. *Patterns of Enterprise Application Architecture* von Fowler [Fow02]
4. *Internet of Things Pattern* von Reihnfurt et al. [RBF+16] [RBF+17] [RBF+19]

Die Erhebung der Cross Language Relations basiert nach den Anmerkungen im vorherigen Abschnitt.

Wichtig ist, dass bei der Untersuchung iterativ vorgegangen wurde. D.h. die Enterprise Integration Patterns dienen als Basis. Für diese Patternsprache wurde nicht explizit nach CLRs gesucht. Stattdessen konzentriert sich dieser Abschnitt auf die Verlinkungen der Autoren. Der nächste Abschnitt behandelt die Cloud Computing Patterns. Zusätzlich zu den Verlinkungen der Autoren werden hier explizit Relationen zu den Enterprise Integration Patterns angemerkt. Jede weitere Sprache wird auf Relationen zu vorherigen Sprachen untersucht.

Die Abbildung 3.2 stellt diesen iterativen Vorgang dar. Die einzelnen Kreise entsprechen dem Betrachtungsradius. Die Enterprise Integration Patterns dienen als Basis, die weiteren Sprachen betrachten alle Pattern der vorherigen Sprachen.

Autoren der Patternsprache nehmen teilweise Verlinkungen zu anderen Sprachen vorweg. Auf diesen Sachverhalt wird an entsprechender Stelle verwiesen. Die weiterführenden Abschnitte behandeln die einzelnen Patternsprachen.

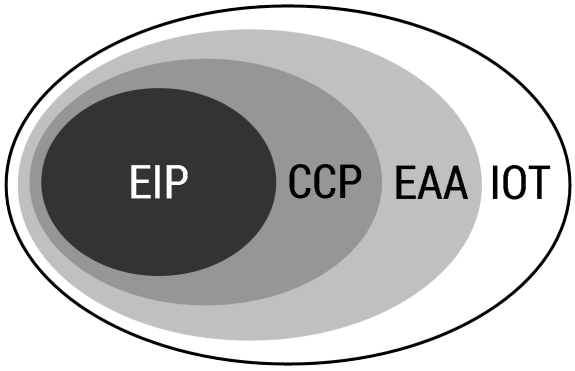

Abbildung 3.2: Betrachtungsradius beim Identifizieren der *CLRs*

3.2.1 Enterprise Integration Patterns

Enterprise Integration beschäftigt sich mit der Frage, wie unabhängige Anwendungen verbunden bzw. integriert werden können, sodass diese zusammenarbeiten können. Unabhängige Anwendungen sind Anwendungen, die selbständig laufen, d.h. ohne Abhängigkeiten zu anderen Anwendungen. Genauer behandelt es die Integration über Messaging, welches zuverlässige, asynchrone program-to-program Kommunikation über Messages erlaubt. Messages sind Datenpakete bestehend aus Header, Metadaten für das Messaging System, und Body, dem Payload bzw. die Daten der Nachricht. Programme sind über Channels logisch verbunden, auf denen die Nachrichten laufen. Die Message-oriented Middleware ermöglicht die Verwendung von Messaging [Gre04].

Hohpe und Woolf [Gre04] beschreiben mit ihren Patterns eine abstrakte Sicht auf Enterprise Integration. D.h. sie sind nicht auf spezielle Messaging-Anbieter fokusiert, sondern gelten übergreifend. Die Patterns sind fokusiert auf Integration über Messaging, sprechen aber auch andere Integrationsmöglichkeiten an.

Dieser Abschnitt geht auf die Enterprise Integration Patterns ein. Jedes der Pattern wird zusammengefasst. Für manche Pattern gehen Hohpe und Woolf [Gre04] auf Verbindungen zu Patterns aus anderen Sprachen ein, welche entsprechend weiter ausgeführt werden. Diese Sprache dient als Basis bzw. Einstiegspunkt für die weiteren Abschnitten, weswegen keine weiteren Verlinkungen zu anderen Sprachen untersucht werden.

File Transfer

Der Informationsaustausch von Anwendungen basiert auf Dateien. Dateien werden in Intervallen von einer Anwendung erzeugt und von einer anderen eingelesen. Der Aufbau bzw. die Struktur der Dateien muss abgesprochen werden [Gre04].

Shared Database

Die Daten der zu integrierenden Anwendungen werden gemeinsam auf eine geteilten Datenbank gespeichert. Hierbei kann eine relationale Datenbank verwendet werden, welche ein standardisiertes Format bietet, im Gegensatz zu einfachen Files. Die Herausforderung dieses Integrationsstils ist das Design der Datenbank, sodass alle Anwendungen sie verwenden können [Gre04].

Remote Procedure Invocation

Über *Remote Procedure Invocation* lassen sich Funktionen mit Parameterdaten einer anderen Anwendung aufrufen. Die Kommunikation erfolgt über bereitgestellte Schnittstellen. Die einzelnen Komponenten teilen sich keine Datenstruktur, sie sind auf Grund von remote calls stark aneinander gekoppelt [Gre04].

Messaging

Messages, also Datenpakete, werden von einer Anwendung an einen Message Bus gesendet. Interessierte Anwendungen können sich die Messages bei Bedarf und nach eigenem Tempo aus dem Bus entnehmen. Anwendungen sind nicht direkt, sondern über den Message Bus verbunden und sind somit lose gekoppelt [Gre04].

Message Channel

Ein *Message Channel* verbindet zwei Anwendungen und ermöglicht eine Kommunikation über Messaging. Die Anwendungen kennen sich untereinander nicht, kennen jedoch den entsprechenden Channel. Wir erreichen eine lose Kopplung durch die Aufteilung von Sender und Empfänger [Gre04].

Message

Informationen werden in Form einer Message verpackt und über Message Channels versendet. Strukturiert sind die Daten in Header, welche für das Messaging System generell relevant sind, sowie Body, welche die eigentlich gesendeten Daten umfässt [Gre04].

Pipes and Filters

Um komplexe Abläufe auch in einem lose gekoppelten System umzusetzen, verwenden wir *Pipes and Filters* als fundamentalen Architekturstil. Die Abläufe werden in kleine unabhängige Komponenten, die Filters, zerlegt und über Channels, die Pipes, miteinander verbunden. Die Filter haben einen Input für eingehende Messages und einen Output für ausgehende Messages [Gre04].

Message Router

Message Router sind spezielle Filter, welche die empfangene Message gemäß gegebener Bedingungen, auf einen anderen *Message Channel* weiterleiten. Er bearbeitet die Message nicht und kümmert sich ausschließlich um das Weiterleiten. Router unterscheiden sich zusätzlich von normalen Filtern der *Pipes and Filters* Architektur, indem sie mehrere Outbound Pipes haben [Gre04].

Message Translator

Komponenten erwarten spezielle Nachrichtenformate, um diese zu bearbeiten. Es kann vorkommen, dass zwei Komponenten unterschiedliche Formate erwarten. Ein Translator überführt ein Format in ein anderes, um die Kompatibilität zweier Komponenten zu ermöglichen, ohne dass diese zwei Komponenten angepasst werden müssen [Gre04].

Hohpe und Woolf [Gre04] verweisen hier auf das *Adapter* Pattern der Gang of Four. Dieses sorgt ebenfalls dafür, dass zwei verschiedene Schnittstellen miteinander kompatibel sind.

Message Endpoint

Damit Anwendungen Messaging nutzen können, müssen sie an das Messaging System angeschlossen werden. Dies geschieht über einen für die Anwendung entwickelten *Message Endpoint*. Der Endpoint dient als Client des Messaging Systems und kann Messages senden oder empfangen [Gre04].

Point-to-Point Channel

Eine Variante der *Message Channel*s, bei dem die Message von nur einem Empfänger empfangen wird. Befinden sich mehrere Empfänger an demselben Channel, kann nur einer von ihnen die Nachricht bearbeiten [Gre04].

Publish-Subscribe Channel

Alle Empfänger eines *Publish-Subscribe Channel*s, Subscriber genannt, bekommen eine Kopie der versendeten Message. Messages, die auf diesem Channel versendet werden, sind Events. Ein Subscriber bekommt eine Event-Nachricht nur einmal [Gre04].

*Publish-Subscribe Channel*s erweitern das *Observer* Pattern der Gang of Four. Beim *Observer* Pattern haben wir Subjekte, welche Änderungen ihres Zustandes ankündigen und Observer, welche sich an diese Änderungen interessieren. Die Anzahl an Observer ist dabei egal, d.h. es können auch keine Observer auf Zustandsänderungen eines Subjektes achten [Gre04].

Im *Observer* Pattern befinden sich Subjekt und Observer in derselben Anwendung, weshalb die Informierung über lokale Methodenaufrufe implementiert ist. Für verteilte Anwendungen im Kontext des Messaging können wir *Publish-Subscribe Channel* verwenden. Subjekte bekommen ein *Message Gateway*, welches Änderungen des Subjekts von `Notify()` auf den Channel propagiert.

Observer bekommen ebenfalls *Message Gateway*s, welche die propagierten Änderungen über den Channel lesen und an die darunterliegenden Observer weiterleitet. Observer registrieren sich, indem sie den *Publish-Subscribe Channel* abonnieren (engl. subscriben) [Gre04].

Hohpe und Woolf [Gre04] verweisen auf das namensähnliche *Publisher-Subscriber* Pattern von Buschmann et al. Bei diesem handelt es sich um das *Observer* Pattern der GoF unter anderem Namen, weswegen es hier nicht weiter ausgeführt wird. Interessanter ist jedoch eine von Buschmann et al. aufgelistete Variante dieses Patterns: *Event Channel*. Dieses findet in verteilten Systemen Verwendung und bietet weitere Entkopplung zwischen Komponenten. Die Idee ist, dass sich Publisher nicht für die Subscriber interessieren und vice versa. Wichtig sind nur die Daten der Zustandsänderung, nicht von welcher konkreten Komponente sie stammen. Bei dieser Variation befindet sich ein Event Channel zwischen Publisher und Subscriber, über dem die Kommunikation läuft [BMR+96]. Dies ist genau der gleiche Sachverhalt wie in *Publish-Subscribe Channel*.

Datatype Channel

Für jeden Datentyp gibt es einen separaten Channel. So kann der Empfänger sicherstellen, dass die empfangenen Messages den erwarteten Datentyp besitzen, ohne diesen explizit zu prüfen [Gre04].

Invalid Message Channel

Kann ein Empfänger eine empfangene Nachricht nicht verarbeiten, muss diese Nachricht aus dem System entsorgt werden. Damit man nachvollziehen kann, weswegen die Nachricht nicht verabeiten werden konnte, darf man die Message nicht einfach löschen. Aus diesem Grund wird eine solche Message an einen *Invalid Message Channel* gesendet [Gre04].

Dead Letter Channel

Nachrichten, die das Messaging System nicht ausliefern kann, werden in den *Dead Letter Channel* gelegt. Gründe, warum das System nicht ausliefern kann sind bspw. fehlerhafte Header Informationen oder abgelaufene Messages. Jede Installation des Messaging Systems hat einen eigenen lokalen *Dead Letter Channel* [Gre04].

Guaranteed Delivery

Damit keine Nachrichten verloren gehen, müssen sie persistent gespeichert werden. Messaging Systeme speichern Nachrichten so lange in einer lokalen Datenbank, bis diese erfolgreich versendet wurden. Die Datenbank sorgt dafür, dass die Nachricht auch bei Ausfällen des Systems noch vorliegt [Gre04].

Channel Adapter

Ein Channel Adapter kann eine nicht-messaging Anwendung in eine Messaging Umgebung integrieren. Der Adapter nimmt Messages entgegen und interagiert entsprechend mit der API der Anwendung. Weiterhin generiert er Messages aus der Anwendung und sendet sie. Channel Adapter sind auf verschiedene Ebenen einer Anwendung möglich: UI-, Business-, Logik- und Datenbank-Ebene [Gre04].

Messaging Bridge

Verwenden wir mehrere Messaging Systeme, müssen diese verbunden werden, damit Messages aus dem einen System auch in den anderen verfügbar sind. Die *Messaging Bridge* verbindet mehrere Systeme miteinander, indem sie mehrere *Channel Adapter* verwendet [Gre04].

Message Bus

Eine Integrationsarchitektur, bei der Anwendungen über einen *Message Bus* miteinander verbunden werden. Der Bus agiert als Middleware. Die Kommunikation innerhalb des Busses basiert auf Messaging [Gre04].

Command Message

Über eine *Command Message* kann eine Funktion in einer anderen Anwendung aufgerufen werden. Die Message hat keinen besonderen Typen. Sie enthält die Information, dass eine Funktion aufgerufen werden soll bspw. in XML als SOAP Message [Gre04].

Bei diesem Pattern handelt es sich um eine Message Variante des *Command* Pattern der Gang of Four, so Hohpe und Woolf [Gre04]. Ein Befehl bzw. Request wird in einem Objekt gespeichert und kann somit weitergegeben werden. Er beinhaltet dabei alle Informationen, welche zum Ausführen des Befehls notwendig sind [GHJV15].

Document Message

Daten können zwischen Anwendungen im Kontext des Messaging über eine *Document Message* ausgetauscht werden. Relevant sind dabei die Daten der Message. Die empfangende Anwendung entscheidet, was sie mit den empfangenen Daten macht [Gre04].

Event Message

Event Messages werden dazu verwendet, Anwendungen über Events zu benachrichtigen. Wir unterscheiden zwischen zwei Modellen des *Event Message* Patterns: *Push Model* informiert über ein Event und stellt alle notwendigen Daten in der Message bereit. *Pull Model* informiert lediglich über ein Event. Verlangt die informierte Anwendung weitere Informationen, müssen diese über eine *Command Message* extra nachgefragt werden [Gre04].

Request-Reply

Messaging über *Message Channel*s verläuft in nur einer Richtung. Funktionsaufrufe haben zu einer Anfrage jedoch einen Rückgabewert, weswegen wir beide Richtungen benötigen. Das *Request-Reply* Pattern sieht vor, dass wir einen Channel für die Anfrage (Request) und einen Channel für die Antwort (Reply) bereitstellen [Gre04].

Return Address

Komponenten sind lose gekoppelt und kennen sich nicht. Für bspw. *Request-Reply* muss eine Anwendung jedoch eine Antwort-Nachricht zurückgeben können. Die anfragende Komponente spezifiziert daher in seiner Anfrage-Nachricht einen *Message Channel*, auf die die antwortende Komponente eine Antwort hinterlegen kann [Gre04].

Correlation Identifier

Im Kontext von *Request-Reply* bekommt eine Komponente eine Antwort-Nachricht auf eine Anfrage. Aufgrund der asynchronen Natur des Messaging muss die Antwort einen *Correlation Identitfier* angeben, damit die Komponente die Antwort der Anfrage zuordnen kann [Gre04].

Hohpe und Woolf [Gre04] verweisen auf das *Asynchronous Completion Token* Pattern aus Pattern-Oriented Software Architecture Volume 2. Bei diesem Pattern wird bei einem asynchronem Serviceaufruf Information mitgegeben, welche dazu dienen, die Antwort eines Services auf die entsprechende Anfrage zu mappen. Als Kontext des Patterns wird ein Event-Driven System mit asynchronen Anfragen auf Services angegeben. Dies passt zu *Correlation Identitfier*, da wir uns im Messaging in ähnlichem Terrain bewegen. Empfängt eine Komponente eine Message, kann sie diese verarbeiten. Dies entspricht einem Event-Driven Verhalten. Asynchronität ist aufgrund von Messaging ebenfalls gegeben. Die Information um die Antwort auf eine Anfrage zu mappen, findet sich im *Correlation Identifier* als primitiver Wert innerhalb der Message.

Message Sequence

Messages können nicht beliebig groß sein. Um dennoch große Nachrichten senden zu können, werden die Daten in mehrere Teile geteilt. Die Teildaten werden als einzelne Nachrichten mitsamt einem Sequenz-Identifizierungs-Feld vesendet [Gre04].

Message Expiration

Messages können über die *Message Expiration* definieren, wie lange sie gültig sind. Abgelaufene Messages werden von dem Messaging System bspw. in den *Dead Letter Channel* gesendet oder gelöscht. Als *Message Expiration* dient ein Zeitstempel mit Datum und Uhrzeit [Gre04].

Format Indicator

Datenformate können sich jederzeit ändern. Ein *Format Indicator* dient dazu, dass eine Message spezifizieren kann, welches Format die beigelegten Daten verwenden. Der Indikator findet sich in den Daten selbst [Gre04].

Content-Based Router

Ein *Content-Based Router* überprüft den Inhalt einer empfangenen Nachricht und entscheidet entsprechend dem Inhalt, auf welchen Channel sie weitergeleitet wird. Der Router verändert den Inhalt der Nachricht nicht [Gre04].

Message Filter

Ein Filter verbindet zwei Channels miteinander. Nachrichten, die in den Filter eingehen, werden entsprechend gegebener Kriterien entfernt, sodass auf dem verbundenen Channel nur gewünschte Nachrichten übrig bleiben [Gre04].

Dynamic Router

Content-Based Router haben eine statische Konfiguration, mögliche Channels und Routingregeln, welche in den Router eingetragen werden müssen. Ein *Dynamic Router* hingegen erlaubt das Setzen von Regeln dynamisch über einen Control Channel. Die Regeln werden in einer Rule Base persistent gespeichert [Gre04].

Hohpe und Woolf [Gre04] verlinken *Dynamic Router* mit dem *Client-Dispatcher-Server* Pattern von Buschmann et al. Ein Dispatcher dient als Lookup-Tabelle. Clients können in einem verteilten System über einen Namen nach Services anfragen. Der Dispatcher kennt die physische Location des Services und stellt eine Verbindung zwischen Client und dem Server, der den Service anbietet, her [BMR+96]. Die Rule Base des *Dynamic Router*s hat Ähnlichkeit zur Lookup-Tabelle des *Client-Dispatcher-Server*s. Der Unterschied ist jedoch, dass der *Dynamic Router* auch mögliche Konflikte lösen muss.

Recipient List

Eine *Recipient List* ist ein Router, welcher eine Liste von Interessenten auf Basis des Nachrichteninhalts erstellt. Jeder Interessent entspricht einem Channel, auf welcher die eingehende Nachricht weitergesendet wird. Regeln zum Identifizeren der Interessenten finden sich entweder fest in der *Recipient List* Komponente, oder lassen sich dynamisch über einen Control Bus setzen [Gre04].

Splitter

Ein Message Objekt, welches mehrere Nachrichten in einem einzelnen Objekt kombiniert, wird in einzelne Nachrichten aufgeteilt, welche anschließend individuell bearbeitet werden kann. Viele Enterprise Integration Systeme speichern Daten in einer Baum-Struktur, welche das Aufspalten in einzelne Teile vereinfacht. Wie genau gesplittet wird, muss in dem *Splitter* definiert werden [Gre04].

Aggregator

Einzelne Nachrichten, die zusammengehören, werden über einen *Aggregator* zu einer einzelnen Nachricht verbunden. Dieser sammelt solange Nachrichten, bis eine komplette Nachricht zusammengetragen wurde und sendet diese dann. Wie Nachrichten zusammengehören, wie die Vollständigkeit bewertet wird und wie die einzelnen Nachrichten zusammengesetzt werden, muss im *Aggregator* definiert werden [Gre04].

Resequencer

Durch die Existenz von *Message Router* kann die Reihenfolge der gesendeten Nachrichten beim Empfang nicht garantiert werden. Ein *Resequencer* sorgt dafür, dass die Reihenfolge der Nachrichten erhalten bleibt. Die Reihenfolge kann über eine Sequenz-Nummer bestimmt werden [Gre04].

Verlinkungen finden sich in *Identity Field* der Patterns of Enterprise Application Architecture. Dieses wird als Sequenz-Nummer der einzelnen Nachrichten verwendet. Die Funktionsweise dieses Pattern findet sich in einem späteren Abschnitt.

Composed Message Processor

Zusammengesetzte Nachrichten können durch einen *Splitter* in individuelle Nachrichten aufgeteilt werden. Jede Nachricht kann so einzeln verarbeit werden. Verarbeitete Nachrichten können über einen *Aggregator* in eine neue zusammengesetzte Nachricht verbunden werden. Den Verbund dieser Pattern in dieser Form entspricht dem *Composed Message Processor* [Gre04].

Scatter-Gather

In *Scatter-Gather* wird eine Nachricht an mehrere Empfänger gesendet. Alle Antworten werden in einem *Aggregator* gesammelt und in eine einzelne Antwort-Nachricht umgewandelt [Gre04].

Routing Slip

Ein *Routing Slip* spezifiziert eine Reihe von Bearbeitungsschritten, die eine Nachricht durchlaufen soll. Jede Komponente entnimmt der Liste die Adresse des nächsten Bearbeitungsschrittes und versendet die Nachricht nach dem Bearbeiten dorthin. Alle Komponenten haben einen generischen Router, welcher die Nachricht nach dem Bearbeiten weiterleitet. Die Reihe von Bearbeitungsschritten muss vorher gesetzt werden. Sie ist nicht dynamisch veränderbar [Gre04].

Routing Slip ist eine Modifikation der *Chain of Responsibility* der Gang of Four, so Hohpe und Woolf [Gre04]. Wir definieren eine statische Zuständigkeitskette, welche Objekte bzw. Komponenten umfasst, die einen Request behandeln können. Eine Anfrage durchläuft diese Kette, bis sie von einem Objekt bearbeitet wurde. Jede Komponente entscheidet ensprechend der Liste nacheinander, ob sie die Anfrage behandeln oder weitergeben möchte [GHJV15].

Process Manager

Ein *Process Manager* dient als zentrale Komponente und entscheidet die nächsten Bearbeitungsschritte einer Nachricht. Dabei speichert er den Zustand der Verarbeitung einer Nachricht sowie Ergebnisse aus weiteren Bearbeitungen um nächste Schritte zu entscheiden. Komplexe Nachrichtenflüsse, wie es bspw. im Business Process Management Bereich der Fall ist, werden über diese Komponente umgesetzt [Gre04].

Message Broker

Hub-and-Spoke Architekturstil. Der zentraler Router, *Message Broker*, verbindet alle Komponenten miteinander. Nachrichten werden an den Router geschickt, dieser entscheidet, an welche anderen Komponenten die Nachricht gesendet wird [Gre04].

Envelope Wrapper

Ein Umschlag enthält alle zusätzlichen Informationen, die für das verwendete Messaging System notwendig sind. Daten, die nicht mit dem Messaging System kompatibel sind, werden über den *Envelope Wrapper* in diesen Umschlag gelegt, über das Messaging System versendet und auf der Empfängerseite geöffnet. Der *Envelope Wrapper* ist nicht nur auf Header Feldern beschränkt. Er kann auch Daten im Body ergänzen [Gre04].

Content Enricher

Die Daten einer Message könnten nicht vollständig sein, d.h. es fehlen benötigte bzw. zusätzliche Daten. Diese sind abhängig vom konkreten Fall. Ein *Content Enricher* hat Zugang zu einer externen Datenquelle, mithilfe er die fehlenden Informationen in die Message eintragen kann. Die Datenquelle könnte dabei sogar ein externes System sein, welches außerhalb der Anwendung liegt [Gre04].

Content Filter

Über einen *Content Filter* werden nicht verwendete Daten aus einer Nachricht entfernt. Dies könnte bspw. aus Sicherheitsgründen sein, falls die Nachricht sicherheitskritische Information enthält. Ein weiterer Grund ist die Reduzierung der Nachrichtengröße, damit der Netzwerkverkehr im System möglichst gering ist. Komplexere Nachrichten können über einen *Content Filter* auch vereinfacht werden [Gre04].

Claim Check

Daten werden über eine Check Luggage Komponente aus der Message in einen Data Store ausgelagert. Die Message bekommt einen *Claim Check*, einen Schlüssel, um die ausgelagerten Daten zu einem späteren Zeitpunkt wieder abrufen zu können. Die reduzierte Message kann von weiteren Komponenten verwendet werden [Gre04].

Normalizer

Wir haben verschiedene Message Formate wie bspw. XML, CSV oder auch Plain Text. Ein *Normalizer* besteht aus einem Translator pro Format, welcher dieses in ein gemeinsames Format übersetzt. Ein Router verteilt die Nachrichten gemäß ihres Formates in den entsprechenden Translator [Gre04].

Canonical Data Model

Mehrer Komponentent nutzen ihr individuelles Datenformat. Ein *Canonical Data Model* ist ein unabhängiges Format für alle Komponenten. Anstatt eine Komponente einen Translator für jede andere Komponente benötigt, werden durch *Canonical Data Model* nur zwei Translators eingesetzt. Einer hin zum unabhängigen Format und einer zurück zum Format der Komponente. Die Anzahl der benötigten Übersetzer sinkt somit auf zwei pro Komponente und ist unabhängig von der Anzahl der anderen Komponenten [Gre04].

Messaging Gateway

Ein *Messaging Gateway* bindet eine Applikation in das Messging System ein. Das Senden von Nachrichten oder auch Empfangen von Nachrichten und Aufrufen von Applikationslogik wird über dieses Pattern geregelt. Die Applikation selbst nimmt so keine Kenntnis über das Messaging System [Gre04].

Hohpe und Woolf [Gre04] erwähnen, dass *Messaging Gateway* eine Variante des *Gateway* Pattern von Folwer et al. ist. Im Wesentlichen umfässt ein *Gateway* eine oder mehrere Komponenten und regelt den Zugriff auf diese. Anstelle direkt auf die Komponenten zuzugreifen, geschieht das nun über das *Gateway*. Das *Gateway* Pattern wird in dem entsprechenden Abschnitt weiter unten beschrieben. Fowler et al. gehen dort auch auf das *Messaging Gateway* Pattern ein.

Hohpe und Woolf [Gre04] verweisen auf das *Service Stub* Pattern von Folwer et al., welches Testen ermöglicht. Das *Messaging Gateway* bietet eine Schnittstelle nach außen hin, damit Komponenten Anfragen stellen können. Im Hintergrund haben wir zwei Implementierungen dieser Schnittstelle: die echte Komponente und einen Dummy bzw. Mock. Wollen wir unser System testen, wechseln wir zur Mock-Implementierung und können entsprechend testen.

Messaging Mapper

Das Datenformat einer Message muss nicht mit dem Datenformat der Domain Objekte übereinstimmen. Ein *Massaging Mapper* ist eine Komponente, welche das Mapping dieser zwei Objekte übernimmt [Gre04].

Der *Messaging Mapper* basiert auf das *Mapper* Pattern von Fowler et al., weswegen Hohpe und Woolf [Gre04] darauf verweisen. Ein Mapper sorgt generell dafür, dass zwei Komponenten miteinander kommunizieren können, obwohl deren Objekte inkompatibel sind. Dabei bleiben die Komponenten unabhängig voneinander und nehmen auch keine Kenntnis von dem Mapper.

Das Mapping zwischen Objekt und Nachricht hat Ähnlichkeiten mit Object-Relational Mapping. Für letzteres finden sich mehrere Pattern von Fowler et al., welche sich um diese Problematik kümmern. Darunter fällt bspw. das *Data Mapper* Pattern, auf das im späteren Verlauf noch eingegangen wird.

Die Absicht des *Messaging Mapper* findet sich auch im *Mediator* Pattern der GoF. Eine Menge von Komponenten kommunizieren untereinander über einen Vermittler, den Mediator, welcher die Interaktionen dieser koordiniert. Beim *Mediator* haben die beteiligten Objekte eine Referenz auf den Mediator, anstelle Referenzen auf die entsprechenden Komponenten [GHJV15]. Der große Unteschied zum *Messaging Mapper* ist jedoch, dass die beteiligten Komponenten den Mapper nicht kennen bzw. keine Kenntnis von ihm nehmen.

Wie oben beschrieben ist das eins zu eins Mapping von Message und Domain-Object nicht immer die beste Lösung. In manchen Szenarien empfiehlt es sich bspw. mehrere kleinere Objekte in einer Nachricht zu bündeln. Der *Messaging Mapper* sorgt dafür, dass die Daten in die entsprechenden Objekte kommt, bzw. dass die richtigen Objekte in eine Nachricht gepackt werden. Diese Vorgehensweise findet sich im *Data Transfer Object* aus den Pattern von Fowler et al. wieder, auf die später eingegangen werden. Auf Grund dieser Ähnlichkeit verweisen Hohpe und Woolf [Gre04] auf dieses Pattern.

Transactional Client

Ein *Transactional Client* eines Messaging Systems kann die Transaktionsgrenzen einer Session selbst setzen. Auf Senderseite werden Nachrichten erst dann richtig versendet, wenn er die sendende Transaktion committet. Auf Empfängerseite werden Nachrichten aus einem Channel erst dann entfernt, wenn er die bearbeitende Transaktion committet [Gre04].

Polling Consumer

Die Verarbeitungsgeschwindigkeit von Messages wird von einem *Polling Consumer* selbst bestimmt. Der Receiver blockiert solange, bis er eine Message empfängt. Nachrichten werden nur dann empfangen, wenn der Receiver bereit dafür ist, sprich auf eine Nachricht wartet [Gre04].

Event-Driven Consumer

Das Messaging System informiert *Event-Driven Consumer* über eintreffende Nachrichten. Die Nachrichten werden der Komponente automatisch zugestellt, sobald sie im Channel vorliegen. Nach Erhalt der Message kann diese bearbeitet werden [Gre04].

Competing Consumers

Mehrere austauschbare Consumer sind Empfänger desselben Channels. Eine eingehende Nachricht wird nur von einem Consumer verarbeitet. Anstelle einer sequenziellen Verarbeitung ist somit eine nebenläufige Abarbeitung der Nachrichten möglich [Gre04].

Message Dispatcher

Ein *Message Dispatcher* nimmt die Nachrichten eines Channels entgegen und versendet sie an Performer. Performer sind Objekte, die die Nachricht letztendlich verarbeiten [Gre04].

Das Versenden der Nachricht an die Performer kann über einen Mediator umgesetzt werden, weswegen Hohpe und Woolf [Gre04] an dieser Stelle darauf verlinken. Das *Mediator* Pattern bietet einen Vermittler, welcher Funktionen innerhalb einer Gruppe von Objekten koordiniert. Die Objekte müssen sich demnach nicht selbst koordinieren [GHJV15].

Selective Consumer

Ein Consumer kann willkürliche Nachrichten aus dem Channel bekommen. *Selective Consumer* hingegen filtern Nachrichten, sodass sie nur Nachrichten verarbeiten müssen, welche ihren Kriterien entsprechen. Erfüllt die Nachricht die Kriterien, wird sie zu einem darunterliegenden Empfänger weitergeleitet und bearbeitet [Gre04].

Durable Subscriber

Abonnierte Komponenten verpassen Nachrichten trotz Subscription, wenn sie inaktiv sind. Bei *Durable Subscriber* ist das nicht der Fall. Das Messaging System speichert Nachrichten für sie so lange, bis sie ausgeliefert werden [Gre04].

Idempotent Receiver

Aufgrund von *Guaranteed Delivery* wird sichergestellt, dass jede Nachricht beim Empfänger ankommt. Nicht sichergestellt ist, dass die Nachricht nur einmal eintrifft. *Idempotent Receiver* sind Empfänger, welche fehlerfrei gleiche Nachrichten mehrmals empfangen können [Gre04].

Service Activator

Services müssen nicht explizit für das Messaging System programmiert sein, um sie in dieser Umgebung verwenden zu können. Ein *Service Activator* nimmt Nachrichten des Messaging Systems entgegen und ruft entsprechende Services auf. Mögliche Rückgaben wandelt er in weitere Nachrichten um und versendet diese über das Messaging System. Aus Sicht des Services wird es wie bei einem synchronen Aufruf aufgerufen [Gre04].

Hohpe und Woolf [Gre04] verweisen auf das *Service Layer* Pattern von Fowler et al. Ein *Service Layer* ist eine Abstraktionsschicht einer Anwendung und bietet eine klare API. Ein bereitgestellter Service einer Anwendung entspricht einer Operation in einem *Service Layer*. Der *Service Activator* sorgt dafür, dass die Operationen über Nachrichten aufgerufen werden können [Fow02].

Control Bus

Für Management erweitern wir das System um einen *Control Bus*, welches als eigenständiges Subsystem neben der Anwendung läuft. Nachrichten und Daten, die relevant für das Management sind, befinden sich in diesem Bus. Komponenten haben neben Inbound und Outbound Pipes einen Pipe für Control Daten [Gre04].

Detour

Detour agiert als Router und leitet eine Nachricht auf Befehl des Control Busses auf einen Umweg. Als Umweg verstehen wir hier mehrere zusätzliche Komponenten bzw. Bearbeitungsschritte, bevor die Nachricht letztendlich zu ihrem eigentlichen Empfänger geleitet wird [Gre04].

Wire Tap

Um Nachrichten eines Channels zu untersuchen, zerteilt *Wire Tap* einen Channel in zwei und verbindet beide Teile. Zusätzlich hat der *Wire Tap* einen weiteren Output Channel, auf dem Kopien der Nachrichten landen, damit sie von einer anliegenden Komponente untersucht werden können. Nachrichten werden unberührt an die Output Channel weitergeleitet [Gre04].

Message History

Komponenten vermerken in einer Liste, wenn sie eine Nachricht empfangen und verarbeitet haben. Die Liste ist Teil der Nachricht und wird *Message History* bezeichnet. Der Verlauf einer Nachricht innerhalb des Systems lässt sich damit im Nachhinein nachvollziehen [Gre04].

Message Store

Ein *Message Store* dient dazu, Daten einer Nachricht an einer zentraler Stelle zu speichern, um bspw. Statistiken zu ermitteln. Senden wir eine Nachricht an einen Channel, wird eine Kopie an den *Message Store* gesendet [Gre04].

Smart Proxy

Wir wollen Anfragen und Antworten im Kontext des *Request-Reply* Patterns untersuchen. *Smart Proxy* fängt eine Nachricht ab, ersetzt die *Return Address* mit einer Adresse eines eigenen Channels und sendet die Nachricht weiter an den Empfänger. Die Antwort des Empfängers trifft am Channel des *Smart Proxy* ein und wird dort an die ursprüngliche *Return Address* weitergeleitet [Gre04].

Test Message

Heartbeat Messages können über den *Control Channel* versendet werden, damit eine Komponente signalisieren kann, dass sie noch verfügbar ist. Ob eine Komponente beschädigt ist, sprich ob sie falsche Ergebnisse produziert, erkennt man damit jedoch nicht. *Test Message* überprüft die korrekte Funktionsweise von Komponenten, ähnlich wie in einem Unit Test: Test Daten mit erwartetem Ergebnis werden in die Komponente eingeschleust, das Ergebnis wird ausgewertet. *Test Message* sorgt dafür, dass solche Test Daten das System auf Business Ebene nicht beeinflussen [Gre04].

Channel Purger

Channel Purger ist eine Komponente, welche unerwünschte Nachrichten aus dem System entfernt [Gre04].

3.2.2 Cloud Computing Patterns

Der Grundgedanke von Cloud Computing ist die Verwendung von IT Ressourcen als Service. Es lassen sich dabei Analogien zu einem Wasserhahn ziehen: Wenn wir Wasser benötigen, drehen wir den Hahn auf. Die Menge lässt sich, je nach aufdrehen des Hahns, selbständig justieren. Bezahlen müssen wir entsprechend der Menge des herausgelassenen Wassers. So ähnlich verhält es sich bei Cloud Computing mit den IT Ressourcen. Wir bekommen die Ressourcen, wann wir sie benötigen und wie viel wir benötigen, bezahlen aber auch nur das, was wir verwenden. Unter Ressourcen fallen dabei Infrastruktur, Middleware, Software oder auch bereitgestellte Geschäftsprozesse [FLR+14].

Die Cloud Computing Patterns stellen Lösungen zu wiederkehrenden Problemstellungen innerhalb des Cloud Computing Kontexts bereit. Die Lösungen sind abstrakt, sodass sie unabhängig von konkreten Cloud-Anbietern, Produkten, Programmiersprachen oder Ähnlichem sind. Durch die gegenseitige Verlinkung der Pattern entsteht die Sprache der Cloud Computing Patterns.

Dieser Abschnitt behandelt die Cloud Computing Patterns von Fehling et al. Die einzelnen Pattern werden hier nacheinander zusammengefasst. Zu jedem Pattern wird anschließend auf mögliche Verlinkungen zu Patterns von anderen Sprachen eingegangen. Der Fokus liegt auf die

Enterprise Integration Patternsprache, welche im vorherigen Abschnitt behandelt wurde. Fehling et al. [FLR+14] erwähnen selbst jedoch teilweise Verbindungen zu Patterns aus anderen Sprachen, auf welche hier ebenfalls eingegangen werden. Diese werden den anderen Sprachen entsprechend vorweggenommen. Findet sich keine Verlinkung, ist auch keine Beschreibung zu anderen Sprachen vorhanden. Zu manchen Pattern wird jedoch explizit beschrieben, warum mögliche Kandidaten doch keine Verlinkungen haben.

Static Workload

Die Arbeitsauslastung der Anwendung verhält sich weitestgehend stabil bzw. statisch und schwankt kaum. Die dynamische Hinzunahme von Ressourcen ist hier nicht notwendig. Eine auf der Cloud gehostete Anwendung kann auch ohne Skalierung von der Cloud profitieren [FLR+14].

Periodic Workload

In bestimmten Zeitabschnitten steigt die Arbeitsauslastung an. Die Cloud ermöglicht hier die Verwendung von zusätzlichen Ressourcen bei Bedarf. Sinkt die Arbeitsauslastung, können Ressourcen freigegeben werden [FLR+14].

Once-in-a-Lifetime Workload

Ähnlich wie in *Periodic Workload* haben wir einen Anstieg der Arbeitsauslastung. Dieser Anstieg findet jedoch nur einmal in einem sehr langen Zeitintervall statt. Auch hier werden Ressourcen dynamisch hinzugezogen, sobald der Anstieg stattfindet [FLR+14].

Unpredictable Workload

Die Auslastung verhält sich willkürlich und kann nicht vorhergesehen werden. Das Hinzuziehen und Freigeben von Ressourcen muss automatisiert werden, um diese dynamisch der Auslastung anzupassen. Clients können die Auslastung der Anwendung überwachen. Die Automatisierung kann durch Nutzung einer Cloud Architektur ermöglicht werden [FLR+14].

Continuously Changing Workload

Wir haben langanhaltende und konsistente Steigung oder Senkung der Arbeitsauslastung einzelner Anwendungen. Die benötigten Ressourcen können schrittweise an den Bedarf angepasst werden, sofern die Änderungsrate bekannt ist. Bei unbekannter Änderungsrate verhält es sich wie in *Unpredictable Workload* [FLR+14].

Infrastructure as a Service (IaaS)

Beim *Infrastructure as a Service* Model werden IT Ressourcen wie Rechenleistung, Speicher und Netzwerkinfrastruktur angeboten. Clienten können diese über Selbstbedienung anfordern oder freigeben. Kosten fallen je nach Benutzung aus [FLR+14].

Eine Messaging System bedient sich im Kern einer Message-oriented Middleware. Bei dieser handelt es sich um eine Software ähnlich wie ein Applikations Server. Anstelle von Applikationen werden auf dieser Middelware jedoch Queues und Topics gehostet. Unter einer *Infrastructure as a Service* kann ein Betriebssystem aufgesetzt werden. Auf diesem Betriebssystem wird anschließend die Message-oriented Middleware aufgesetzt. Manche Betriebssysteme enthalten bereits eine solche Middleware, wie bspw. Windows, mit der Microsoft Messaging Queue (MSMQ). Ein Messaging System lässt sich demnach auf der IaaS realisieren.

Platform as a Service (PaaS)

PaaS bietet Betriebssystem und Middleware an, sodass ein Client seine Anwendung in die Cloud einbringen kann. Bei diesem Service Model geht es nur um das Hosting der Anwendungen. Bspw. wird eine Java Virtual Machine (JVM) bereitgestellt. Clients können ihre Java Anwendung auf dieser JVM betreiben [FLR+14].

Selbige Argumentation wie in *Infrastructure as a Service*. Unterschied ist hier jedoch, dass kein extra Betriebssystem installiert werden muss. Die PaaS kann ein Messaging System ohne explizite installation bereitstellen. Beispiel findet sich in den Amazon Web Services mit dem Simple Queue Service (`https://aws.amazon.com/de/sqs/`).

Software as a Service (SaaS)

Die Software wird auf der Cloud über das *Software as a Service* Model gehostet. Anstelle Software auf einem Computer zu installieren, erfolgt der Zugriff auf die Software bspw. über einen Internet Browser [FLR+14].

Public Cloud

Ressourcen werden für eine große Anzahl an Benutzer bereitgestellt, welche diese untereinander teilen [FLR+14].

Private Cloud

Im Gegensatz zur *Public Cloud* werden die Ressourcen einer *Private Cloud* nur für einen Benutzer bzw. eine Benutzergruppe bereitgestellt. Dadurch wird ein hohes Maß an Sicherheit und Privatsphäre ermöglicht [FLR+14].

Community Cloud

Eine Guppe von Benutzern, welche sich gegenseitig vertrauen, teilen sich Ressourcen in der *Community Cloud*. Nur die Gruppe hat Zugang zu den Ressourcen [FLR+14].

Hybrid Cloud

Verschiedene Cloud Modelle werden unter *Hybrid Cloud* integriert und als eine Hosting Umgebung angeboten. Bei den Cloud Modellen handelt es sich um die *Public Cloud*, *Private Cloud* und *Community Cloud* Modelle. Hierunter fällt nicht das Zusammenbringen von bspw. verschiedenen Public Cloud Anbietern wie Amazon AWS oder Mircosoft Azure [FLR+14].

Elastic Infrastructure

Um Ressourcen in der Cloud dynamisch anfordern oder freigeben zu können, brauchen wir *Elastic Infrastructure*. Über vorher konfigurierte sowie gepoolte Ressourcen lässt sich dies automatisch durchführen [FLR+14].

Elastic Platform

Elastic Platform ist eine Middleware, welche eine Ausführungsumgebung für Anwendungen anbietet. Die darauf laufenden Clients teilen sich die darunterliegende Hosting Umgebung. Die umfasste Funktionalitäten der *Elastic Platform* basieren auf der enthaltenen *Execution Environment* [FLR+14].

Node-Based Availability

Die Verfügbarkeit wird für einzelne Nodes in Prozent angegeben. Als Nodes gelten einzelne Elemente wie Server, Middleware- oder Anwendungskomponenten [FLR+14].

Environment-Based Availability

Die Verfügbarkeit der gesamten Umgebung wird in Prozent angegeben. Anstelle Angaben über einzelne Elemente zu machen, wird bspw. die komplette *Elastic Infrastructure* über die *Environment-Based Availability* abgedeckt. Die Verfügbarkeit einer gehosteten Anwendung muss selbst ermittelt werden, da hier nicht wie in *Node-Based Availability* die Verfügbarkeit der einzelnen Elemente zusammengerechnet werden können [FLR+14].

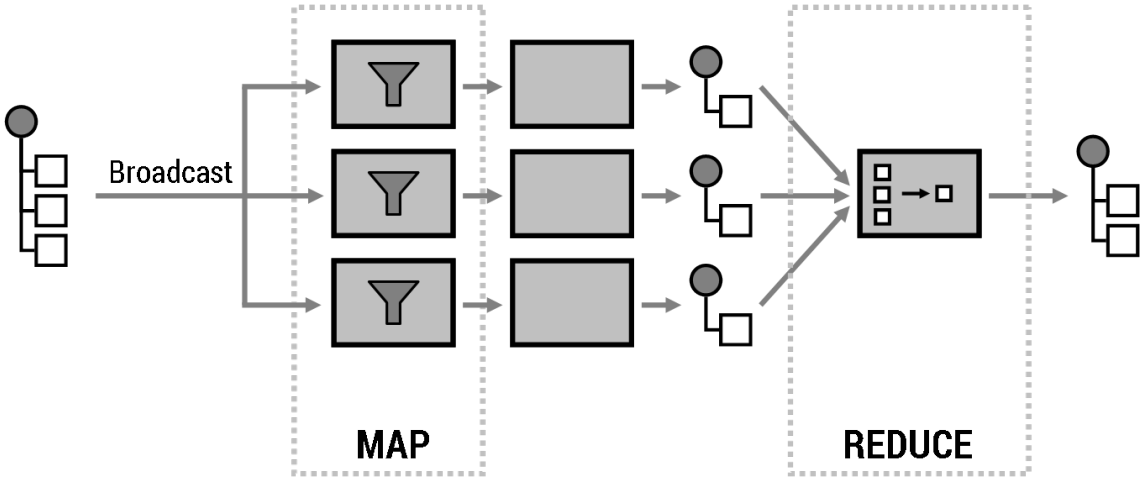

Abbildung 3.3: *Map Reduce* über eine reine *Scatter-Gather* Implementierung

Hypervisor

Ressourcen werden virtualisiert, damit sie den Clients schneller zugewiesen werden können als echte Hardware. Typ 1 *Hypervisor* sitzen direkt auf der Hardware und virtualisieren diese. Typ 2 *Hypervisor* hingegen werden als Software auf ein existierendes Betriebssystem installiert und virtualisieren dann [FLR+14].

Execution Environment

Funktionalitäten, welche von mehreren Anwendungen benötigt werden, finden sich gebündelt in der *Execution Environment*. Als Beispiel dient die JVM, welche als solche Ausführungsumgebung für Java Anwendungen dient [FLR+14].

Map Reduce

Ein großer Datensatz wird in kleinere zerlegt und auf einzelne Bearbeitungskomponenten gemappt. Auf jeden dieser Teilsätze wird eine Query ausgeführt. Die Ergebnisse der einzelnen Komponenten werden anschließend auf einen einzelnen Ergebnisdatensatz reduziert [FLR+14].

Fehling et al. [FLR+14] verweisen auf das *Scatter-Gather* Pattern der *Enterprise Integration Patterns* Dieses Pattern hat eine ähnliche Arbeitsweise wie *Map Reduce*. Eine Nachricht wird an mehrere Empfänger gesendet. Die Antworten der Empfänger werden zusammengetragen, um die beste Antwort zu ermitteln. Die beste Antwort wird anschließend zurückgegeben.

Jedoch muss man hier stark unterscheiden, dass die Nachricht in *Scatter-Gather* nicht aufgeteilt, sondern komplett an die Empfänger gesendet wird. Abbildung 3.3 zeigt eine mögliche Implementierung von *Map Reduce* über *Scatter-Gather*. Die ganze Nachricht wird über einen Broadcast an eine Reihe von Filtern gesendet. Die Filter nehmen sich einen Teil der Daten und senden sie weiter an Bearbeitungskomponenten. Diese führen die Query auf den Teildaten aus und leiten das Ergebnis an einen *Aggregator*. Der *Aggregator* konsolidiert die Teilergebnisse und führt sie zu dem Ergebnis.

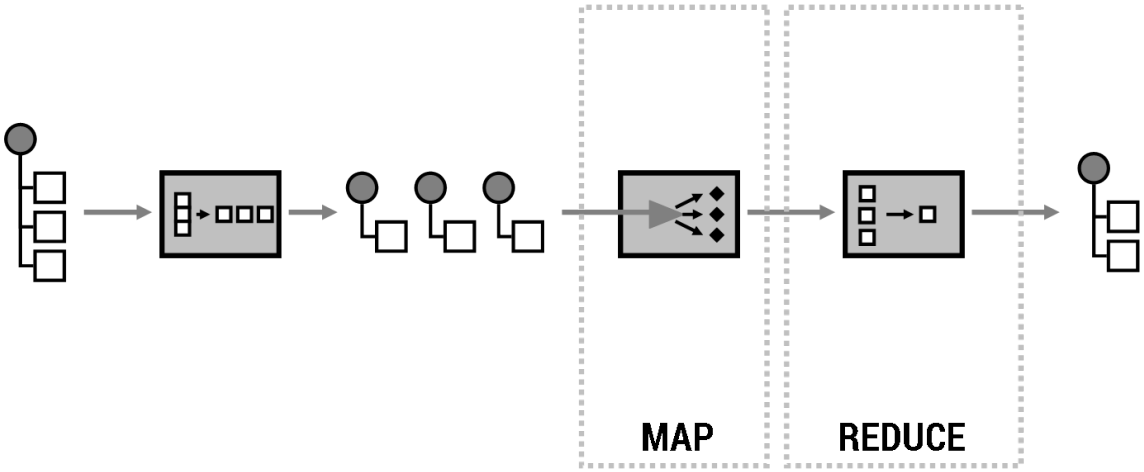

Abbildung 3.4: *Map Reduce* über eine reine *Composed Message Processor* Implementierung

Problematik in dieser Variante ist, dass die gesamte Nachricht an alle Bearbeitungskomponenten gesendet wird. Dies führt zu einer evtl. zu hohen Netzwerkauslastung. Weiterer Problempunkt sind die Filter. Damit die Daten richtig verteilt werden, müssen die Filter miteinander kooperieren. Aufgrund dieses Managementoverheads und der hohen Netzwerkauslastung ist diese Lösung nicht geeignet.

Eine Alternative zu *Scatter-Gather* ist der *Composed Message Processor*. Diese Variante findet sich in Abbildung 3.4. Die ganze Nachricht wird über einen *Splitter* in einzelne Teildaten aufgeteilt. Die Teildaten führen zu einer Queue eines *Competing Consumers*. Die darin enthaltenen Bearbeitungskomponenten führen die Query auf die Teildaten aus und senden das Ergebnis zu einem *Aggregator*, welche diese konsolidiert. Das Ergebnis findet sich nach dem Konsolidierungsschritt.

Bei dieser Alternative vermeiden wir die Problempunkte der *Scatter-Gather* Lösung. Jedoch ist die Verwendung von *Composed Message Processor* hier evtl. nicht ganz so akkurat, wie es von Hohpe und Woolf gedacht ist. Nach dem Splitten werden die Teildaten über einen Router zu speziellen Bearbeitungskomponenten gesendet [Gre04]. Das ist bei der *Map Reduce* Implementierung nicht der Fall. Abgesehen davon ist diese Version die Empfehlenswertere.

Block Storage

Server und Festplattenspeicher werden getrennt. Fällt ein Server aus, kann eine neue Instanz gestartet und mit entsprechendem Festplattenspeicher verbunden werden. Für die Anwendung verhält sich der Speicher trotz getrenntem, also remotem Speicher, wie das lokale Filesystem [FLR+14].

Blob Storage

Große Dateien, wie bspw. Videos, werden hierarchisch, ähnlich der Ordnerstrukturen im Filesystem, als Binary Large Objects (Blob) in einen zentralen Speicher gelagert. Die Dateien haben Identifier bestehend aus Speicherort und Namen, womit diese aus dem Speicher angefragt werden können [FLR+14].

Relational Database

Daten werden in Tabellen gespeichert. Die Spalten der Tabelle repräsentieren die Attribute einzelner Datensätze, eine Zeile entspricht einem Datensatz. Elemente werden durch Schlüssel eindeutig identifiziert. Sie können auf andere Elemente über deren Schlüssel verweisen. Relationen zu diesen anderen Elementen wird so gewährleistet [FLR+14].

Key-Value Storage

Daten werden als Schlüssel-Wert-Paare gespeichert. Für einfache Abhängigkeiten zwischen den Daten eignet sich *Key-Value Storage*, bei Komplexen eher *Relational Database*. Die Nutzung von *Key-Value Storage* ermöglicht hohe Flexibilität aufgrund semi-strukturierten Daten. Hohe Verfügbarkeit und High-Performance werden dadurch erreicht, dass, im Gegensatz zu *Relational Database*, keine Relationen über Komponenten gewährleistet werden muss [FLR+14].

Das *Domain Model*, bestehend aus Daten und die Funktionalität auf diese zuzugreifen, entscheidet, ob *Key-Value Storage* die richtige Wahl ist. Patterns zu *Domain Model* finden sich in den Patterns of Enterprise Application Architecture von Fowler et al. [Fow02], weswegen Fehling et al. [FLR+14] auf diese verweisen.

Strict Consistency

Datenreplikate finden sich auf mehreren Komponenten, um die Verfügbarkeit zu erhöhen. Damit die Konsistenz der Daten gewährleistet wird, muss eine gewisse Anzahl an Schreib- und Lesezugriffe durchgeführt werden. *Strict Consistency* liegt dann vor, wenn $n < w + r$ gilt. Dabei ist n die Gesamtanzahl der Replikas, w die Anzahl der Replikas, aus denen gelesen und r die Anzahl der Replikas, auf denen geschrieben wird [FLR+14].

Eventual Consistency

Daten müssen nicht konsistent sein, ein Lesevorgang kann somit einen älteren Zustand liefern. Lese- und Schreibvorgänge passieren auf nur einem Replika. Weitere Replika werden asynchron geupdatet. Daten werden über die Zeit konsistent [FLR+14].

Virtual Networking

Netzwerkressourcen wie Switches, Router, etc. werden virtualisert. Clients können eigenständig bspw. Netzwerke konfigurieren oder Firewall Regeln erstellen [FLR+14].

Message-oriented Middleware

Verteilte Komponenten kommunizieren asynchron mit Nachrichten über eine *Message-oriented Middleware*. Nachrichten werden in Message Channels gelegt und von anderen Komponenten aus dieser gelesen. Die Komponenten haben Verlinkungen zu den Channels, kennen sich gegenseitig aber nicht, weswegen eine lose Kopplung zwischen den Komponenten ermöglicht wird [FLR+14].

Mit *Message-oriented Middleware* haben wir eine Verbindung zu sämtlichen Messaging relevanten Patterns von Hohpe und Woolf [Gre04]. Durch asynchrone Kommunikation müssen nicht alle Teilnehmer zur gleichen Zeit bereit sein. Im Messaging Kontext haben wir dafür das *Pipes and Filters* Architekturpattern. Komponenenten können nach ihrem eigenen Tempo, d.h. wenn sie bereit sind, die eingehenden Nachrichten bearbeiten. Über *Guaranteed Delivery* wird sichergestellt, dass Nachrichten beim Empfänger auch sicher eintreffen und nicht unterwegs verloren gehen [FLR+14].

Exactly-Once Delivery

Nachrichten werden dank *Guaranteed Delivery* an den Empfänger zugestellt. Mehrfache Zustellung ist jedoch möglich und kann in Komponenten zu Fehlern führen. *Exactly-Once Delivery* sorgt durch explizites Filtern dafür, dass eine gleiche Nachricht nur einmal an der Komponente ankommt [FLR+14].

Das *Message Filter* Pattern der EIP kann Duplikate auf dem Weg vom Sender zum Empfänger filtern. Das Messaging System verseht dabei jede Nachricht mit einer ID. Trifft eine Nachricht an dem *Message Filter* ein, wird die ID auf bereits empfangene Nachrichten überprüft. Ist die ID schon vorhanden, wird die Nachricht nicht weitergeleitet und aus dem Nachrichtenpfad entfernt. Der Filter kann entweder als *Stateful Component* umgesetzt werden oder einen externen Datenspeicher für die Speicherung der IDs verwenden [FLR+14].

Idempotent Receiver aus den EIP sind Komponenten, welche Duplikate ohne Probleme empfangen können. Eine Verbindung zu *Exactly-Once Delivery* bietet sich an. Wichtig ist, dass bei *Exactly-Once Delivery* die Nachricht nur ein einziges Mal zugestellt werden soll. Die Elimierung von Duplikaten passiert also auf dem Weg der Nachricht von selbst. Bei *Idempotent Receiver* kann die Nachricht mehrmals gelesen werden, was darauf schließen lässt, dass die Nachricht öfters gesendet werden musste. Ein *Idempotent Receiver* kann durch de-duping Duplikate, ähnlich wie bei *Message Filter*, entfernen [Gre04]. Duplikate werden also erst an der Komponente selbst eliminiert. Bei beiden Patterns ist die Nachricht letztendlich nur noch einmal vorhanden.

At-Least-Once Delivery

Es ist wichtig, dass Nachrichten überhaupt ankommen, die Anzahl spielt dabei keine Rolle. Nachrichten werden solange gesendet, bis die empfangende Komponente eine Bestätigung (engl. *Acknowledgement*) zurücksendet [FLR+14].

Transaction-Based Delivery

Exactly-Once Delivery oder *At-Least-Once Delivery* sorgen dafür, dass die Nachricht ankommt. Ob die Nachricht jedoch erfolgreich von einer Komponente gelesen wurde, wird über *Transaction-Based Delivery* sichergestellt. Das Lesen und Löschen einer Nachricht aus einer Queue fällt unter einer Transaktion. Beide Operationen müssen erfolgreich durchgeführt werden [FLR+14].

Fehling et al. [FLR+14] verweisen auf das *Transactional Client* Pattern von Hohpe und Woolf. Ein *Transactional Client* kann die Transaktionsgrenzen selbst setzen, weswegen er auch die *Transaction-Based Delivery* abdecken kann. Auf Empfängerseite wird bestätigt, wenn die Nachricht erfolgreich gelesen wurde.

Ein *Transactional Client* ist jedoch nicht nur auf diese Aufgabe beschränkt. Auch die Senderseite kann die Transaktionsgrenzen setzen. So kann bspw. das Senden einer Nachricht unter einer Transaktion stehen. Eine Nachricht wird erst dann auf einen Channel gesetzt, wenn der Sender die Transaktion committet. So kann sichergestellt werden, dass die Nachricht richtig gesendet wurde [Gre04].

Timeout-Based Delivery

Nachrichten einer Message Queue sollen erst dann gelöscht werden, wenn sie von einer Komponente erfolgreich gelesen wurden. Liest eine Komponente eine Nachricht, wird sie innerhalb der Queue als unsichtbar gekennzeichnet, sodass andere Komponenten diese nicht lesen können. Wird innerhalb eines Zeitintervalls keine Bestätigung gesendet, wird die Nachricht für andere Komponenten sichtbar gemacht [FLR+14].

Loose Coupling

Die Abhängigkeiten zwischen verteilten Komponenten sollen reduziert werden. Komponenten kommunizieren über einen Broker, also einen Vermittler, wie bspw. *Message-oriented Middleware* oder einem Enterprise Service Bus. Dadurch sind die Annahmen, die Komponenten einander nehmen, reduziert [FLR+14].

Fehling et al. [FLR+14] verweisen auf das *Broker* Pattern von Buschmann et al. In einem verteilten System kümmert sich das *Broker* Pattern um die Kommunikation von einzelnen Komponenten. Komponenten können sich über den Broker, zur Laufzeit, an- oder abmelden. Benötigen Komponenten Services von anderen Komponenten, können sie auf diese Funktionen über den Broker zugreifen [BMR+96].

Befinden wir uns im Kontext des Messaging und verwenden eine *Message-oriented Middleware* als Broker, verlinken Fehling et al. [FLR+14] auf die *Enterprise Integration Patterns*. In diesem Kontext können alle Messaging Pattern referenziert werden, da sich diese mit der losen Kopplung von Anwendungskomponenten beschäftigen.

Distributed Application

Die Funktion einer Anwendung wird in mehrere einzelne Komponenten aufgeteilt. Komponenten sind unabhängig voneinander und werden miteinander verbunden, bspw. über Messaging, um die verteilte Anwendung zu realisieren. Die Aufteilung erfolgt bspw. in Schichten, in Prozessen oder auch als Pipes-and-Filters Struktur [FLR+14].

Pipes and Filters ist ein Architekturpattern aus EIP, welches genau dies beschreibt, weswegen Fehling et al. [FLR+14] darauf verweisen. Komponenten sind Filter, welche einen Input annehmen und einen Output abgeben können. Die Filter sind über Pipes miteinander verbunden.

Das *Pipes and Filters* Pattern findet sich ebenfalls in den Pattern-Oriented Software Architecture Pattern wieder. Wichtig ist, dass bei dieser Variation nicht von Messaging die Rede ist. Der Datenaustausch geschieht bspw. über Files und nicht über konkrete Nachrichten wie in der EIP Version. Weiter gibt es keine Router-Komponenten, welche den Bearbeitungspfad beeinflussen könnten. D.h. die Bearbeitung der Daten hat immer denselben Weg und ist unabhängig von den konkreten Daten [BMR+96].

Die Aufteilung einer Anwendung, entsprechend der Prozesse, findet sich auch im *Process Manager* Pattern der EIP. Ein *Process Manager* dient als zentrale Einheit, welche den Ablauf einer Nachricht regelt. Prozesse, also eine Folge von Bearbeitungsschritten, lassen sich in diesem definieren. In hub-and-spoke Manier werden Komponenten aufgerufen und auf Basis der Rückgabe entschieden, wohin die Nachricht als nächstes soll [Gre04].

Aufteilung einer Anwendung in Schichten ist Bestandteil des *Layers* Pattern von Buschmann et al. Komponenten, die ähnliche Aufgaben haben, werden in einer Schicht gruppiert. Die Schichten werden anhand ihrer Funktionalitäten sortiert. High-Level-Funktionen finden sich in der obersten Schicht und Low-Level-Funktionen entsprechend in der untersten. Als High-Level zählt bspw. UI, während Low-Level bspw. die Daten umfasst. Eine Schicht kann nur mit der von ihr aus oberen und unteren Schicht kommunizieren, um die Abhängigkeiten zwischen den Schichten möglichst gering zu halten [BMR+96].

Stateful Component

Anwendungskomponenten speichern einen Zustand. Haben wir in einem verteilten System mehrere Instanzen derselben Komponente, muss der Zustand auf allen Instanzen gleich und somit synchronisiert werden, damit alle Instanzen gleich arbeiten können. Das Updaten des Zustands kann entweder in *Strict Consistency* oder über *Eventual Consistency* erfolgen [FLR+14].

Die Enterprise Integration Patterns umfassen mehrere Patterns, bei denen es sich ebenfalls um *Stateful Components* handelt. Ein Beispiel wäre das *Aggregator* Pattern. Dabei handelt es sich um einen zustandsorientierten Filter [Gre04]. Damit er die Nachrichten zusammensetzen kann, muss der *Aggregator* eingehende Nachrichten speichern und hat somit einen Zustand. Erst wenn alle zusammengehörende Nachrichten beim *Aggregator* vorliegen, kann die Antwort erstellt werden.

Auch beim *Resequencer* handelt es sich um einen zustandsorientierten Filter [Gre04]. Damit die Reihenfolge der Nachrichten wiederhergestellt werden kann, müssen spätere Nachrichten solange aufbewahrt werden, bis frühere Nachrichten eintreffen. Frühere Nachrichten werden dann gefolgt von

den späteren Nachrichten versendet. Mit früher und später sind hier die ursprünglichen Reihenfolgen der Nachrichten gemeint. Eine frühere Nachricht ist eine Nachricht, die ursprünglich früher als eine andere gesendet wurde. Sie muss jedoch nicht früher beim *Resequencer* eintreffen.

Stateless Component

Komponenten speichern ihren Zustand in externen Speicher. Laufen mehrere Instanzen, müssen diese den Zustand nicht mehr synchronisieren und können über eine ID auf den Zustand zugreifen [FLR+14].

Wie der Zustand in externen Speicher ausgelagert werden kann, behandeln die Session State Patterns von Fowler. Beim *Client Session State* Pattern wird der Zustand auf Clientseite gespeichert. Bei jeder Anfrage wird der Zustand mitgesendet, damit die Komponenten diesen nicht selbst speichern. Änderungen des Zustands müssen wieder mit zurückgesendet werden [Fow02].

Server Session State speichert den Zustand an der entsprechenden Komponente. Da die Komponente dadurch zu einer *Stateful Component* wird, eignet sich dieses Pattern nicht für diesen Zweck [Fow02].

Der Zustand einer Session wird bei *Database Session State* in einer Datenbank gespeichert. Die Komponente, die den Zustand benötigt, kann diese aus der Datenbank laden und bearbeiten. Anschließend wird der Zustand aktualisiert. Die Komponente selbst ist nicht im Besitz des Zustands und ist somit zustandslos. Ein Client muss bei einer Anfrage jedoch eine Session-ID o.Ä. mitsenden, mit der die Komponente den entsprechenden Zustand aus der Datenbank anfragen kann [Fow02].

User Interface Component

User Interface Components sind zustandslose Komponenten, auf denen Benutzer operieren können. Die Logik findet asynchron im Hintergrund auf anderen Komponenten statt. Die Konfigurationen des User Interface also bspw. Sprache, Schriftarten, etc. werden, ähnlich dem State, extern gespeichert [FLR+14].

Fehling et al. [FLR+14] verweisen in diesem Kontext auf die Web Presentation Patterns der Enterprise Application Architecture Patterns. Diese können zur konkreten Umsetzung der *User Interface Component* beitragen. Auf die Patterns wird im Detail im entsprechenden Abschnitt der Fowler Patterns eingegangen, weswegen diese hier nicht weiter ausgeführt werden.

Processing Component

Processing Funktionalitäten der Anwendung werden in einzelne Komponenten, den *Processing Components*, geteilt. Durch die Aufteilung wird das Skalieren der Anwendung ermöglicht. Eine Komponente hat eine einzige spezifische Funktion [FLR+14].

Wie die *Processing Components* umgesetzt werden, hängt von der konkreten *Distributed Application* ab. Die Aufteilung der Anwendung in einzelne Komponenten variiert je nach Architekturstil. Im Rahmen des *Pipes and Filters* Patterns sind die Filter die einzelnen Bearbeitungskomponenten [FLR+14]. Für diesen Stil haben wir eine Verlinkung zum *Processing Component* Pattern.

Verarbeiten mehrere *Processing Components* Nachrichten derselben Queue, verweisen Fehling et al. [FLR+14] auf *Competing Consumer* der EIP. Dadurch, dass die Komponenten selbst bestimmen, wann sie die Nachrichten entgegennehmen, entsteht ein Load Balancing. Nur Komponenten, die frei sind, nehmen sich die Nachrichten aus der Queue heraus und bearbeiten diese.

Batch Processing Component

Bearbeitungsanfragen werden in der *Batch Processing Component* gesammelt. Erst wenn bestimmte Bedingungen erfüllt sind, werden die gesammelten Anfragen von *Processing Components* verarbeitet. Eine Bedingung könnte z.B. die Anzahl gesammelter Nachrichten sein [FLR+14].

Data Access Component

Versteckt die Komplexität des Datenzugriffs, indem sie den Zugriff der Daten vereinheitlicht. Die Komplexität befasst sich auf verschiedene Datastores, die im Hintergrund verwendet werden, Authentifizierungen, etc. Komponenten fragen nicht bei den tatsächlichen Datastores an, sondern greifen über die *Data Access Component* auf diese zu[FLR+14].

Fehling et al.[FLR+14] verweisen an dieser Stelle auf die Patterns von Fowler. Unter diesen finden sich verschiedenen Datenzugriffspatterns, welche für die Umsetzung der *Data Access Component* beitragen können. Darunter fällt bspw. das *Table Data Gateway* Pattern. Damit operiert man an einer Datenbanktabelle über bereitgestellte Methoden, anstelle direkt SQL Befehle auszuführen [Fow02].

Die *Data Access Component* isoliert Komplexität und bietet einheitlichen Zugriff auf Daten. Um einheitlichen Zugriff zu ermöglichen, muss die *Data Access Component* mit verschiedenen Interfaces zurechtkommen. Eine Komponente, welche die Inkompatibilität von Schnittstellen überbrückt, ist ein *Adapter*. Das *Adapter* Pattern der GoF implementiert jene Schnittstelle, die ein Client erwartet und führt im Hintergrund die Funktionen der Zielkomponente aus [GHJV15]. In diesem Fall ist der Client eine beliebige Komponente, welche auf die *Data Access Component* zugreift. Die *Data Access Component* agiert als *Adapter* und die Zielkomponente ist das dahinterliegende Datastore, auf dem die Daten liegen.

Das *Proxy* Pattern von Buschmann et al. [BMR+96] definiert, dass Clients über einen Stellvertreter kommunizieren, anstelle an der Komponente direkt zu arbeiten. Gründe dafür sind bspw. dass Änderungen der Komponenten keinen Einfluss auf die Clients haben soll. Desweiteren aus Sicherheitsgründen: Ein Client soll nicht direkt auf der Komponente arbeiten können, um mögliche Angiffe auf das System zu verhindern. Beide Gründe finden sich auch in der *Data Access Component* wieder, weswegen Fehling et al. [FLR+14] auf das *Proxy* Pattern verweisen.

Der Unterschied von *Adapter* und *Proxy* liegt in dem Interface, welches die Komponente bereitstellt. Bei *Proxy* hat die Komponente dasselbe Intrface wie der dahinterliegende Service [BMR+96]. Bei *Adapter* hat die Komponente das Interface, welches der Client erwartet und passt die Funktionalität entsprechend dem Service an [GHJV15]. Das *Data Access Component* Pattern kann jedoch Gebrauch beider Pattern machen. Schnittstellen der Datastores können beliebig sein, weswegen die *Data Access Component* als *Adapter* fungiert und das Interface der Clients implementiert, welches die verschiedenen Schnittstellen vereinheitlicht. Auf der anderen Seite könnten *Stateful Component*,

anstelle von Datastores, die datenhaltigen Komponenten sein, welche eine Schnittstelle bereitstellen [FLR+14]. Die *Data Access Component* kann hier dieselbe Schnittstelle nach außen bieten und gilt dann als *Proxy*.

Data Abstractor

Daten aus Eventual Consistency können nicht aktuell sein. Ein Data Abstractor greift auf die Daten zu und liefert eine abstraktere Sicht auf den Zustand der Daten. Der aktuelle Zustand wird mit evtl. alten Daten approximiert. Als Beispiel dienen Online Versandhäuser, welche die Menge der verfügbaren Einheiten einer Ware mit "begrenzt verfügbaränstelle einer exakten Zahl kennzeichnen [FLR+14].

Idempotent Processor

Ein Processor, der so konzipiert ist, dass mehrere gleiche Messages oder inkonsistente Daten die Komponente nicht zum Ausfall bringen, ist ein *Idempotent Processor*. Duplikate bei den Nachrichten können entweder gefiltert werden, oder die Funktionen werden so ausgelegt, dass die Eingabe mehrerer gleicher Eingaben zum immer gleichen Ergebnis führen. Inkonsistente Daten, welche bei *Eventual Consistency* aufgrund der Updates auftreten, müssen bei erneuten Updateanfragen berücksichtigt werden [FLR+14].

Message Filter können Duplikate auf dem Nachrichtenweg filtern, deswegen der Verweis von Fehling et al. [FLR+14].Eingehende Nachrichten werden auf ihre ID überprüft, bereits gesehene Nachrichten werden aussortiert. Dieser Sachverhalt wurde bereits unter dem *Exacly-Once Delivery* Pattern beleuchtet. Hinzu kommt jedoch noch die Betrachtung der inkonsistenten Daten, auf welche nachfolgend genauer eingegangen wird. Bei Anfragen auf einen Datastore mit *Eventual Consistency* während dem Propagieren der Änderungen, kann es vorkommen, dass obsolete Daten gelesen werden, d.h. Daten die gerade geupdatet werden. Daten bekommen einen Versions-Identifier, welcher aussagt, um welche Version der Daten es sich handelt. Werden Daten bearbeitet, ändert sich der Versions-Identifier. Der *Message Filter* vergleicht die Versions-Identifier der Daten und sorgt, dass keine alten Daten weitergeleitet werden.

Fehling et al. [FLR+14] verweisen auf das *Idempotent Receiver* Pattern im Messaging Kontext. Dieses behandelt jedoch Nachrichten-Duplikate und nicht die Konsistenz von Daten aus Datastores. Der *Idempotent Processor* erweitert somit die Konzepte des *Idempotent Receiver* [FLR+14].

Obige Abschnitte behandeln die Inconsistency Detection, also das Erkennen und Filtern von Duplikaten. Eine andere Variante des *Idempotent Processors* ist, die Komponente so zu gestalten, dass diese Duplikate ohne Probleme bearbeiten kann. In der Mathematik ist eine Funktion idempotent, wenn gilt: $f(f(x)) = f(x)$. Das mehrmalige Ausführen der Funktion mit gleichem Input bewirkt gleichen Output. Ein Beispiel für eine solche Funktion findet sich in Setter-Methoden von Objekten. `setValue(10)` kann beliebig oft ausgeführt werden. Der Wert der Variable `value` ist im Endeffekt immer `10`. Entscheidend ist jedoch die Reihenfolge der Nachrichten. Haben wir bspw. erst `setValue(10)` und dann `setValue(20)`, soll der Wert von `value` am Ende `20` sein. In Messaging-Systemen ist die Reihenfolge der Nachrichten nicht gewährleistet [Gre04]. Aus diesem Grund verweisen Fehling et al. [FLR+14] auf den *Resequencer* der EIP, welcher dafür sorgt, dass die Nachrichten in der richtigen Reihenfolge eintreffen.

Transaction-Based Processor

Transaction-Based Delivery sorgt dafür, dass eine Komponente die Nachricht erhält. Fällt die Komponente beim Bearbeiten jedoch aus, geht die Nachticht verloren. *Transaction-Based Processor* erweitert den Transaktionskontext. Erst wenn die Nachricht bearbeitet wurde, ist die Transaktion fertig und die Nachricht kann aus der Queue gelöscht werden [FLR+14].

Hierbei handelt es sich um eine ähnliche Arbeitsweise wie beim *Transactional Client* Pattern der EIP. Die Ähnlichkeit verhält sich so wie beim *Transaction-Based Delivery* Pattern, weswegen sie hier nicht erneut aufgeführt wird.

Timeout-Based Message Processor

Erweitert *Timeout-Based Delivery* dahingehend, dass *Timeout-Based Message Processor* erst eine Bestätigung bzw. Acknowledgement sendet, wenn die Nachricht bearbeitet wurde. Wird die Nachricht nach einer definierten Zeit nicht erfolgreich bearbeitet, wird sie sichtbar gemacht, sodass andere Komponenten diese Nachricht bearbeiten können [FLR+14].

Multi-Component Image

Ein Image enthält mehrere Komponenten. Ein virtueller Server hostet diese Komponenten je nach Bedarf. Werden bestimmte Komponenten nicht benötigt, sind sie inaktiv und die verbleibenden Ressourcen können anders verwendet werden [FLR+14].

Shared Component

Mehrere Benutzer bzw. Benutzergruppen teilen sich eine gemeinsame Komponente. Die Funktion der Komponente ist für alle Benutzer gleich. Konfigurationen können mit der Anfrage mitgeschickt werden, um Individualität zu unterstützen [FLR+14].

Tenant-isolated Component

Komponenten werden unter mehreren Benutzergruppen, also Tenants, geteilt, diese sollen sich jedoch nicht gegenseitig beeinflussen können. Komponenten des gesamten Anwendungsstacks müssen so erstellt werden, dass sie mit Tenant-IDs umgehen können [FLR+14].

Dedicated Component

Eine Komponente wird exklusiv von einem Tenant verwendet. Der spezielle Tenant wird anhand seiner Tenant-ID ausgemacht. Im System können jedoch weiterhin Komponenten verwendet werden, welche unter den Benutzergruppen geteilt werden. Die *Dedicated Component* hingegen bleibt unter alleiniger Verwendung eines Tenants [FLR+14].

Restricted Data Access Component

Unsichere Umgebungen sollen keinen Zugriff auf die kompletten Daten haben. Eine *Restricted Data Access Component* kümmert sich darum. Daten werden für diese Umgebungen bspw. verschleiert, indem Werte durch andere Inhalte ausgetauscht werden. Das Mapping wird gespeichert, damit die Daten in der sicheren Umgebung wieder zurückgewandelt werden können [FLR+14].

Verlinkungen zu *Adapter* der GoF [GHJV15] und *Proxy* von Buschmann et al. [BMR+96] wurden bereits beim *Data Access Component* genauer betrachtet. Die gleichen Konzepte lassen sich auch auf die *Restricted Data Access Component* übertragen, weswegen sie hier nicht erneut gelistet werden.

Fehling et al. [FLR+14] gehen weiter auf die *Remote Facade* der EAA Patterns ein. Sie merken an, dass die Verwendung der *Remote Facade* die Effizienz beim Abrufen der Daten verbessern kann. Durch eine *Remote Facade* werden feingranulare Objekte gebündelt, um die Aufrufe zu minimieren [Fow02]. Dieses Pattern eignet sich generell für die Übertragung von Daten innerhalb des Netzwerkes und nicht speziell für *Restricted Data Access Component*, weswegen dies nicht weiter vertieft wird.

Message Mover

Nachrichten aus einer Cloud Umgebung soll auch in anderen Umgebungen verfügbar sein, wenn diese gemeinsam verwendet werden. Ein *Message Mover* integriert Message Queues und verschiebt Nachrichten aus einer Umgebung in eine andere, ohne dass Komponenten davon Kentnis nehmen [FLR+14].

Eine *Messaging Bridge* aus den EIP verbindet mehrere Messaging Systeme miteinander. Damit haben wir eine direkte Verbindung zu *Message Mover*, welches das gleiche Problem beschreibt. Die *Messaging Bridge* kopiert Nachrichten aus einem Channel in die entsprechenden Channels der anderen Messaging Systeme [Gre04].

Fehling et al. [FLR+14] gehen noch speziell auf die möglichen Unterschiede der Nachrichtenformate ein. In den Hohpe und Woolf Patterns finden sich unter dem *Message Translator* mehrere Pattern, welche sich diesem Problem widmen. Die Unterschiede der Nachrichtenformate entstehen auf Grund der Verwendung mehrere Messaging Systeme, welche ursprünglich unabhängig voneinander verwendet wurden und durch den *Message Mover* zusammengeführt werden.

Application Component Proxy

Komponenten können in einer Umgebung liegen, auf die wir keinen Zugriff haben. Ein *Application Component Proxy* hat dasselbe Interface wie die gewünschte Komponente und ermöglicht den Zugriff auf diese von außerhalb. Anstelle mit der Komponente direkt zu kommunizieren, findet die Kommunikation über den Proxy statt [FLR+14].

Die Ähnlichkeit zum *Proxy* Pattern sowie dem *Facade* Pattern der GoF wird von Fehling et al. [FLR+14] an dieser Stelle nähergebracht. Anstelle mit einer Komponente direkt zu interagieren wird ein *Proxy*, also ein Stellvertreter verwendet, welches dasselbe Interface wie die gewünschte Komponenente bereitstellt. Eine *Facade* vereinfacht Schnittstellen, indem es die einzelnen Komponenten

unter einem einzigen Interface bündelt [GHJV15]. Die Abhängigkeit zu mehreren Komponenten wird dadurch reduziert, da die Kommunikation nun über die *Facade* läuft. Ein *Application Component Proxy* kann im Kontext einer *Facade* also das Interface für mehrere dahinterliegende Komponenten sein. Das *Proxy* Pattern findet sich auch in den Buschmann et al. Patterns [BMR+96]. Bei diesem handelt es sich jedoch um dasselbe Pattern, welches bereits von den GoF beschrieben wurde.

Compliant Data Replication

Wir haben Replika von Daten in unsicheren Umgebungen. Daten dürfen auf diese Replika, aufgrund von Gesetzen oder Unternehmensvorschriften, nicht eins zu eins gespeichert werden. Beim Updaten der Replika auf Basis von *Eventual Consistency* werden Daten beim Übergehen in die unsichere Umgebung gemäß den Regeln gefiltert [FLR+14].

Da es hierbei um das Entfernen und Hinzufügen von Daten geht, verweisen Fehling et al. [FLR+14] auf die *Message Filter* und *Message Enricher* Patterns aus den EIP. Genauer handelt es sich um die Pattern *Content Filter* und *Content Enricher*. Beim *Content Filter* werden Inhalte einer Nachricht entsprechend gesetzter Regeln gefiltert. Der gefilterte Teil wird in einem Datastore abgelegt, damit dieser im späteren Verlauf der Nachricht ergänzt werden kann. Die reduzierte Nachricht kann dann zur unsicheren Umgebung gesendet werden [Gre04]. Trifft die Nachricht aus der unsicheren Umgebung wieder in die Sichere ein, müssen gefilterte Daten in die Nachricht zurück. Der *Content Enricher* nimmt sich die entsprechenden Teildaten aus dem Datastore heraus und fügt sie der Nachricht an [Gre04].

Integration Provider

Anwendungskomponenten laufen in verschiedenen Umgebungen, welche durch einen *Integration Provider* integriert werden. Dieser third-party Provider stellt Messaging-Funktionalitäten und geteilten Speicher bereit [FLR+14].

Provider Adapter

Um Anwendungskomponenten und Anbieterfunktionalitäten zu entkoppeln, stellt ein *Provider Adapter* eine einheitliche Schnittstelle bereit, auf welcher die Komponenten arbeiten. Im Hintergrund kümmert sich der *Provider Adapter* darum, dass die richtige Implementierung eines Anbieters verwendet wird. Der Austausch von Anbietern betrifft nicht die Komponenten, sondern lediglich den *Provider Adapter* [FLR+14].

Ein *Provider Adapter* ist eine Abstrahierungsschicht des Systems und stellt Schnittstellen zur Verfügung. Fehling et al. [FLR+14] verweisen auf Konzepte des *Adapter* und *Facade* Pattern der GoF. Das *Proxy* Pattern von Buschmann et al. [BMR+96] beschreibt ebenfalls Konzepte, welche sich hier anwenden lassen.

Die *Data Access Component* ist bspw. ein Implementierung des *Provider Adapter* für Datenspeicher. Verbindungen zu den oben genannten Pattern finden sich im Detail unter dem *Data Access Component* Pattern, weswegen sie hier nicht wiederholt werden.

Managed Configuration

Diverse Konfigurationen wie bspw. User Interface Einstellungen oder Standorte von Datenkomponenten werden außerhalb einer Komponente zentral gespeichert. Änderungen in der Konfiguration müssen somit nicht über einzelne Instanzen der Komponente synchronisiert werden. Stattdessen können die Änderungen in Intervallen speziell nachgefragt (poll) werden oder der Configuration-Manager selbst sendet die Änderungen an die Instanzen (push) [FLR+14].

Der Configuration-Manager kann die Änderungen an interessierte Komponenten senden. Diese Funktion kann über einen *Publish-Subscribe Channel* der EIP realisiert werden, so Fehling et al. [FLR+14]. Die Komponenten subscriben den Channel und erhalten dann die Änderungen über *Event Messages*.

Elasticity Manager

Die Nutzung von Komponenten wird von einem *Elasticity Manager* überwacht. Je nach Auslastung werden weitere Komponenten gestartet oder beendet. Dabei interagiert der Manager über die *Elastic Platform* oder *Elastic Infrastructure* [FLR+14].

Elastic Load Balancer

Ein *Elastic Load Balancer* regelt die Anzahl der Anwendungskomponenten über die Anzahl synchron eingehender Anfragen auf das System. Die Komponenten werden über die *Elastic Platform* oder *Elastic Infrastructure* skaliert [FLR+14].

Elastic Queue

Asynchrone Anfragen treffen in einer *Elastic Queue* ein. Diese entscheidet anhand der Anzahl der Nachrichten innerhalb der Queue, wie viele Anwendungskomponenten benötigt werden. Ist das System ausgelastet und es können keine weiteren Komponenten mehr erworben werden, sorgt die *Elastic Queue* dafür, dass höher priorisierte Nachrichten den Vorrang haben [FLR+14].

Watchdog

Komponenten können ausfallen. Ein *Watchdog* überwacht die Komponenten auf ihre Funktionsfähigkeit über Heartbeat, Netzwerkdaten aus der Infrastruktur oder über Test-Nachrichten. Ist eine Komponente fehlerhaft, wird sie ausgetauscht [FLR+14].

Fehling et al. [FLR+14] verweisen auf die Patterns *Control Bus*, *Test Message* und *Wire Tap* der EIP. In Kombination haben sie die gleiche Funktionsweise wie der *Watchdog*. Im Folgenden wird das Zusammenspiel genauer erklärt.

Beim *Control Bus* werden die einzelnen Filter um eine weitere Verbindung für Kontrolldaten erweitert. Über diese Verbindung werden Informationen gesendet, die nicht für die Anwendungslogik, sondern für das Management wichtig sind. Darunter zählen auch Heartbeats, welches kleine Nachrichten sind, die signalisieren, dass die Komponente noch verfügbar bzw. erreichbar ist. Dies zählt als passiver Check, da dabei Daten von der Komponente bereitgestellt werden [Gre04].

Das *Test Message* Pattern ist ein aktiver Monitoring Mechanismus, welcher eine Komponente auf ihre Funktionalität testet, nicht auf ihre Verfügbarkeit. Das Pattern besteht aus vier Komponenten, welche dieses realisieren. Der *Test Data Generator* erstellt Testdaten. Ein *Test Message Injector* verpackt die Daten in eine Nachricht und versendet diese. Zusätzlich sorgt sie dafür, dass die Nachricht als Testnachricht gekennzeichnet ist, bspw. über ein spezielles Feld im Header der Nachricht. Der *Test Data Separator* kümmert sich darum, dass die Antworten der zu testenden Komponente keine Auswirkungen im System hat, indem sie die Antwort auf die Testnachricht an den Verifier leitet. Dabei hilft die Markierung des Message Injectors. Es ist wichtig, dass die Testnachricht keine realen Auswirkungen auf das System hat. Werden bspw. Bestellungen überprüft, dürfen diese Testbestellungen nicht wirklich bestellt werden. Der *Test Data Verifier* vergleicht Ist und Soll Antworten der Komponente. Stimmen die Daten nicht überein, haben wir eine fehlerhafte Komponente [Gre04].

Die Daten verlaufen trotz verfügbarem *Control Bus* über den normale Anwendungskanal. Jenem über dem die anderen Nachrichten auch verlaufen. Aus diesem Grund ist die Separation so wichtig. Die Überprüfung läuft nicht auf dem *Control Bus*, da es sich bei den Nachrichten nicht um Managementdaten handelt. Die Nachrichten könnten echte Nachrichten sein, werden aber zum Testen verwendet. Die Definition des Control Busses verschmilzt an dieser Stelle mit dem Nachrichtenfluss der Anwendung, weswegen beide Messaging Systeme verwendet werden [Gre04].

Wird eine fehlerhafte Komponente erkannt, kann ein Fail Over stattfinden. Über den Control Channel informiert der Verifier eine *Management Console*. Diese sendet Steuerungsdaten an einen Router, welcher dafür sorgt, dass Nachrichten an eine Ersatzkomponente geleitet werden. Die neue Komponente kann jedoch den gleichen Antwortchannel verwenden [Gre04].

Wichtig für dieses Zusammenspiel ist, dass wir uns im Kontext der *Pipes-and-Filters* Architektur via Messaging bewegen. Das *Watchdog* Pattern beschreibt die Überwachung und den Austausch in einer *Distributed Application* generell. Es hängt nicht von der konkreten Umsetzung bspw. über Messaging ab. Für andere Implementierung benötigt man andere Komponenten.

Elasticity Management Process

Entsprechend der Arbeitsauslastung werden Instanzen von Komponenten hinzugefügt oder entnommen. Der *Elasticity Management Process* bedient sich dabei dem *Elasticity Manager*, dem *Elastic Load Balancer* sowie der *Elastic Queue*. Es lässt sich jedoch auch manuell einstellen, ob weitere bzw. weniger Ressourcen benötigt werden [FLR+14].

Feature Flag Management Process

Erfahren wir eine hohe Arbeitsauslastung und es können keine weitern Ressourcen angefordert werden, sorgt *Feature Flag Management Process* dafür, dass weniger wichtige Komponenten deaktiviert werden. Dadurch sind entsprechende Ressourcen frei und können für die wichtigen Komponenten verwendet werden [FLR+14].

Update Transition Process

Anwendungskomponenten können Updates erhalten. Laufende Komponenten können jedoch nicht einfach ausgeschalten und durch neue Vesionen ausgetauscht werden. Stattdessen erfolgt der Austausch wie folgt: Werden im Kontext des *Elasticity Managers* weitere Instanzen der Komponente angefordert, werden diese aus dem neuen Image, welches die Updates enthält, erstellt. Bei Freigabe von Ressourcen werden die Instanzen mit der alten Version aus dem System genommen [FLR+14].

Standby Pooling Process

Das Anfordern neuer Instanzen einer Komponente dauert Zeit. Um diesen Prozess zu beschleunigen werden nicht-verwendete Instanzen nicht ausgeschalten, sondern lediglich in einen Standby-Zustand versetzt. Benötigen wir Instanzen, werden diese aus dem Standby-Zustand herausgeholt und betrieben [FLR+14].

Resiliency Management Process

Beim *Resiliency Management Process* werden Anwendungskomponenten auf ihre Funktionalität überwacht. Fehlerhafte Komponenten werden automatisch, d.h. ohne manuelles Eingreifen, ausgetauscht. Der Prozess wird von einem *Watchdog* ausgeführt [FLR+14].

Dieses Pattern ist eine Abstraktion des *Watchdog* Patterns, weswegen Fehling et al. [FLR+14] erneut auf Control Bus und Test Message der EIP verweisen. Das Zusammenspiel wurde bereits oben ausgeführt und wird hier nicht wiederholt.

Two-Tier Cloud Application

Die Anwendung wird in zwei Ebenen, engl. Tiers, aufgeteilt. Präsentation- und Geschäftslogik, beides *Stateless*, finden sich gemeinsam auf einer Ebene. Auf der zweiten Ebene befinden sich die Daten. Die Aufteilung ermöglicht separates Skalieren [FLR+14].

Three-Tier Cloud Application

Die Anwendung wird in Präsentation, Geschäftslogik und Daten Ebene aufgeteilt. Die Aufteilung ermöglicht individuelles Skalieren. Der Unterschied zur *Two-Tier Cloud Application* liegt darin, dass rechenintensive Anwendung mit komplexer Geschäftslogik unabhängig des User Interfaces skaliert werden kann [FLR+14].

Content Distribution Network

Zugriffe auf Datencenter welche physikalisch weiter von einem weg liegen, dauern länger. Vor allem bei größeren Dateien wie Videos. Bei *Content Distribution Network* werden Datenreplikate an verschiedenen globalen Standorten platziert. Ein User wird zu dem Replikat verbunden, der die beste Verbindung ermöglicht [FLR+14].

Hybrid User Interface

User Interface Komponenten werden bei *Hybrid User Interface* in der Elastic Cloud angeboten, während die Anwendung selbst auf statischer Seite verbleibt. Je nach Arbeitsauslastung kann die UI entsprechend skaliert werden, während die Bearbeitung in sicherer Umgebung stattfindet. Da die statische Umgebung nur bedingt skalierbar ist, können Benutzer bspw. via E-Mail informiert werden, wenn die Bearbeitung fertig ist [FLR+14].

Hybrid Processing

In *Hybrid Processing* finden sich Teile der *Processing Components* in der Elastic Cloud, während die restliche Anwendung auf statischer Seite verbleibt. Informationen und Daten, welche die Bearbeitungskomponenten benötigen, finden sich in der Anfrage-Nachrichten. Die ausgelagerten Komponenten haben keinen direkten Zugriff auf die Daten der Anwendung [FLR+14].

Hybrid Data

Der Datenspeicher, welcher Daten variierender Größe speichert, wird außerhalb der Anwendung in einer Elastic Cloud gehostet. Je nach Bedarf lässt sich dann der Datenspeicher dort skalieren [FLR+14].

Hybrid Backup

Ein Backup der Daten einer Anwendung liegt in der Elastic Cloud. Periodisch, bspw. bei geringer Systemauslastung, werden die Daten der Anwendung auf das Backup repliziert. Bei Ausfällen können die Daten aus der Cloud zur Wiederherstellung beitragen [FLR+14].

Hybrid Backend

Teile des Backends mit variierender Arbeitsauslastung werden in die Elasitc Cloud ausgelagert. Unter Backend fallen Bearbeitungseinheiten und der von denen verwendete Datenspeicher [FLR+14].

Hybrid Application Functions

Teile der Anwendung, bestehend aus UI, Bearbeitungskomponenten sowie Speicher, finden sich in der Elasitc Cloud, damit sie gemäß deren Auslastung skaliert werden können. Auf statischer Seite finden sich Funktionalitäten gleicher Art bspw. wegen Sicherheitsvorschriften [FLR+14].

Hybrid Multimedia Web Application

Multimedia Inhalte wie Videos oder Musik liegen in der Elasitc Cloud und werden zu Clienten gestreamt. Die restliche Anwendung findet sich auf statischer Seite [FLR+14].

Hybrid Development Environment

Anwendungen unterlaufen verschiedene Stages: Development, Test und Production. Je nach Stage haben wir verschiendene Anforderungen an die Laufzeitumgebung für die Anwendung. Aufgrund der Flexibilität der Cloud kann die Umgebung für die Anwendung entsprechend angepasst werden [FLR+14].

3.2.3 Patterns of Enterprise Application Architecture

Nach Fowler [Fow02] gibt es keine genaue Definition von Architektur eines Systems. Als Architektur einer Software gelten alle Teile, die wichtig oder nur schwer änderbar sind. Ebenso verhält es sich mit dem Begriff „Enterprise Application“. Stattdessen gibt es gewisse Eigenschaften, welche eine Anwendung als solche auszeichnen lässt. Unter diesen Eigenschaften verstehen wir bspw. eine große Datenmenge, welche persistiert werden muss und von vielen Benutzern nebenläufig verwendet wird. Oder auch verschiedenste Benutzerschnittstellen, welche nach außen zeigen. Eine Enterprise Anwendung muss nicht immer eine große Anwendung sein [Fow02].

Eine Architektur muss für Anwendungen speziell angepasst werden. Es gibt keine Architektur, welche für alle Problemstellungen die beste ist. Es gilt die Herausforderungen der Anwendung zu identifizieren und die Architektur danach zu richten. Sie muss anhand von Anforderungen und Bedürfnissen der Anwendung justiert werden [Fow02].

Die Pattern von Fowler et al. geben Hilfestellungen auf verschiedene Aspekte der Achitektur von Enterprise Anwendungen. Sie bieten verschiedene Lösungsansätze, welche in Kombination oder auch als Alternativen verwendet werden können. Untersucht wurden mögliche Verbindungen zu den bereits behandelten Enterprise Integration Patterns von Hohpe und Woolf sowie den Cloud Computing Patterns von Fehling et al. Fowler geht an manchen Stellen selbst auf Verlinkungen zu anderen Patternsprachen ein, welche hier entsprechend genauer untersucht wurden.

Transaction Script

Die Business-Logik wird in einzelnen Prozeduren aufgeteilt. Jede Prozedur ist dabei ein Verbund von Domain-Logik-Schritten wie bspw. Validierungs- oder Kalkulationsschritte. Transaktionsgrenzen lassen sich bei Beginn und Ende der Prozedur setzen. Für jede Aktion, welche ein Client ausführen könnte, benötigen wir einen *Transaction Script* [Fow02].

Mehrere *Transaction Scripts* können sich entweder alle in einer Klasse finden oder wir haben eine Klasse pro Prozedur. Bei der zweiten Variante handelt es sich um eine Umsetzung des *Command* Patterns der GoF, weswegen Fowler [Fow02] auf dieses verweist. Wir erstellen eine Oberklasse, welche eine Methode für die Umsetzung der Funktionalität des *Transaction Scripts* bereitstellt. Unterklassen implementieren ihre konkreten Funktionen in diese Methode.

Domain Model

In einer objektorientierten Umgebung haben wir Objekte, welche Daten und Verhalten des Geschäftsbereichs modellieren. Beim Simple-Domain-Model werden die Daten der Objekte wie die Daten der Datenbank modelliert, beim Rich-Domain-Model sind sie beliebig. Die Herausforderung dieses Patterns ist es, wie die Daten und Verhalten in Objekte modeliert werden [Fow02].

Wichtig ist dabei, dass wir Daten in den Objekten integrieren. Werden nur Funktionen benötigt, empfiehlt sich das *Transaction Script* Pattern.

In diesem Pattern geht es um die Modellierung der Objekte. Im Enterprise Integration Patterns Kontext haben wir eine Gruppe von Patterns unter dem Message Construction Bereich. Dabei geht es darum, wie einzelne Nachrichten aus vorhandenen Daten aufgebaut werden können. Die Konzepte des *Domain Model* können hierauf übertragen werden.

Table Module

Eine gesamte relationale Datenbank Tabelle wird unter einer Objektinstanz verwaltet. Anstelle pro Eintrag, d.h. einer Zeile der Tabelle, ein Objekt erzeugt wird, verwendet man eine Instanz, welche auf der ganzen Tabelle operiert. Die Domain-Logik wird nicht in einzelne Prozeduren aufgeteilt. Stattdessen designt man die Logik um die Tabelle herum. *Table Module* arbeitet mit *Record Set* Objekten, welche bei Anfragen auf die Datenbank erzeugt werden [Fow02].

Service Layer

Ein Client interagiert über *Service Layer*, welches eine Menge an Applikations-Operationen umfasst. Wie bei *Transaction Script* und *Domain Model* geht es um die Organisation von Business-Logik. Intern können die Operationen innerhalb des *Service Layers* implementiert werden. Alternativ dient sie nur als Fassade, welche im Hintergrund mehrere *Domain Models* verwendet [Fow02].

Table Data Gateway

Anstelle die Datenbank Tabelle über SQL zu bearbeiten, nutzen wir ein *Table Data Gateway*, welches entsprechende Methoden bereitstellt. Empfohlen wird ein *Table Data Gateway* pro Tabelle oder View. Herausforderung ist hier, wie die Daten einer Abfrage zurückgegeben wird. *Table Data Gateway* eignet sich in Kombination mit *Table Module* [Fow02].

Das *Table Data Gateway* kann als Teil der *Data Access Component* der *Cloud Computing Patterns* verwendet werden. Das Gateway regelt den Zugriff auf die hinterliegende Datenbank. Die *Data Access Component* definiert über das *Table Data Gateway* eine Schnittstelle, mit welcher andere Komponenten auf die Daten zugreifen können. Diese Komponenten stellen ihre Anfragen nicht über SQL und müssen sich auch nicht mit datenbankspezifischen Anforderungen, wie das Aufbauen einer Verbindung kümmern.

Nach gleicher Argumentation gibt es eine Verbindung zu *Restricted Data Access Component*. Die Methoden des *Table Data Gateway* können die Daten vor der Rückgabe gemäß Regeln eingeschränkt werden, sodass Clients keinen Zugriff auf sämtliche Daten bekommen. Diese Filterung findet dann innerhalb der Komponente statt.

Row Data Gateway

Ein *Row Data Gateway* trennt Business-Logik vom Datenbank-Code. Der Unterschied zu *Table Data Gateway* liegt darin, dass einzelne Zeilen anstelle der gesamten Tabelle pro Instanz verwaltet werden. Jedes Feld entspricht dabei einer Spalte des Eintrags. Somit verhält sich der Zugriff auf die Daten wie der Zugriff auf ein Objekt, anstelle einer Datenbank [Fow02].

Active Record

Ein Objekt, welches neben der Datenbank-Zugriffslogik auch Domain-Logik enthält, ist ein *Active Record*. Die Datenstruktur hat den gleichen Aufbau wie der dazugehörige Datenbankeintrag. Ist die Business-Logik zu kompliziert, eignet sich das Pattern aufgrund der Kombination mit der Datenbank-Zugriffslogik nicht [Fow02].

Data Mapper

Mapper mappen die Daten aus Objekt-Sicht zu Datenbank-Sicht und vice versa und separiert Domain und Datenbank. Aus Objekt-Sicht existiert keine Datenbank. Empfehlenswert ist dieses Pattern, falls Objekt-Model und Datenbank-Schema sich unterschiedlich schnell entwickeln. Anpassungen sind dann nur im Mapper notwendig. In Verbindung mit *Domain Model* wandelt der *Data Mapper* die Daten aus der Datenbank in die Struktur des *Domain Model* [Fow02].

Unit of Work

Daten der Datenbank werden in Objekte, welche im Speicher liegen, bearbeitet. Damit Änderungen an dem Objekt zu Änderungen in der Datenbank führen, müssen die Daten wieder herausgeschrieben werden. Während einer Business-Transaktion hält ein *Unit of Work* alle Änderungen fest und kümmert sich um das Updaten der Datenbank [Fow02].

Identity Map

Lädt man gleiche Daten in mehrere Objekte, kann es bei Änderungen Inkonsistenz geben. Verschiedene Objekte haben verschiedene Änderungen, welche beim Übertragen in die Datenbank auf einen gemeinsamen Nenner gebracht werden müssen. Eine *Identity Map* hält Referenzen auf Objekte mit bereits gelesenen Daten, damit die Daten nicht in mehrere Objekte geladen wird. Über einen Lookup erhält man das entsprechende Objekt. Ist kein Objekt vorhanden, werden die Daten einmalig in ein Objekt geladen. Als Key der Map fungiert bspw. der Schlüssel aus der Datenbank [Fow02].

Die *Data Access Component* der Cloud Computing Patterns stellt ein Interface für den Zugriff auf die Daten bereit. Diese Schnittstelle kann entweder generisch oder für entsprechende Anforderungen angepasst sein [FLR+14]. Im letzteren Fall können innerhalb der *Data Access Component* Objekte aus den Daten der Datenbank erzeugt werden, welche dann an die Clients gesendet werden. Die Verwaltung dieser Objekte kann dann in einer *Identity Map* stattfinden. Bei einer generischen Schnittstelle müssen sich die Clients selbst um die Erzeugung von geeigneten Objekten kümmern. Im Zuge dessen können Clients eine *Identity Map* für sich halten, um die Objekte zu verwalten.

Lazy Load

Verlinkte Objekte gleich mitzuladen ist praktisch für einen Entwickler, da somit alle vermeintlich relevanten Daten verfügbar sind. Das Problem ist dabei, dass dadurch zu viele Daten geladen werden könnten und damit den Speicher unnötig füllen. *Lazy Load* lädt nur einen Teil der Daten. Bei Bedarf werden weitere Daten an der unterbrochenen Stelle nachgeladen [Fow02].

Bei *Lazy Load* über Lazy Initialization werden die Objektfelder vor der Verwendung auf null überprüft. Ist ein Feld null, wird es jetzt geladen und anschließend zurückgegeben, damit es verwendet werden kann. Dieses Vorgehen entspricht dem *Virtual Proxy Pattern* der GoF, weswegen Fowler [Fow02] darauf verweist. Das *Virtual Proxy* ist eine Variante des *Proxy* Patterns. Dieses erzeugt erst bei Aufrufen der Methoden die hinter dem *Proxy* liegenden Objekte [GHJV15]. Das *Proxy* selbst hat jedoch dieselbe Schnittstelle wie die Objekte.

Identity Field

Primärschlüssel eines Dateneintrags der Datenbank wird in einem *Identity Field* Objekt gespeichert. Dadurch lassen sich Daten in Objektform und Dateneintrag zusammenführen [Fow02].

Beim *Claim Check* Pattern der *Enterprise Integration Patterns* werden Daten ausgelagert, um die Message-Größe zu verkleinern. Ein Identifier in Form des *Claim Checks* wird der Message hinterlegt, damit die Daten später wieder entnommen werden können. Als Identifier könnte ein *Identity Field* verwendet werden. Übergebene Daten werden in die Datenbank eingetragen, der Primärschlüssel wird in das *Identity Field* gespeichert und der Message hinzugefügt.

Foreign Key Mapping

Fremdschlüssel der Datenbank werden auf Objektreferenzen gemappt. Verweist ein Objekt auf ein anderes, hat es in der objektorientierten Welt eine Referenz auf das andere Objekt. Als Fundament dient *Identity Field*. Das *Identity Field* des referenzierten Objektes entspricht dem Fremdschlüssel in der Datenbank [Fow02].

Trotz der Ähnlichkeit zu *Identity Field* gibt es für dieses Pattern keine Verbindung zu *Claim Check*. Entscheidend ist, dass der Fokus dieses Patterns auf die Referenzierung zu anderen Objekten liegt, nicht um das Identifizieren der Objekte in der Datenbank.

Association Table Mapping

Many-to-Many Beziehungen lassen sich über *Foreign Key Mapping* nicht abbilden. Stattdessen wird für dieses Pattern eine extra Tabelle in der Datenbank angelegt, welche die Many-to-Many Beziehungen speichert. Um Beziehungen abzufragen, führt man entsprechende Queries auf diese Tabelle an. Eine Objekt-Version dieser Tabelle im Speicher existiert nicht [Fow02].

Dependent Mapping

Bei Objekte mit einer Kompositionsbeziehung, also eine Ganzes-Teil-Beziehung, mappt *Dependent Mapping* die Objekte. Anstelle eines separaten Mappers für das Objekt, welches in Besitz des anderen steht, wird es direkt mit gemappt. Das Mapping beschreibt hier den Vorgang, die Daten aus der Datenbank in Objekte zu überführen [Fow02].

Embedded Value

Anstelle kleine Objekte, welche Daten innerhalb eines Objektes weiter thematisch bündeln, in eine extra Tabelle zu speichern, werden sie bei *Embedded Value* zusammen in eine Tabelle hinterlegt. Hat die Klasse `Student` ein Feld vom Typen `Person`, welches den Vornamen beinhaltet, haben wir in der Datenbank eine Tabelle für `Student`, welche den Namen aus `Person` integriert [Fow02].

Serialized LOB

Ein großes Objekt wird in einem einzigen Datenbankfeld gespeichert. Als großes Objekt dient hier ein Objekt bestehend aus mehreren Objekten. Serialisierung in dem Datenbankfeld erfolgt bspw. als Binaray Large Object (BLOB) oder bspw. als XML String [Fow02].

Fowler [Fow02] verweist auf das *Memento* Pattern der GoF. Bei *Memento* wird eine Momentaufnahme eines Objektes gemacht. Die Momentaufnahme beinhaltet den kompletten inneren Zustands eines Objektes. Darunter fallen die Werte der einzelnen Felder [GHJV15]. Das *Memento* kann serialisiert gespeichert werden und wird somit zum *Serialized LOB*.

Single Table Inheritance

Die komplette Hierarchie einer Klasse wird in einer einzelnen Tabelle gespeichert. Die Tabelle deckt die Menge aller Felder der Klassenhierarchie ab. Ein extra Feld in der Tabelle hilft, die ursprüngliche Unterklasse zu instanziieren [Fow02].

Class Table Inheritance

Die Datenbank wird so modelliert, dass jede Klasse der Klassenhierarchie eine einzelne Tabelle bekommt. Die Herausforderung hierbei ist, wie die Daten aus den einzelnen Tabellen in das entsprechende Objekt gemappt werden [Fow02].

Concrete Table Inheritance

Jede konkrete Klasse bekommt eine Tabelle in der Datenbank. Als konkret wird jede Klasse bezeichnet, die auch instanziiert wird. Abstrakte Oberklassen fallen nicht darunter. Die Felder der Oberklasse werden innerhalb der Tabellen der konkreten Klassen gespeichert [Fow02].

Inheritance Mappers

Für jede Unterklasse in einer Klassenhierarchie existiert ein spezieller Mapper. Die Mapper selbst unterliegen ebenfalls derselben Hierarchie. Der Vorteil eines speziellen Mappers pro Unterklasse ist der, dass beim Laden der direkte Typ und nicht eine passende Oberklasse zurückgegeben wird [Fow02].

Metadata Mapping

Metadaten für das Mapping werden gespeichert. Darunter fallen der Name der Tabelle und der Name der Klasse, auf welche sie gemappt werden soll, sowie Spaltennamen der Tabelle und Feldnamen der Klasse. Über Code-Generierung oder Reflection kann dann das Mapping von Datenbankdaten auf Objekte vollbracht werden [Fow02].

Für die Erzeugung von Nachrichten im Kontext von Messaging kann *Metadata Mapping* verwendet werden. Die eigentliche Verwendung dieses Patterns findet sich im Bereich von Datenbanken, die damit verbundenen Mechaniken lassen sich jedoch auch für Messaging anwenden. Ein *Messaging Mapper* der Enterprise Integration Patterns generiert eine Nachricht aus einem Objekt auf Basis der Metadaten und umgekehrt. Wird als Nachrichten-Typ bspw. XML gewählt, sind die Namen für Tabelle und Spalten entsprechend XML Tags oder Attribute. Objekt-Felder werden mit diesen Tags und Attributen gemappt.

Query Object

Eine Datenbank-Query wird über ein Objekt abgebildet anstelle direkt bspw. SQL anzuwenden. Die Einzelteile der Query findet sich in Feldern des Objektes. Das *Query Object* wird intern in SQL-Code umgewandelt und angewendet [Fow02].

Repository

Ein *Repository* agiert als weitere Schicht, welche zwischen Domain und Data-Mapping Schichten vermittelt. Clients definieren eine Reihe an Kriterien für Objekte und fragen das *Repository* über ein Interface an. Das *Repository* kümmert sich intern um die Beschaffung des gefragten Objektes [Fow02].

Model View Controller

Das User Interface wird in die drei Rollen Model, View und Controller aufgeteilt. Das Model steht für die Domain Daten und die View stellt die Daten für den User auf dem Display dar. Der Controller dient für die Benutzereingaben, passt Daten ggf. an und aktualisiert sie für die View [Fow02].

Page Controller

Ein *Page Controller* entscheidet zu einer Anfrage, welche View angezeigt und welches Model verwendet wird. Daten können innerhalb einer HTML Anfrage mitgesendet werden. Der *Page Controller* nimmt sich die Daten und erstellt sich ein passendes Model-Objekt. Anschließend wird die View ausgesucht, welche die Daten anzeigt [Fow02].

Front Controller

Alle Anfragen auf einer Website werden von einem *Front Controller* behandelt. Dieser besteht aus einem Web Handler und mehreren Befehlen, welche Anfragen bearbeiten können. Der Web Handler empfängt die HTTP Anfrage, entscheidet welcher konkrete Befehl dafür notwendig ist und führt diesen aus [Fow02].

Template View

Eine statische Website wird, wegen dynamischen Inhalten, um Marker erweitert. Wird die Seite geladen, werden die Marker mit den geladenen Daten ersetzt. Die Website dient somit als Template, also Schablone, welche beim Laden mit weiteren Daten ergänzt wird [Fow02].

Transform View

Ein Transformer ließt die Daten des Models und überführt sie in eine Website. Der Unterschied zu *Template View* ist der, dass dieses Pattern als extra Komponente zwischen Input und Output steht anstelle direkt in der View zu sitzen. Die Transformation kann bspw. über XSLT durchgeführt werden, welches die Domain Daten aus XML in HTML umwandelt [Fow02].

Bei diesem Pattern haben wir Ähnlichkeiten zu dem *Message Translator* Pattern von Hohpe und Woolf. Wir haben in dieser Konstruktion zwei Systeme. Auf der einen Seite haben wir die Website, welche die Daten im HTML-Format erwartet. Auf der anderen Seite steht das Model. Die *Transform View* dient als Translator und überführt die Model Daten in das HTML-Format. Der große Unterschied ist, dass wir uns dabei nicht im Kontext von Messaging bewegen. Es geht hierbei nur um das Überführen in ein Zielformat. Auf Messaging lässt es sich übertragen, wenn es sich bei den beiden Systemen um entfernte Systeme handelt und die Kommunikation über Messaging läuft.

Two Step View

Die Transformation in HTML erfolgt über zwei Phasen. In der ersten Phase werden die Daten in eine logische Präsentation ohne spezielle Formatierung umgewandelt. Der zweite Schritt überführt das Ergebnis der ersten Schicht letztendlich in die HTML Repräsentation. Änderungen am globalen Look-and-Feel der Seiten können durch Änderungen an der zweiten Schicht vorgenommen werden [Fow02].

Application Controller

Ein *Application Controller* steuert den Fluss der Anwendung, indem er entscheidet, welche Domain-Logik ausgeführt werden muss und welche View das Ergebnis der Logik anzeigt. Einzelne Controller des MVC Patterns fragen dann beim *Application Controller* nach der Logik und der View für

deren konkreten Kontext. Eine Möglichkeit ist es, den *Application Controller* als State Model zu verwenden, bei dem die Controller den Zustand der Anwendung über den *Application Controller* ändern [Fow02].

Remote Facade

Die Interaktion von Objekten über ein Netzwerk ist immer teuerer als die lokale Interaktion, da es mehr Faktoren gibt, die man berücksichtigen muss. Um die Interaktion möglichst gering zu halten, sind feingranulare Objekte zu vermeiden. *Remote Facade* bündelt mehrere feingranulare Objekte, sodass sie von außen ein grobes Interface besitzen. Anfragen auf die Fassade liefert zusammengefasste Daten. Sollen Daten gesetzt werden, sorgt *Remote Facade* für die richtige Zuordnung zu den feingranularen Objekten. Die *Remote Facade* besitzt keine Domain-Logik und dient lediglich zum Überführen [Fow02].

Fowler [Fow02] verweist darauf, dass dieses Pattern eine Variaton des *Facade* Patterns der GoF ist. Beim *Facade* Pattern werden Schnittstellen mehrerer Komponenten gebündelt. D.h. es existiert eine Schnittstelle, die Facade, welche zentralen Zugriff auf diese Komponenten ermöglicht. Die Remote Facade hat dieselbe Funktion, nur ist sie speziell für Remote Komponenten, also entfernte Komponenten, welche über das Netzwerk angesprochen werden, gedacht.

Data Transfer Object

Methodenaufrufe über das Netzwerk sind teuer. Deswegen versuchen wir sie möglichst gering zu halten. Ein *Data Transfer Object* speichert alle benötigten Daten in ein einzelnes Objekt, damit nicht Teile über extra Methodenaufrufe nachgefragt werden müssen [Fow02].

Bei diesem Pattern geht es um das Senden von Objekten über das Netzwerk. In den Enterprise Integration Patterns im Kontext von Messaging werden Objekte in Form eines *Document Message* versendet. Der Payload dieser Nachricht entspricht dem Objekt. Um die Anzahl der Nachrichten zu senken, kann hier das *Data Transfer Object* Pattern verwendet werden, indem alle benötigten Daten innerhalb der *Document Message* hinterlegt werden.

Optimistic Offline Lock

Bei nebenläufigen Transaktionen kann es zu Konflikten kommen. Optimistic Offline Lock geht davon aus, dass es eher unwahrscheinlich ist, dass es zu Konflikten kommt. Es stellt sicher, dass Daten während dem Laden nicht von einer anderen Transaktion verändert wurden, da es anschließend sonst zu Konflikten kommen könnte. Diese Überprüfung findet am Ende der Transaktion statt. Der Commit wird nur dann durchgeführt, wenn die Überprüfung positiv ausfällt [Fow02].

Pessimistic Offline Lock

Durch Locks, also Sperren, wird dafür gesorgt, dass nur eine Transaktion gleichzeitig auf Daten Zugriff hat. Da somit nicht mehrere Transaktionen nebenläufig auf dieselben Daten agieren können, werden Konflikte vermieden [Fow02].

Coarse-Grained Lock

Gruppen von Objekten werden komplett gesperrt anstelle einzelne Elemente daraus mit Sperren zu versehen. Der Vorteil ist, dass das Setzen und Freigeben von Sperren vereinfacht wird. Die Herausforderung ist, wie man möglichst natürliche Relationen zu den Objekten aufbaut, um diese zu sperren [Fow02].

Implicit Lock

Sperren müssen konsequent verwendet werden. D.h. Daten mit gesetzten Locks dürfen nicht an anderen Stelle einfach gelesen werden. Auf der anderen Seite müssen die Sperren auch wieder entfernt werden, wenn sie nicht mehr benötigt werden. Die Anwendung bzw. die Anwendung in Form eines Frameworks sorgt in Form des *Implicit Lock* Patterns, dass Sperren implizit gemanagt werden, sodass sich die Entwickler nicht um das Locking kümmern müssen [Fow02].

In einer Cloud Umgebung regelt eine *Data Access Component*, der Cloud Computing Patterns, den Zugriff auf Datenbanken. Einzelne Komponenten kommunizieren darüber anstelle direkt auf den Datenbanken zu operieren. Sie bietet dadurch eine zentral Schnittstelle. *Implicit Lock* findet daher Verwendung in der *Data Access Component*. Anstatt anfragende Komponenten die Sperren explizit setzen und wieder lösen, regelt die *Data Access Component* dies implizit. Aus Sicht der Komponenten passiert dies automatisch. Innerhalb der Data Access Component können verschiedene Locking Mechanismen, wie *Optimistic Offline Lock*, *Pessimistic Offline Lock* oder *Coarse-Grained Lock* im Hintergrund verwendet werden.

Client Session State

Daten einer Session werden auf dem Client gespeichert. Bei Anfragen an den Server werden alle Session Daten mitgesendet. Der Server muss sich nicht um die Speicherung der Client Session Daten kümmern [Fow02].

Server Session State

Der Server speichert sich den Zustand der Session eines Clients. Der Client sendet bei einer Anfrage auf den Server seine Client ID, damit der Server die dazugehörige Session laden kann [Fow02].

Database Session State

Eine Datenbank hält den Zustand einer Session. Bei Anfragen eines Clients an einen Server sendet der Client eine Session ID. Der Server entnimmt die Daten der Session aus der Datenbank mithilfe der übergebenen Session ID [Fow02].

Gateway

Ressourcen müssen nicht objektorientiert sein. Deren API unterscheidet sich somit von dem Verhalten eines normalen Objektes. Ein *Gateway* bietet einen Wrapper um externe Systeme und Ressourcen an, sodass Aufrufe wie die eines gewöhnlichen Objektes erscheinen [Fow02].

Verlinkungen finden sich in den Gang of Four Pattern *Facade* und *Adapter*. Fowler [Fow02] unterscheidet sein *Gateway* Pattern von den oben genannten wie folgt: Eine *Facade* simplifiziert eine API, *Gateway* hingegen passt die API auf die eigenen Bedürfnisse an. Ein *Adapter* hingegen passt zwei unterschiedliche Interfaces so an, dass sie zusammenarbeiten können, während *Gateway* *Adapter* als Teil seiner Implementierung nutzen kann.

Ein Beispiel für die Nutzung dieses Patterns findet sich laut Fowler im Kontext von Messaging der Enterprise Integration Patterns. Bei Messaging haben wir ein Interface, um Messages zu senden. Die Methoden sind generisch, damit alle mögliche Arten von Nachrichten gesendet werden können. Dadurch ist nicht sofort ersichtlich, welche Parameter für eine spezielle Message notwendig sind. Ein *Gateway* stellt eine spezifische Methode für bspw. eine Bestätigungsnachricht bereit, welche genaue Parameter definiert. Intern nutzt sie das generische Interface des Messaging Systems, nach außen hin haben wir jedoch explizite Methoden.

Mapper

Zwei unabhängige Subsysteme können über einen *Mapper* kommunizieren, ohne dass sie von dem *Mapper* Kenntnis nehmen. Dieser sorgt dafür, dass beide Systeme entkoppelt bleiben. *Mapper* können auf verschiedenen Ebenen eingesetzt werden, weswegen ihre Arbeitsweise sich je nach Einsatz unterscheidet. Im Kontext von *Data Mapper* werden Daten aus einer relationalen Datenbank in ein Domain Objekt überführt [Fow02].

Die Herausforderung diese Patterns ist es, den *Mapper* aufzurufen, da die Subsysteme diesen nicht kennen sollen. Eine Möglichkeit sieht sich in dem *Observer* Pattern der GoF, weswegen Fowler [Fow02] auf diesen verweist. Der Mapper agiert als *Observer* für einen der Systeme und wird bei Events getriggert, also aufgerufen, ohne dass dieses System den *Mapper* explizit kennt. Aus seiner Sicht ist er lediglich ein Listener in seiner Liste.

Fowler [Fow02] referenziert die Ähnlichkeit zu dem *Mediator* Pattern der GoF, welches ebenfalls Systeme entkoppelt. Der große Unterschied ist jedoch, dass die Systeme den *Mediator* kennen, was beim *Mapper* nicht der Fall ist. Systeme können den *Mediator* deswegen explizit aufrufen.

Layer Supertype

Für alle Objekte eines Layers führen wir eine Oberklasse ein, welche gleiche bzw. kopierte Funktionalitäten der Objekte anbietet [Fow02].

Separated Interface

Interface und Implementierung sind separiert, bspw. in zwei verschiedenen Packages. Die Abhängigkeit der zwei Teile kann so aufgeteilt werden, da die Implementierung von dem Interface abhängt, andersherum jedoch nicht. Anwendung findet dieses Pattern, wenn die Applikation in mehreren Laufzeitumgebungen läuft, da jede Laufzeit verschiedene Implementierungen benötigen könnte [Fow02].

Registry

Eine *Registry* bietet eine zentrale Anlaufstelle, um benötigte Objekte und Services zu finden. Objekte und Services registrieren sich, damit andere Komponenten diese über die *Registry* abrufen können [Fow02].

Value Object

Simple Objekte, welche sich fast wie primitive Datentypen verhalten, sind *Value Object*s. Als simpel gelten bspw. Objekte wie Money oder Date. Der Vergleich auf Gleichheit passiert nicht auf Referenzebene, sondern hängt von den konkreten Werten der einzelnen Felder ab [Fow02].

Money

Haben wir es mit mehreren Geldwährungseinheiten zu tun, müssen wir umwandeln. Die Problematik beim Umwandeln sind mögliche Rundungsfehler, die aufgrund der verschiedenen Währungen auftreten können. *Money* speichert einen Geldwert und dient als Abstrahierung aller möglichen Währungen [Fow02].

Special Case

Null-Werte können in Programmen Fehler auslösen. Um auf Null-Checks in mehreren Stellen zu vermeiden, werden spezielle Unterklassen einer Klasse geschrieben. Diese hat dasselbe Interface wie eine echte Instanz, hat aber keinerlei Funktion. Methoden dieser Instanz können also ohne Fehler aufgerufen werden. Anstelle null zurückzugeben, wird eine Instanz dieser Klasse zurückgegeben [Fow02].

Plugin

Zur Laufzeit werden konkrete Implementierungen über eine Konfiguration verlinkt. Eine Plugin Factory kümmert sich um die Erzeugung konkreter Instanzen und gibt sie zurück [Fow02].

Service Stub

Anstelle der Verwendung eines echten Services wird zu Testzwecken eine Dummy-Version des Services verwendet. Nach der Entwicklung und Testphase wird der *Service Stub* mit dem tatsächlichen Service erstzt [Fow02].

Record Set

Record Set speichern tabellarische Daten aus relationalen Datenbanken in eine ähnliche Struktur mit Zeilen und Spalten. Die Struktur befindet sich jedoch im Hauptspeicher, sodass die Applikation direkt auf ihr arbeiten kann [Fow02].

3.2.4 Internet of Things Pattern

Reinfurt et al. [RBF+16] beschreiben in ihren Internet of Things Pattern die Kernprinzipien des Internet of Things. Anstelle sich auf spezielle Produkte zu fokusieren, bieten diese Pattern eine abstrakte Sicht ohne auf konkrete Produkte oder Hersteller angewiesen zu sein.

Die Internet of Things Pattern unterscheiden sich von den bisher behandelten Patternsprachen, da sie nicht in einem Buch, sondern über mehrere Paper verstreut publiziert wurden. Im Rahmen dieser Arbeit wurden daher [RBF+16], [RBF+19] und [RBF+17] betrachtet.

In diesem Abschnitt untersuchen wir die Internet of Things Patterns. Die einzelnen Pattern werden auch hier wieder kurz Zusammengefasst. Anschließend finden sich evtl. Anmerkungen zu Relationen zu Patterns aus anderen Sprachen. Im Fokus stehen hier alle bisher behandelten Patternsprachen.

Device Gateway

Ein *Device Gateway* verbindet Geräte an das Backend über IP Kommunikation. Die verbundenen Geräte sind von sich aus nicht in der Lage sich mit dem Backend zu verbinden. Das *Device Gateway* ist ein eigenständiges Gerät, an welchem die einzelnen Geräte verbinden sind [RBF+16].

Reinfurt et al. [RBF+16] verweisen auf das *Adapter* Pattern von Gamma et al. Da es sich bei dem *Device Gateway* um ein echtes Gerät handelt, entspricht es einer physischen Umsetzung bzw. Variante des *Adapter* Patterns.

Das Device Gateway kann defekte Geräte, welche an diesem verbunden sind erkennen und ersetzen. Aus diesem Tatbestand haben wir eine Relation zum Watchdog Pattern von Fehling et al., so Reinfurt et al. [RBF+16].

Device Shadow

Geräte im IoT Kontext können offline und somit nicht verfügbar sein. Eine virtuelle Abbildung des Gerätes findet sich auf Serverseite. Der Server operiert auf bzw. kommuniziert nur über diese Abbildung und hat somit eine lose Kopplung zu dem Gerät. Ist das Gerät wieder online, werden Abbildung und Gerät synchronisiert[RBF+16].

Aufgrund möglicher Synchronisierungsprobleme, welche beim Synchronisieren des States auftreten können verweisen Reinfurt et al. [RBF+16] auf das *Optimistic Offline Lock* Pattern von Fowler.

Bis das Gerät wieder online ist, kann es sein, dass die gesendeten Operationen nicht mehr von Relevanz sind. Abgelaufene Befehle sollen nicht berücksichtigt werden. Dieser Mechanismus findet sich in dem *Message Expiration* Pattern von Hohpe und Woolf, weswegen Reinfurt et al. [RBF+16] darauf verweisen.

Rules Engine

Ein System bekommt verschiedenste Nachrichten. Die Rules Engine entscheidet, was mit der eingetroffenen Nachricht passieren soll. Aktionen und Regeln lassen sich über eine grafische Benutzeroberfläche auf Serverseite einpflegen. Trifft eine Regel ein, wird die damit assoziierte Aktion bzw. Aktionen durchgeführt [RBF+16].

Ein *Content Based Router* kann auf Basis des Inhalts einer Nachricht den Zielort entscheiden. Reinfurt et al. [RBF+16] verweisen deswegen auf die Ähnlichkeit zu ihrer *Rules Engine*. Eine *Rules Engine* kann jedoch zusätzliche Daten für die Entscheidungen hinzunehmen, was beim *Content Based Router* nicht der Fall ist.

Device Wakeup Trigger

Um den Energieverbrauch zu senken können Geräte in einen Sleep-Zustand wechseln. Dadurch ist die Kommunikation zum Server unterbrochen, Nachrichten können nicht mehr gesendet oder empfangen werden. Bei Device Wakeup Trigger existiert eine Low-Power Verbindung, auf die der Server dem registrierten Gerät eine Nachricht senden kann um das Gerät aufzuwecken [RBF+16].

Damit der Server zuordnen kann, welche Antwort zu welchem Wakeup Call gehört, kann *Correlation Identifier* aus EIP verwendet werden, so Reinfurt et al. [RBF+16]. Bei dem Wakeup Call kann es sich entweder um eine *Command Message* oder einer *Event Message* handeln.

Remote Lock and Wipe

Aus Sicherheitsgründen sollen Daten aus einem Gerät gelöscht werden können. Bspw. damit Angreifer die Daten nicht für Zugang zum System missbrauchen können. Das Gerät wird so umgerüstet, dass es Managementbefehle von einem Server empfangen kann, mit welche sich entsprechende Daten löschen lassen [RBF+16].

Ähnlich wie beim Wakeup Call: Der Startschuss für die obige Prozedur kann entweder eine *Command Message* oder eine *Event Message* aus den Pattern von Hohpe und Woolf sein.

Delta Update

Wir haben eine limitierte Netzwerkbandbreite weswegen wir nicht Messages mit kompletten Daten versenden können. Bei Delta Update berechnen wir das Delta der aktuellen sowie der vorherigen Daten. Nur das Delta mitsamt dem Hash-Wert der aktuellen Daten werden versendet. Wie man Delta und Hash berechnet hängt von den konkreten Daten ab [RBF+19].

Remote Device Management

Um mehrere Geräte entfernt zu verwalten wird ein Management-Server im Backend eingerichtet. Management-Befehle werden von diesem Server zu den einzelnen Geräten gesendet, welche diese lokal bearbeiten und eine Antwort zurücksenden [RBF+19].

Visible Light Communication

Binärstrings werden ähnlich wie in Morsezeichen über schnelles an und ausschalten von Licht übertragen. Die Übertragung findet in Lichtgeschwindigkeit und über kurze Distanzen statt. Als Sender dient bspw. eine LED. Der Empfänger besteht bspw. aus einer Fotodiode und einer Einheit zum Dekodieren der Nachricht, oder aber auch aus einer Kamera [RBF+19].

Period Energy-Limited Device

Geräte sind nicht an einer dauerhaften Stromversorgung angeschlossen. Als Energiequelle dient bspw. eine Batterie welche nach einer Zeitperiode ersetzt werden muss. Das Gerät benachrichtigt, wenn die Energiequelle zuneige geht und sie ersetzt werden soll [RBF+17].

Energy-Harvesting Device

Ein Gerät hat nur einen geringen Energiebedarf. *Period Energy-Limited Device* wäre zu viel Aufwand. Das Gerät verfügt über eine Energiegewinnungsvorrichtung wie bspw. Solarzellen, um Energie aus der Umgebung zu gewinnen [RBF+17].

Normally-Sleeping Device

Wir wollen die Energienutzung eines Gerätes minimieren. Hauptkomponenten des Gerätes sind für lange Zeit inaktiv. Sie werden in bestimmten Zeitperioden oder durch bestimmte Events reaktiviert [RBF+17].

4 Implementierung

Im vorherigen Kapitel wurde das Konzept und die Beschaffung der Cross Language Relations erläutert.

Die Navigation durch die Patternsprachen und ihren Relationen verläuft größtenteils über die Literatur. D.h. man benötigt Zugriff auf die ensprechenden Bücher, Artikel, Paper u.Ä. Bei Autorenverweise auf Patterns von anderen Sprachen muss in der verwiesenen Literatur weitergelesen werden. Bei semantisch ähnlichen Pattern, wie sie im vorherigen Kapitel nähergebracht worden, muss das Wissen bereits vorhanden sein. Kennt man die Literatur nicht, so kennt man auch nicht die Relationen zwischen den Patternsprachen. Inhalte einer Patternsprache finden sich meist reduziert auf verschiedenen Webseiten wieder. Relationen von Patterns werden dort als Links zwischen den Webseiten realisiert, um die Navigation zu vereinfachen. Bsp. hierfür sind die Enterprise Integration Patterns (https://www.enterpriseintegrationpatterns.com/) und die Cloud Computing Patterns (http://www.cloudcomputingpatterns.org/). Problem ist hierbei, dass sich keine Cross Language Relations finden. Weder als Hyperlink, noch in Fließtext, da es sich bei den online verfügbaren Daten um reduzierte Inhalte aus der Literatur handeln.

Der zweite Teil dieser Arbeit befasst sich mit der Implementierung einer Patternlandkarte. Die Landkartenanalogie befasst sich mit folgendem Sachverhalt. Bei einer Landkarte haben wir verschiedene Gebiete, welche interessante Standpunkte umfassen. Diese sind über verschiedene Wege miteinander verbunden. Die Wege können auch aus den Ländergrenzen hinaus und in andere Länder führen. Auf das Konzept von Patternsprachen übertragen, ergibt sich: Länder sind einzelne Patternsprachen, die Gebiete, die sie umfassen, sind die Patterns. Die Wege verbinden Gebiete und entsprechen daher den Relationen von Pattern. Pattern können Verlinkungen zu Pattern aus anderen Sprachen haben, diese sind die Wege, welche über Ländergrenzen hinweggehen.

Ein Gebiet ist eine abstrakte Bezeichnung. Darunter können z.B. Bezirke, Städte oder auch Kreise fallen. Das bedeutet Gebieten können Untergebiete zugeordnet werden, was für die Patternsprache verschiedene Gruppierungsvarianten ermöglicht. Über Zooming lassen sich nähere Details einer Gruppe enthüllen. Im Rahmen dieser Arbeit findet sich jedoch nur einen einfachen Zoom. D.h. wir haben Patternsprachen auf oberer Ebene und direkt darunterliegend die einzelnen Patterns. Über die Landkartenanalogie könnte jedoch komplexeres Zooming erklärt werden. Sprich, eine Patternsprache umfasst mehrere Gruppen von Patterns, welche ihrerseits wieder in Gruppen aufgeteilt sind usw.

Die Patterngraphen sind über die Cross Language Relations aus dem vorherigen Kapitel miteinander verbunden. Die Anwendung, welche im Rahmen dieser Arbeit geschrieben wurde, visualisiert Patterns und ihre Relationen und ermöglicht so die Navigation durch die Sprachen.

Der Abschnitt 4.1 befasst sich mit den Daten der Anwendung. Die eigentliche Anwendung wird in Abschnitt 4.2 erläutert.

4.1 Daten

Die Anwendung visualisiert Patternsprachen, ihre Patterns und die entsprechenden Relationen. Der wichtigste Teil sind hierbei die Daten, welche als Treibstoff für die Anwendung fungieren. Die Daten umfassen all die oben genannten Punkte. Die Anwendung bekommt die Daten und kann die Inhalte darstellen.

Wie in Kapitel 2 bereits erläutert, dient *PatternPedia* als Ausgangspunkt für diese Anwendung. Die Problematik bei dieser Umsetzung von PatternPedia ist, dass sich die Daten alle auf einer zentralen Datenbank befinden. Wird die Datenbank ausgeschalten, sind alle dokumentierten Patterns und Relationen nicht mehr verfügbar. Um dem entgegenzuwirken soll PatternPedia komplett umstrukturiert werden. Ziel ist die zentrale Datenverwaltung durch die Ideen des Semantic Webs zu ersetzen. Grundlagen des Semantic Webs finden sich ebenfalls in Kapitel 2.

Über das Semantic Web liegen die Daten verteilt an beliebigen Standorten. Ressourcen werden über Universal Resource Identifier (URI) eindeutig identifiziert. D.h. selbst wenn die Daten verstreut sind, können sie sich auf dasselbe Objekt beziehen, wenn sie dies auf die gleiche URI machen. Dadurch wird die dezentrale Speicherung von Daten ermöglicht.

Die folgenden Abschnitte gehen auf genauere Problemstellungen der Daten ein. In Abschnitt 4.1.1 befassen wir uns mit der konkreten Umsetzung der Semantic Web Ideen. Der Abschnitt 4.1.2 beschreibt die Modellierung der Daten. Wie die Cross Language Relations in den Daten realisiert werden, findet sich in Abschnitt 4.1.3.

4.1.1 Semantic Web

Die Kerntechnologien des Semantic Webs nach Berners-Lee et al. [BHL01] sind XML, RDF und Ontologien. RDF sind Fakten der Form: Subjekt, Prädikat und Objekt. Die Fakten werden standardmäßig über XML definiert. Im Rahmen dieser Anwendung wurden die Daten über das Turtle-Format erfasst. Beim Turtle-Format handelt es sich um eine konkrete Syntax für RDF. Anstelle von XML wird entsprechend die Turtle-Syntax für die Daten verwendet.

RDF dient, neben dem Semantic Web Aspekt, gut zur formalen Definition einer Patternsprache nach Falkenthal et al. [FBL18]. Eine Patternsprache lässt sich als Graph mit Patterns und Relationen ausdrücken. RDF-Daten lassen sich ebenfalls als Graphen interpretieren. Die Ressourcen und Werte entsprechen den Knoten, die Prädikate oder Eigenschaften stellen die Verbindungen zwischen den Knoten dar. Ein erster Modellierungsansatz könnte daher wie in Abbildung 4.1 oben lauten. Zwei Patterns werden über das Prädikat *hasRelationTo* miteinander verbunden. Nach Falkenthal et al. [FBL18] können Relationen jedoch zusätzliche Daten wie Gewichte oder Typen beinhalten. An einem Prädikat lassen sich keine weiteren Informationen hinzufügen. Aus diesem Grund sind die Daten nach Abbildung 4.1 unten modelliert. Relationen sind eigene Entitäten. Die Verlinkung der beiden Pattern läuft nun indirekt über die Relation. Die Relation verweist auf die Pattern als *Source* und *Target*. Die Pfeile, welche von der Relationsentität nach unten zeigen, symbolisieren zusätzliche Eigenschaften, welche eine Relation haben kann.

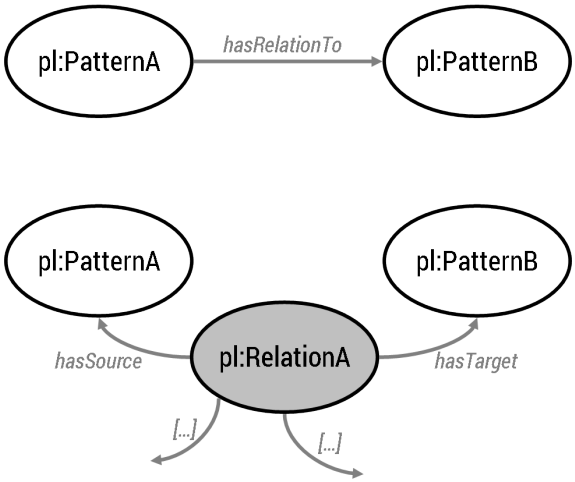

Abbildung 4.1: Modellierung von Relationen über Prädikate (oben) und über separate Entitäten (unten)

4.1.2 Modellierung

Nach dem Semantic Web Ansatz identifiziert eine URI jede Entität eindeutig. Die Konvention für die Anwendung ist, dass jede URI eine URL darstellt, welche auf die entsprechende Ressource verweist. So kann die Anwendung anhand der URI die notwendigen Daten aus dem World Wide Web laden. Für die Modellierung der Daten bedeutet das, dass jede Entität in eine eigene Turtle-File hinterlegt wird. Unter Entitäten bzw. Ressourcen fallen Patternsprachen, Pattern und Relationen.

URLs können sich bspw. bei Umstrukturieren der Server-Hardware ändern. Versucht man anschließend auf die URL anzufragen, bekommt man eine HTTP 404 Response: die angefragte Ressource findet sich nicht weiter an dieser Location. Um dem entgegenzuwirken, empfiehlt sich die Nutzung von Uniform Resource Names (URNs). Eine Umsetzung dieser Technologie sind Persistent Uniform Resource Locators (PURLs). Bei den PURLs handelt es sich vorerst um normale URLs. Eine Anfrage auf diese URL, führt zu einem PURL Resolver. Dieser sorgt dafür, dass die echte URL entsprechend zurückgegeben wird. Bspw. über einen HTTP Redirect [SWJF96]. Die Nutzung von PURLs zur Identifizierung von Ressourcen im Kontext des Semantic Webs wird vom W3C empfohlen [W3C08].

Die Aufteilung der Entitäten in einzelne Dateien mag theoretisch ein gutes Konzept sein, auf praktischer Seite jedoch ergeben sich dadurch Performanzprobleme. Da jede Entität eine separate Datei darstellt, muss diese vorher entsprechend angefragt werden, damit sie in den Speicher geladen werden kann. Die Dateien belaufen sich alle auf wenige Kilobyte, doch die Anzahl der Anfragen führt zu mangelhafter Performanz.

Für jede Datei wird ein Request an die PURL gesendet. Die Anfrage auf die PURL führt zu einem Redirect auf GitHub. GitHub wird im Rahmen dieser Arbeit für das Hosting der Turtle Files verwendet. Dieser Ablauf wird in Abbildung 4.2 verdeutlicht.

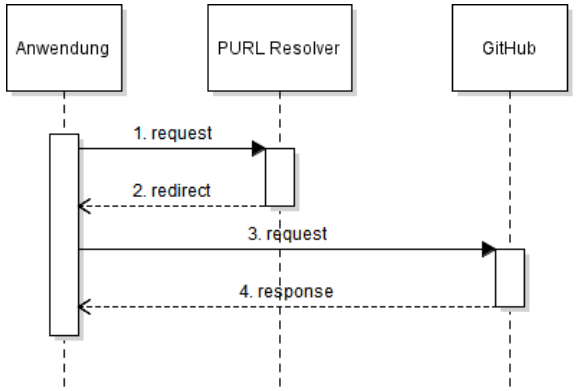

Abbildung 4.2: Eine Anfrage auf die PURL führt zu einem *Redirect* zu GitHub, wo sich die tatsächliche Datei findet

Pro Datei ergeben sich somit zwei Anfrageschritte. Bei den Cloud Computing Patterns haben wir insgesamt 429 Dateien, welche zu 858 Anfrageschritten führen. Cross Language Relations zählen nicht dazu. Bis alle notwendigen Daten in der Anwendung geladen wurden, dauert es zu lange. Das Semantic Web versagt an dieser Stelle. Um dem entgegen zu wirken, relaxieren wir das Konzept: Entitäten werden zusammengefasst. Pro Patternsprache existieren dann folgende drei Dateien:

- Base File – Enthält die Informationen der Patternsprache selbst, darunter fallen Aufbau der Sprache sowie welche Patterns und Relationen sie enthält.
- Patterns File – Definiert alle Patterns der Sprache
- Relations File – Definiert alle Relationen der Patterns innerhalb der Sprache

Es findet sich pro Entität nur noch eine konkrete URL. Diese führt auf die Base File.

Wie Patterns einer Sprache genau aufgebaut sind, wird in der Base File definiert. Unter dem Aufbau eines Patterns verstehen wir die *Sections*, welche auf verschiedene Fragestellungen eines Patterns eingehen. Autoren von Patternsprachen können sich diese frei auswählen. Für die Anwendung muss jedoch der Name des Patterns gegeben sein. Neben dem Aufbau finden sich innerhalb des Base Files sowohl eine Liste an enthaltenen Patterns sowie eine Menge von Relationen. Damit bleiben wir der formalen Definition von Falkenthal et al. [FBL18] treu.

4.1.3 Modellierung der Cross Language Relations

Nach der formalen Definition einer Patternsprache von Falkenthal et al. [FBL18] umfassen Relationen vorerst nur Patterns der eigenen Sprache. Das bedeutet es ist nicht erlaubt Relationen zu erstellen, welche Patterns aus anderen Sprachen enthalten. Neben der Verletzung des Formalismus gibt es einen weiteren Grund, weswegen dies keine gute Lösung ist. Hat Patternsprache *A* eine solche Relation zu einem Pattern aus Patternsprache *B*, hat Patternsprache *B* keinerlei Kenntnisse darüber. Interessiert uns nun, welche Sprachen auf *B* zeigen, müssten wir alle anderen Sprachen nach solchen Relationen durchsuchen. Um sie durchsuchen zu können, müssen sie vorher jedoch mitsamt all ihren Relationen geladen werden. Dies ist nicht praktikabel.

Für Cross Laguage Relations führen Falkenthal et al. [FBL18] den Aggregations-Operator ein, welcher zwei gegebene Patterngraphen unter Zunahme einer Kantenmenge zu einem neuen Patterngraphen kombiniert. Die Anwendung ermöglicht dies über Views. Eine View enthält Verweise auf zwei Patternsprachen und hat eine Liste von Relationen. Dies entspricht dem Resultat des Aggregators. Die Relationen können dann Patterns aus den beiden angegebenen Patternsprachen enthalten und bilden somit die CLRs ab. Eine View wird durch zwei Dateien realisiert: *(1)* einer Base File sowie *(2)* einer File mit allen CLRs. Die Base File hat einen ähnlichen Aufbau wie die Base File der einzelnen Patternsprachen. Außnahme ist, dass sich dort keine Verweise auf Patterns, sondern auf Patternsprachen befinden. Damit werden indirekt alle Patterns der relevanten Sprachen referenziert. Weiter findet sich in der Base File die Liste an Relationen. Die Relationen selbst werden in der zweiten File ausformuliert. Die dort enthaltenen Relationen umfassen Patterns aus beiden Sprachen. Für jedes Zweierpärchen aus der Menge der Patternsprachen existiert eine View, wenn sich Relationen zwischen den Sprachen finden.

Patternsprachen, die in einer View vorkommen, müssen dies in ihrer Base File definieren. Ohne diesen Vermerk hätten wir erneut die eingangs erwähnte Verlinkungsproblematik.

4.2 Anwendung

Ziel der Anwendung ist es, Navigation über die Patternsprachen, Patterns und Relationen zu ermöglichen. Dafür nimmt sie die gegebenen Daten und visualisiert sie auf geeignete Weise.

Aufgrund der frei definierbaren Struktur von Patterns und Patternsprachen, kann die Darstellung auf verschiedenste Weisen umgesetzt werden. Aus diesem Grund ist die Anwendung so konzipiert, dass sie verschiedene Darstellungsarten unterstützt. Ein *Renderer* ist eine Komponente innerhalb der Anwendung, welche gegebene Daten visualisiert. Je nach Sprache können verschiedene solcher Komponenten erstellt werden, um so verschiedenste Darstellungen umzusetzen.

Über die Darstellung hat der Benutzer der Anwendung einen Überblick über die Patternsprache. Die Renderer erlauben zusätzlich Interaktion, um so die Navigation durch die Sprachen zu ermöglichen. Als Navigation verstehen wir bspw. das Anzeigen der Details von einem gewünschten Pattern, oder auch das Filtern von unerwünschten Details.

Den genauen Aufbau und die Details der Anwendung werden in den folgenden Abschnitten näher gebracht. Abschnitt 4.2.1 geht auf den Aufbau der Anwendung sowie verwendete Frameworks und Libraries ein. Das Konzept der Renderer findet sich in Abschnitt 4.2.2, die Architektur dieser Komponenten in Abschnitt 4.2.3. Abschnitt 4.2.4 behandelt Netzwerkgraphen als allgemeine Darstellungsart. Filter erlauben das Ausblenden von Patterndaten und werden in Abschnitt 4.2.5 erläutert. Abschnitt 4.2.6 behandelt weitere Visualisierungen, welche vielversprechend waren, die es jedoch nicht in die Anwendung geschafft haben.

4.2.1 Aufbau

Abbildung 4.3 zeigt den abstrakten Aufbau von PatternPedia in einer früheren und in der jetzigen Variante. Bei der vorherige Version von PatternPedia, links im Bild, handelt sich um eine Client-Server-Architektur. Der Server hat Zugang zu einer zentralen Datenbank, auf welcher alle

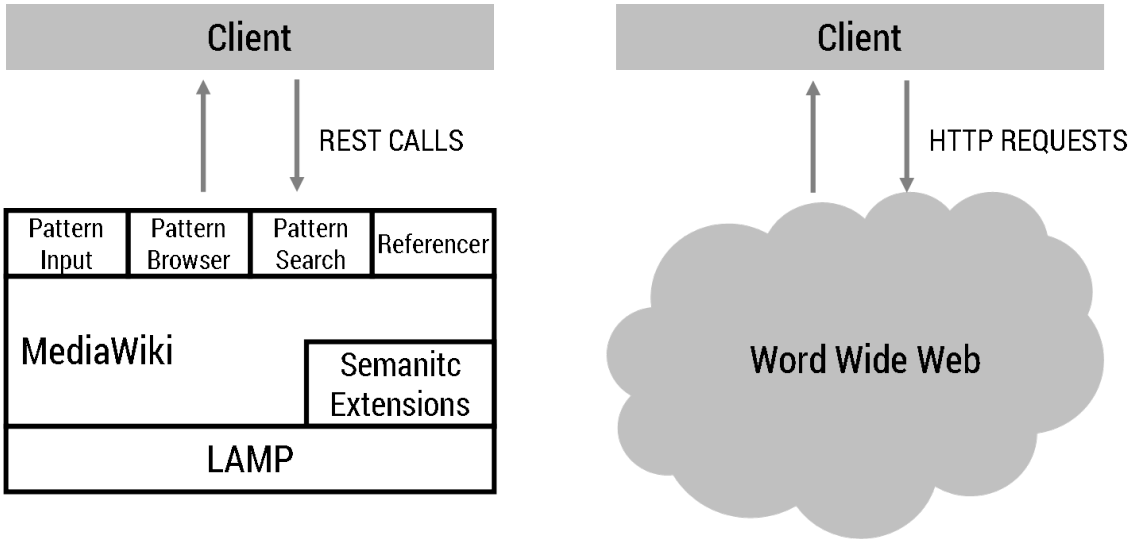

Abbildung 4.3: Alte Architektur von *PatternPedia* (links) im Vergleich zur neuen (rechts)

notwendigen Daten gespeichert sind. Clients konnten über REST-Calls die Daten vom Server anfragen. Die neue Vesion von PatternPedia, welche die Anwendung dieser Arbeit darstellt und sich rechts im Bild befindet, soll von dieser Architektur weg. Daten liegen dezentralisiert im World Wide Web verstreut. Ein Client lädt sich benötigte Dateien über HTTP-Anfragen.

Die Anwendung besteht demnach aus den Daten, welche im obigen Abschnitt 4.1 bereits erläutert wurden sowie einem Client. Bei dem Client handelt es sich um eine Angluar Anwendung, die mit Typescript implementiert ist und auf einem Internet Browser läuft. Der Client hat Zugang zur PatternPedia Base File, diese dient als Einstiegspunkt für weitere Ladevorgänge. Diese Datei, bei dem es sich ebenfalls um eine Turtle File handelt, definiert eine Liste an vorhandenen Patternsprachen. Nach Konvention sind die URIs der Sprachen URLs, welche auf den Standort der jeweiligen Datei führen. Benötigt eine Komponente der Anwendung Zugang auf eine Sprache, kann er diese über eben diese URL anfragen und in die Anwendung laden.

Bei den Daten handelt es sich um RDF Tripel. Damit die Anwendung diese laden und verwenden kann, nutzt sie die `rdfstore-js` Library (`https://github.com/antoniogarrote/rdfstore-js`). Diese ermöglicht die Nutzung eines lokalen RDF Stores, in welcher die Daten geladen werden können. Weiter erlaubt es Anfragen auf die Daten über SPARQL. SPARQL ist eine Querylanguage für Triplestores. Darüber lassen sich Anfragen auf die geladenen Daten stellen.

4.2.2 Renderer

Der Aufbau von Patternsprachen und Patterns ist grundsätzlich dem Patternautor überlassen. D.h. es gibt keine strikte Vorgabe, wie diese auszusehen haben. Folglich kann die Darstellung der Daten auf verschiedenste Weisen umgesetzt werden. Nach *The Nature of Pattern Languages* von Falkenthal et al. [FBL18] lassen sich Patternsprachen immer als Graph mit Patterns und Relationen formulieren. Graphen lassen sich als Netzwerkgraph visualisieren. Aus diesem Grund fiel die Entscheidung auf einen Netzwerkgraphen als universelle Darstellungsart von Patternsprachen. Die Anwendung ermöglicht jedoch mehrere verschiedene Darstellungsarten für eine Sprache.

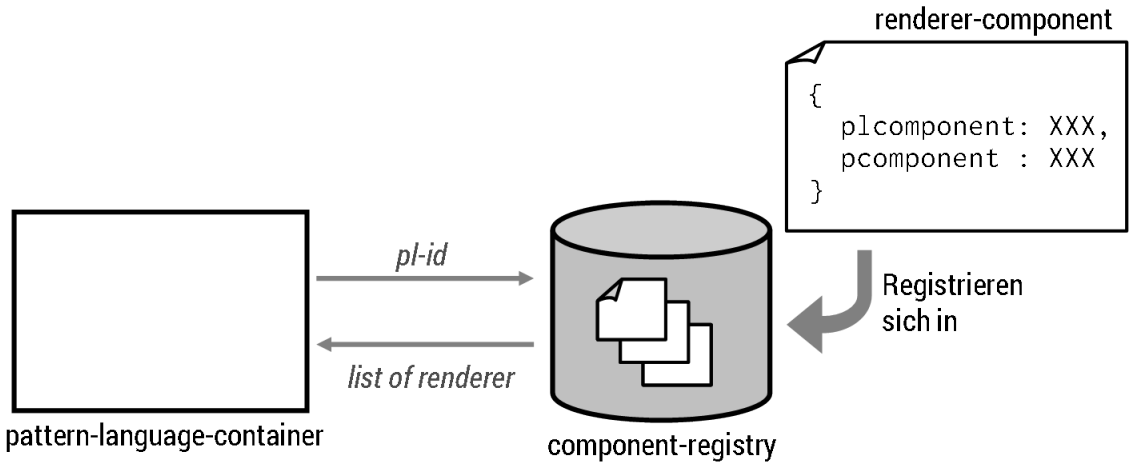

Abbildung 4.4: Architektur des Rendering-Systems

Die Umsetzung des Netzwerkgraphen übernimmt die Data-Driven Documents Library (D3) `https://d3js.org/`. D3 erlaubt die Manipulation des Document Object Models (DOM) durch die Bindung von Daten. So lassen sich, bspw. durch das Hinzufügen von Daten in einem Array, neue DOM Elemente generieren. Relevant für uns ist jedoch nur der Force Directed Graph, da sich Angular im Hintergrund um die View-Components kümmert. Ein Force Directed Graph versucht einen Graphen möglichst schön darzustellen.

4.2.3 Architektur des Rendering-Systems

Die Anwendung ermöglicht mehrere Renderer für eine Patternsprache, da man spezielle Darstellungen haben kann. In den Enterprise Integration Patterns und den Cloud Computing Patterns findet sich jeweils eine Übersicht über alle behandelten Patterns. Diese Übersicht ist speziell auf die Sprache zugeschnitten. Sie auch für andere Sprachen anzuzeigen, hat somit keinen Sinn. Um diese verschiedenen Renderer zu ermöglichen, wird in diesem Abschnitt auf die Architektur des Renderers eingegangen.

Abbildung 4.4 zeigt die Architektur des Rendering-Systems. Renderer sind Komponenten, welche Patternsprachen (`plcomponent`) und Patterns (`pcomponent`) dem Benutzer anzeigen können. Die Komponenten registrieren sich in der Component-Registry unter einer speziellen Patternsprachen-ID. Das Routing innerhalb der Anwendung ist generisch. Navigieren wir auf eine Patternsprache, wechselt die Anwendung zum Pattern-Language-Container und übergibt diesen die ID der Patternsprache. Über die ID fragt der Container die Component-Registry nach allen verfügbaren Renderer ab. Sie dient lediglich als Container und überlässt den Renderer die Darstellung der Inhalte. Findet sich kein registrierter Renderer, gibt es einen Defaul-Renderer als Fallback-Mechanismus. Der Default-Renderer zeigt alle verfügbaren Daten einer Sprache in einer Liste an. Gibt es für Patternsprachen spezielle Darstellungsarten, kann ein solcher Renderer implementiert und in der Component-Registry unter der entsprechenden Patternsprachen-ID registriert werden.

Renderer haben Zugriff auf verschiedene Loader, über welche sie die Daten aus dem lokalen Tripelstore abfragen können. Die Daten werden über SPARQL angefragt, die Antwort der Query wird in eine entsprechende Datenstruktur gemappt und vom Renderer dem Benutzer visualisiert.

4.2.4 Netzwerk Graph

Den Netzwerkgraphen zu den Enterprise Integration Patterns findet sich in Abbildung 4.5. Die Knoten des Graphen entsprechen einzelne Pattern. Der Name des Patterns dient als Beschriftung der Knoten. Die Links zwischen den Knoten entsprechen den Relationen zweier Patterns. Der Pfeil einer Kante zeigt die Richtung der Beziehung. Die Farbe des Knoten entspricht der Gruppe des Patterns.

Hovert man mit dem Mauszeiger über einen Knoten, so bleiben dieser Knoten und all seine Nachbarn sichtbar. Die restlichen Knoten werden fast komplett ausgeblendet. Dies findet sich in Abbildung 4.6 am Beispiel des *Messaging Gateway* Patterns.

Selektieren lässt sich ein Pattern, indem man auf den entsprechenden Knoten klickt. Dadurch wird das Routing in den Gang gesetzt und es wird zu diesem Pattern navigiert. Das bedeutet, es wird in der Component-Registry nachgeschaut, ob es für die aktuelle Sprache einen korrespondierenden Pattern-Component-Renderer gibt. Der Netzwerkgraph dient als Rendering-Komponente, sowohl für die Sprache als auch für einzelne Pattern. Beim Rendering für einzelne Pattern, wird dieses in dem Graph markiert. Dies wird in Abbildung 4.7 illustriert. Hier wurde das *File Transfer* Pattern ausgewählt. Dieses und seine Nachbarknoten werden im Graph rot markiert. Neben der Markierung findet sich eine Infobox auf der linken Seite wieder.

Die Infobox zeigt den Namen des selektierten Patterns, dessen Gruppe, eine Zusammenfassung und die Relationen zu anderen Patterns an. Abbildung 4.8 zeigt die Infobox zum *Content Enricher* Pattern. Die Relationen des Patterns sind nach Sprache gruppiert. An erster Stelle, also unter Enterprise Integration Patterns, finden sich alle Verlinkungen innerhalb derselben Sprache. Unter Cloud Computing Patterns finden sich die CLRs. Die CLRs werden über die definierten Views abgefragt. In dieser Abbildung ist zu erkennen, dass *Content Enricher* eine Relation zum *Compliant Data Replication* Pattern der CCP aufweist. Die Richtung der Pfeile links neben dem Namen sagen aus, ob es sich um ausgehende oder eingehende Kanten handelt. Das Infosymbol neben der Verlinkung deutet an, dass es zu dieser Relation eine Beschreibung gibt.

4.2.5 Filtering System

Die Screenshots im vorherigen Abschnitt haben die Patternsprachen mit all ihren Patterns und Relationen gezeigt. Aufgrund der Menge an Verlinkungen verliert der Graph an Übersichtlichkeit. Diese Arbeit befasst sich mit der Navigation durch Patternsprachen. Folglich ist die Übersichtlichkeit ein wichtiger Teil davon. Ein Filtering System ermöglicht es Patterns, die uns nicht interessieren, und dazugehörige Verlinkungen auszublenden, um so die Übersichtlichkeit zu erhöhen. Wichtig ist hierbei, dass das Filtering nur auf Basis der Patterns arbeitet. Relationen lassen sich nicht direkt mit dem Filter ausblenden. Nur in Zusammenhang mit den Patterns.

Module für Patternsprachen kennen die expliziten Daten ihrer Patternsprache, da sie die Renderer in der Component-Registry registrieren. Sie könnten dem Filtering System eine Liste an relevanten Eigenschaften mitgeben, über welche der Filter konfiguriert werden könnte. Eine Anforderung an den Filter ist jedoch, dass dieser generell sein soll, sprich, für alle Renderer funktionieren soll. Das gilt auch für den Default-Renderer, welcher keinerlei Annahmen über die Felder der Patternsprachen

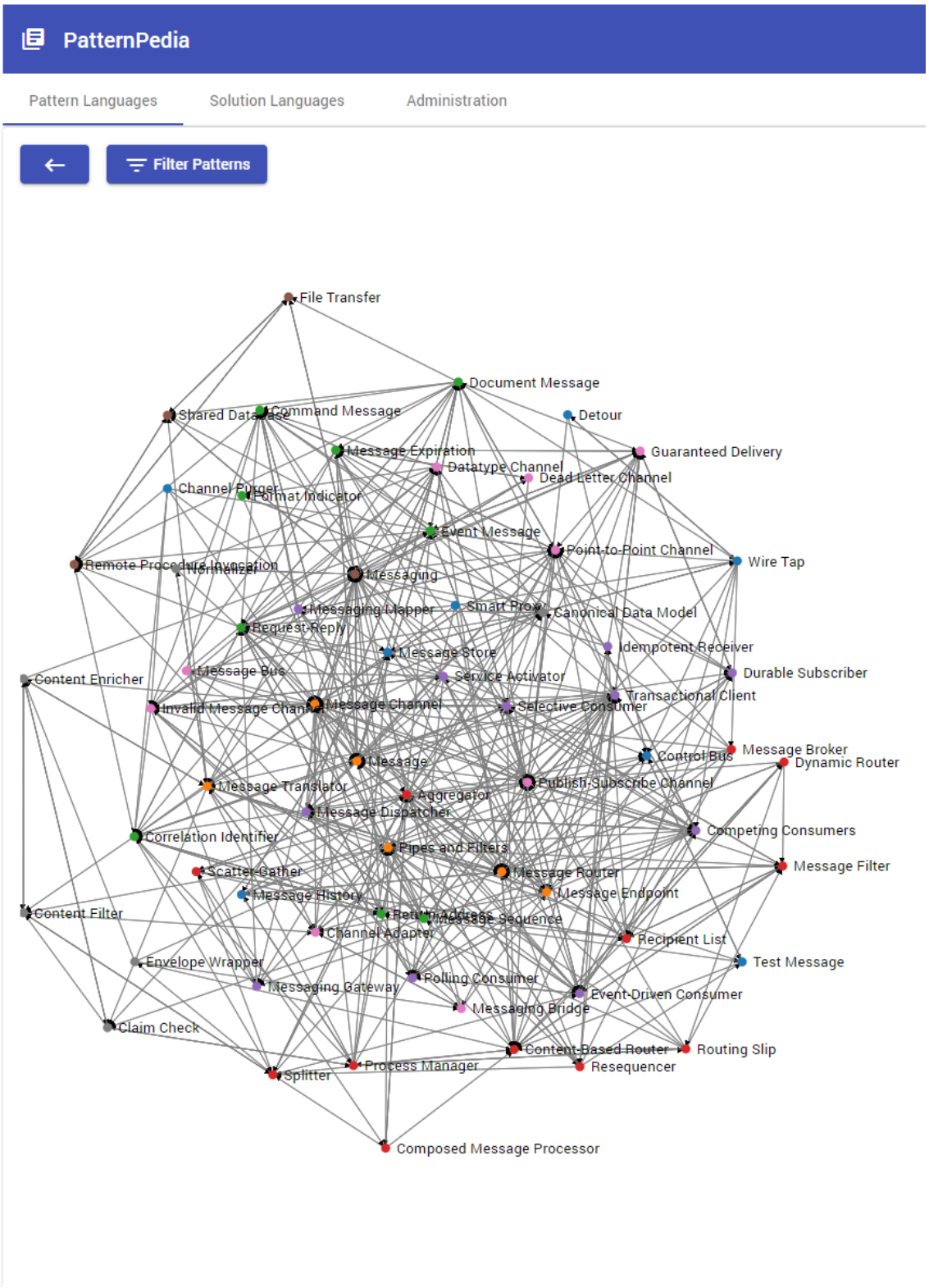

Abbildung 4.5: Netzwerkgraph der Enterprise Integration Patterns

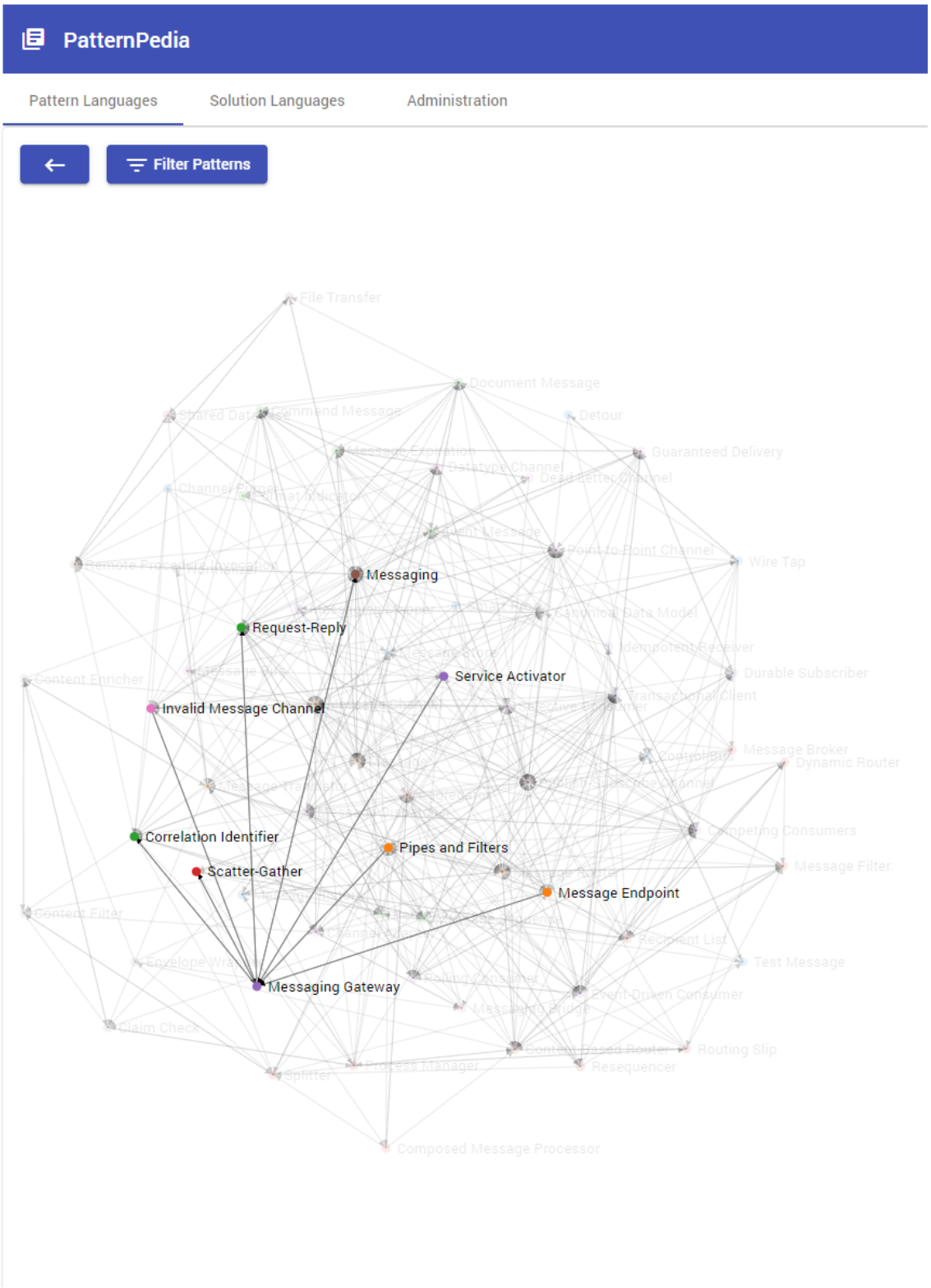

Abbildung 4.6: Netzwerkgraph der EIP bei hovern über einen Knoten

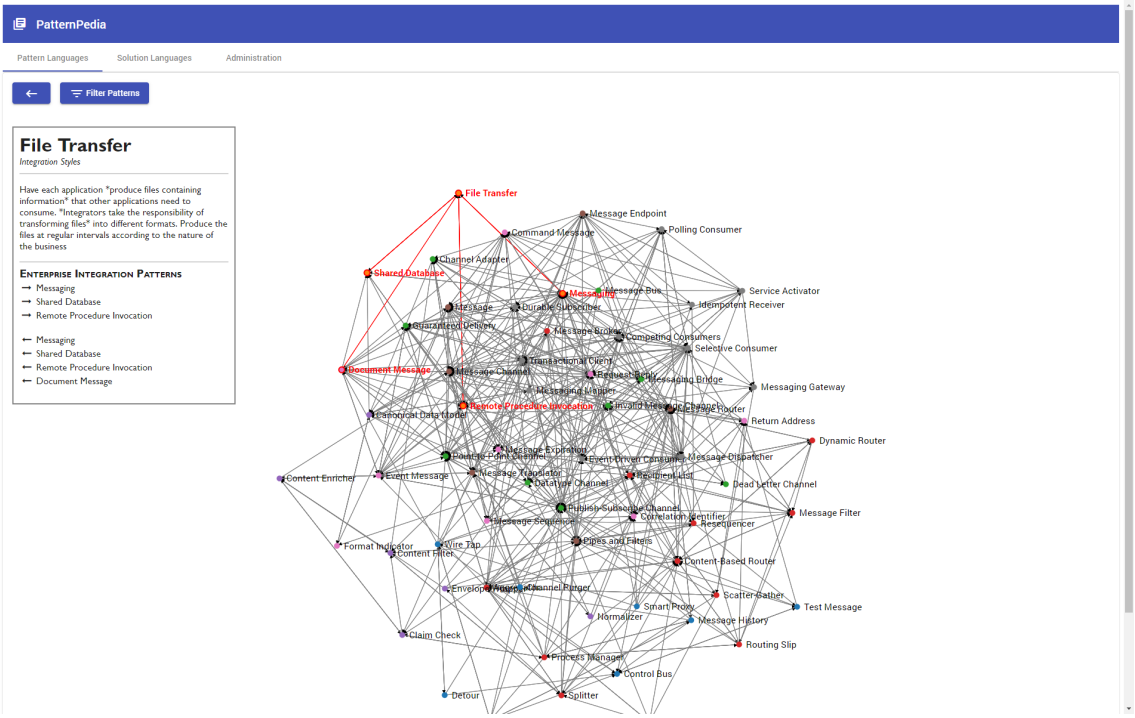

Abbildung 4.7: Netzwerkgraph der EIP bei auswählen eines Knotens

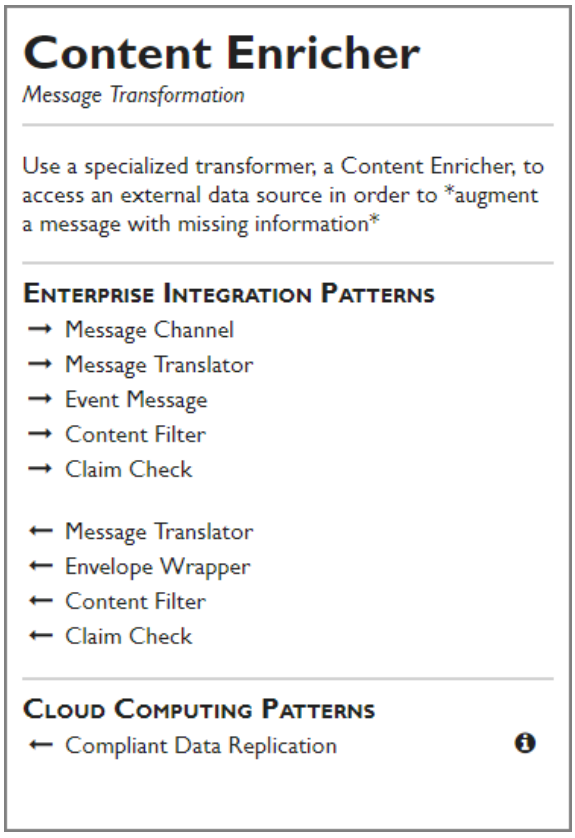

Abbildung 4.8: Infobox bei Selektion eines Patterns

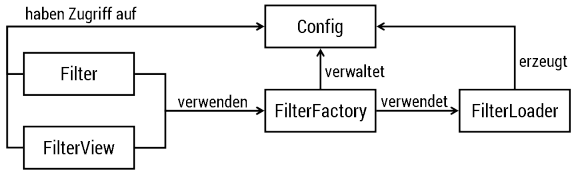

Abbildung 4.9: Architektur des Filtering Systems

machen kann. Das hat zur Folge, dass das Filtering System die filterbaren Eigenschaften anhand der Ontologie der Daten ermitteln muss. Wir führen folgende Konvention ein: als filterbar gelten alle Eigenschaften, welche in der Sprachen-Klasse auf `string` Werte beschränkt sind.

Das Zusammenspiel der Komponenten des Filtering Systems findet sich in Abbildung 4.9. Die einzelnen Komponenten des Systems werden im Folgenden näher erläutert.

Die `Config` enthält alle filterbaren Eigenschaften der Patterndaten sowie die Filterwerte, also jene Werte, die die Ähnlichkeit zu einem Pattern ausmachen. Ein Beispiel: angenommen es findet sich in `Config` ein Feld „name“, welches mit „message“ belegt ist. Der spätere Filtervorgang nimmt sich alle Pattern her und untersucht deren Namen-Eigenschaft auf den Wert „message“. Kommt der Wert in dem Namen vor, bleibt das Pattern erhalten. Wenn nicht, wird es herausgefiltert. Der Netzwekgraph nimmt sich die herausgefilterten Patterns und blendet diese aus dem Graphen aus.

Die `FilterFactory` dient als zentrale Anlaufstelle. Sie verwaltet einzelne Konfigurationsobjekte, die `Config`, für jede Patternsprache. Die Factory verwendet einen `FilterLoader`, um die Felder von `Config` zu ermitteln.

Der `FilterLoader` geht durch die Ontologie der Patternsprache und sucht sich alle Eigenschaften, welche auf `string` beschränkt sind. Gefundene Eigenschaften werden dem `Config` als Felder angehangen.

Der `Filter` bekommt die Patterndaten und filtert sie entsprechend der `Config`. Wichtig ist, dass die Rendering Komponenten den `Filter` nach dem Laden der Daten explizit aufrufen müssen.

Über die `FilterView` kann ein Benutzer die Filterwerte setzen. Sie bekommt die `Config` über die `FilterFactory` und zeigt Inputfelder gemäß dessen Felder an. Benutzer können die Filterwerte in diesen Inputfeldern anpassen. Abbildung 4.10 zeigt den Benutzerdialog, über welchen ein Benutzer die Filterwerte eingeben kann. Neben den filterbaren Eigenschaften auf Patterndatenebene, findet sich eine Checkbox für eine Relationsoption. Ist sie aktiviert, werden alle Pattern gefiltert, welche keine CLRs besitzen. Jedes Pattern könnte potentiell eine Relation zu einem Pattern aus einer anderen Sprache besitzen oder vice versa. Aus diesem Grund kann die Filteroption unabhängig von der konkreten Ontologie angeboten werden. Ein Klick auf *Apply Filter* führt dazu, dass die gesetzten Werte in der `Config` übernommen werden. Bei *Clear Values* werden alle Filterwerte gelöscht. *Cancel* schließt den Dialog ohne Änderungen anzuwenden. Werden die Filterwerte der `Config` hier gesetzt, gelten die Änderungen an allen anderen Komponenten des Filtering Systems, da sie sich die Konfiguration über die `FilterFactory` besorgen.

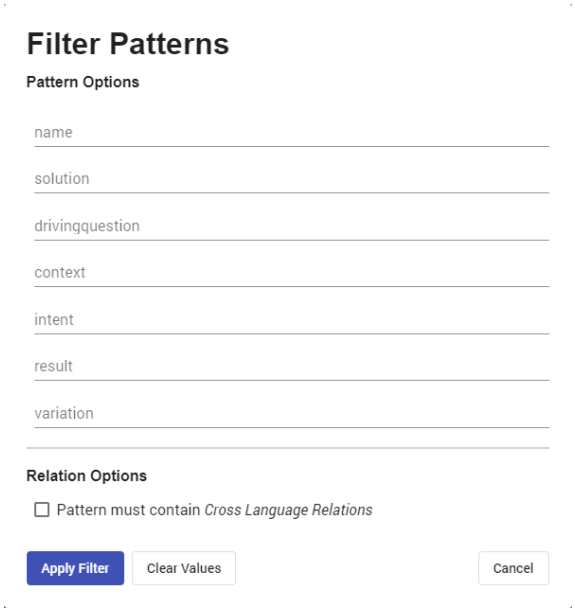

Abbildung 4.10: Benutzeroberfläche des `FilterView`

4.2.6 Weitere Rendering Möglichkeit

Wie oben bereits mehrfach erwähnt, gibt es viele verschiedene Darstellungsmöglichkeiten von Patternsprachen. Aus Zeitgründen konnte nur der Netzwerkgraph realisiert werden. Die Wahl fiel auf die Graphdarstellung, da diese mit allen Patternsprachen kompatibel ist und eine gute Übersicht über die Patternsprache ermöglicht.

In diesem Abschnitt wird noch eine weitere Darstellungsmöglichkeit beleuchtet: die Adjazenzmatrix. Eine Adjazenzmatrix ist eine quadratische Matrix. Die Pattern einer Patternsprache werden den Zeilen und Spalten der Matrix zugewiesen. Haben Zeilen-Pattern und Spalten-Pattern eine Relation zueinander, findet sich in der entsprechende Zelle ein Wert ungleich null. Ein Renderer stellt die Matrix über quadratische Felder dem Benutzer dar. Hat eine Zelle einen Wert ungleich null, wird diese ausgefüllt. Ein Klick auf die ausgefüllte Zelle zeigt nähere Informationen über die Relation. An den Rändern finden sich die Patternnamen. Ein Klick auf die Patternnamen zeigt entsprechend weitere Daten über das Pattern an.

Bei dieser Darstellungsart handelt es sich um eine ähnliche Repräsentation wie beim Netzwerkgraphen. Bei der Implementierung einer Graph-Datenstruktur werden Adjazenzmatritzen teilweise verwendet. Die Begründung, warum man eine Adjazenzmatrix als Visualisierung der Daten verwendet, hat also Ähnlichkeiten mit der des Graphen. Entscheidend ist hier jedoch die Übersichtlichkeit. Bei einem Graph mit 74 Patterns, so wie es bei den Cloud Computing Patterns der Fall ist, führt das zu einer Matrix mit 74 Zeilen und 74 Spalten. Diese auf einem Bildschirm zu rendern, könnte unübersichtlicher als der Netzwekgraph sein. Aus diesen Bedenken wurde sich nicht für diese Darstellungsart entschieden. Da es sich dabei jedoch um eine gute Alternative handelt, wurde sie in diesem Abschnitt erwähnt.

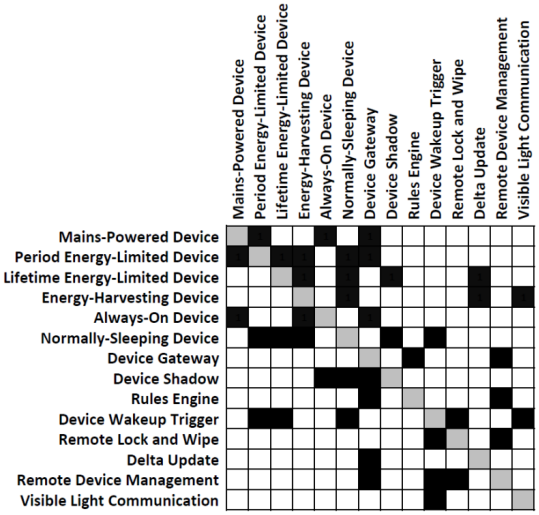

Abbildung 4.11: Die Internet of Things Pattern als Adjazenzmatrix dargestellt [RBF+17]

Abbildung 4.11 zeigt die Darstellung einer Adjazentmatrix für einen Teil der Internet of Things Patterns an. Diese Visualisierung eignet sich sehr gut für eine recht kleine Anzahl an Patterns, da die Matrix so auf einem Blick überschaut werden kann.

5 Zusammenfassung und Ausblick

Diese Arbeit hat sich mit der Identifizierung von Cross Language Relations aus den Patternsprachen *Enterprise Integration*, *Cloud Computing*, *Enterprise Application Architecture* sowie *Internet of Things* beschäftigt. Weiter wurde ein Renderer, also eine Darstellungskomponente, implementiert, welche die Navigation durch Patternsprachen ermöglicht.

Die Anwendung findet sich online auf https://github.com/PatternPedia/pattern-pedia. Die Daten liegen auf einem gesonderten Repository, zu finden auf https://github.com/PatternPedia/patternpediacontent.

Die Cross Language Relations der untersuchten Patternsprachen konnten über Expertenwissen extrahiert werden. Dieses Wissen erlangt man über die Literatur sowie anderen Informationsquellen. Patterns der Sprachen wurden auf Ähnlichkeiten untersucht und die entsprechende Relationen ausformuliert.

Die Daten wurden im Rahmen des Semantic Webs über Turtle Files als Triples gespeichert und auf GitHub hochgeladen. Als Konvention gilt, die URIs sind URLs, welche den Standort der Datei beschreibt. Diese Konvention führt zu lange Wartezeiten beim Laden der Daten, da jedes Pattern und jede Relation einer Sprache separate Dateien darstellen. Um dem entgegenzuwirken, wurden die Daten umstrukturiert. Patterns und Relationen finden sich jeweils in einer Datei gesammelt.

Die vorherige Variante von PatternPedia hatte eine Server-Client Architektur unter Verwendung einer zentralen Datenbank. Durch die Umstellung auf die Praktiken des Semantic Webs findet sich nur noch ein Client, welcher sich die Daten aus dem World Wide Web lädt.

Die Daten werden in der Anwendung als Netzwerkgraph dargestellt. Diese Darstellungsart fußt auf der formalen Definition einer Patternsprache nach Falkenthal et al. [FBL18]. Der Graph ermöglicht einen Überblick über die Patterns und ihren Relationen zueinander. Ein Klick auf ein Pattern zeigt dessen inhaltliche Zusammenfassung zusammen mit einer Liste von relevanten Verlinkungen zu anderen Patterns. Die aufgelisteten Relationen sind nach Patternsprachen gruppiert, sodass auch Cross Language Relations angezeigt werden.

Die Anwendung ist so aufgebaut, dass mehrere Renderer umgesetzt werden können. Dadurch lassen sich auch Renderer für spezielle Patternsprachen umsetzen, sofern die Sprachen eine geignete Darstellung ermöglichen.

Die Visualisierung der Daten der Patternsprache ist aufgund der stark verlinkten Pattern recht unübersichtlich. Ein Filter kann unerwünschte Patterndaten daher ausblenden. Er ist dabei so aufgebaut, dass er für alle möglichen Daten funktionieren kann. Filterbare Eigenschaften sucht sich der Filter über die Ontologien der Daten zusammen. Über eine UI lassen sich Filterwerte für diese Eigenschaften setzen, um die Daten zu filtern.

Ausblick

Im Rahmen dieser Arbeit wurden vier Patternsprachen auf Cross Language Relations untersucht. Mit den Pattern-oriented Software Architecture Pattern von Buschmann et al. [BMR+96] finden sich weitere Pattern, die von den Autoren der untersuchten Sprachen verlinkt wurden. Weitere vielversprechende Patternsprachen wären bspw. noch die Core J2EE Patterns von Malks et al. [MAC01], welche Patterns im Umfeld von Java 2 Platform Enterprise Edition (J2EE) bereitstellen. Die Green Business Process Patterns von Nowak et al. [NLS+11] beleuchten Patterns für die Optimierung von Business Prozessen mit Hinblick auf ökologischen Aspekten und wäre ein weiterer Kandidat.

Das Einpflegen der Daten geschieht über Turtle-Files. Für Informatiker und Softwareentwickler stellt dies kein Hindernis dar. Für Personen ohne diesen Hintergrund kann es jedoch schwierig sein, Daten anzulegen. Patternsprachen sind nicht auf das Informatikgebiet beschränkt, zumal Patterns nach Alexander et al. [Ale78] ursprünglich aus dem Architekturbereich stammen. Eine Erweiterung der Anwendung könnte das Einpflegen der Daten umfassen. Die frühere Version von PatternPedia hatte dieses Feature im Rahmen der MediaWiki bereits implementiert.

Die Hauptdarstellungsart dieser Arbeit ist ein Netzwerkgraph. Neben diesem finden sich Default-Renderer, welche die Daten als Liste ohne spezielles Format aufzeigen. In Kapitel 4 finden sich weitere Darstellungsideen, welche es jedoch nicht in die Anwendung geschafft haben. Interessant wäre es daher, herauszufinden, welche weitere mögliche Darstellungsarten für Patternsprachen Sinn ergeben.

Die Daten mussten während der Bearbeitung dieser Arbeit mehrmals umstrukturiert werden. Grund sind die Performanzprobleme, auf die im obigen Abschnitt eingegangen wurde. Daher stellt sich die Frage, ob das Semantic Web aus praktischer Sicht eine gute Entscheidung darstellt. Diese Frage erfordert weitere Nachforschungen, welche in Zukunft durchgeführt werden könnten.

Das in der Anwendung umgesetzte Filtering ist nur auf Patternbasis. Nach der formalen Definition können Relationen verschiedene Gewichte und Typen annehmen. Ein weiterer Filter könnte sich dieser Daten annehmen und auf Relationsbasis filtern. Weiter könnten auch Patternsprachen als Ganzes gefiltert werden. Die Filter-Komponente bietet daher ebenfalls Potenzial, um weiter ausgebaut zu werden. Neben dem Filtern der Daten würde sich auch eine Suche nach Daten anbieten. Beide Aspekte können die Navigation durch die Patternsprachen bereichern.

Literaturverzeichnis

[Ale78] C. Alexander. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1. Aug. 1978. ISBN: 0195019199 (zitiert auf S. 9, 11, 84).

[BHL01] T. Berners-Lee, J. Hendler, O. Lassila. „The Semantic Web“. In: *Scientific American* 284.5 (Mai 2001), S. 34–43. URL: http://www.sciam.com/article.cfm?articleID=00048144-10D2-1C70-84A9809EC588EF21 (zitiert auf S. 15, 16, 70).

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. *Pattern-Oriented Software Architecture*. John Wiley & Sons Inc, 12. Juli 1996. 476 S. ISBN: 0471958697 (zitiert auf S. 23, 26, 41, 42, 44, 47, 48, 84).

[FBFL15] C. Fehling, J. Barzen, M. Falkenthal, F. Leymann. „PatternPedia –Collabroative Pattern Identificationand Authoring“. In: *Proceedings of the International Conference on Pursuitof Pattern Languages for Societal Change (PURPLSOC)*. epubli GmbH, Juni 2015 (zitiert auf S. 13–15).

[FBL18] M. Falkenthal, U. Breitenbücher, F. Leymann. „The Nature of Pattern Languages“. In: *PURPLSOC* (2018) (zitiert auf S. 9, 11–13, 19, 70, 72–74, 83).

[FLR+14] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Cloud Computing Patterns*. Springer-Verlag KG, 1. Feb. 2014. ISBN: 3709115671 (zitiert auf S. 11, 19, 33–53, 56).

[Fow02] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison Wesley, 1. Nov. 2002. 533 S. ISBN: 0321127420 (zitiert auf S. 11, 17, 19, 32, 39, 43, 44, 47, 53–65).

[GHJV15] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns*. MITP Verlags GmbH, 26. Jan. 2015. 480 S. ISBN: 3826697006 (zitiert auf S. 24, 28, 30, 31, 44, 47, 48, 56, 58).

[Gre04] B. W. Gregor Hohpe. *Enterprise Integration Patterns*. Addison Wesley, 1. Jan. 2004. 480 S. ISBN: 0321200683 (zitiert auf S. 19–33, 38, 40–42, 45, 47, 48, 50).

[MAC01] D. Malks, D. Alur, J. Crupi. *Core J2EE Patterns: Best Practices and Design Strategies*. Pearson Education, 2001. ISBN: 0130648841 (zitiert auf S. 84).

[NLS+11] A. Nowak, F. Leymann, D. Schleicher, D. Schumm, S. Wagner. „Green business process patterns“. In: *Proceedings of the 18th Conference on Pattern Languages of Programs - PLoP '11*. ACM Press, 2011. DOI: 10.1145/2578903.2579144 (zitiert auf S. 84).

[RBF+16] L. Reinfurt, U. Breitenbücher, M. Falkenthal, F. Leymann, A. Riegg. „Internet of Things Patterns“. In: *Proceedings of the 21st European Conference on Pattern Languages of Programs (EuroPLoP)*. ACM, 2016. DOI: 10.1145/3011784.3011789 (zitiert auf S. 19, 65, 66).

[RBF+17] L. Reinfurt, U. Breitenbücher, M. Falkenthal, F. Leymann, A. Riegg. „Internet of Things Patterns for Devices“. In: *Ninth international Conferences on Pervasive Patterns and Applications (PATTERNS) 2017*. Xpert Publishing Services (XPS), 2017, S. 117–126 (zitiert auf S. 19, 65, 67, 82).

[RBF+19] L. Reinfurt, U. Breitenbücher, M. Falkenthal, F. Leymann, A. Riegg. „Internet of Things Patterns for Communication and Management“. In: *Lecture Notes in Computer Science*. Springer International Publishing, 2019, S. 139–182. DOI: 10.1007/978-3-030-14291-9_5 (zitiert auf S. 19, 65, 67).

[SWJF96] K. Shafer, S. Weibel, E. Jul, J. Fausey. *Introduction to Persistent Uniform Resource Locators*. 1996. URL: http://www.opengis.net/docs/long_intro.html (zitiert auf S. 71).

[W3C04] W3C. *OWL Web Ontology Language Guide*. Hrsg. von M. K. Smith, C. Welty, D. L. McGuinness. 10. Feb. 2004. URL: https://www.w3.org/TR/2004/REC-owl-guide-20040210/ (zitiert auf S. 16).

[W3C08] W3C. *Best Practice Recipes for Publishing RDF Vocabularies*. Hrsg. von D. Berrueta, J. Phipps. 28. Aug. 2008. URL: https://www.w3.org/TR/swbp-vocab-pub/#purls (zitiert auf S. 71).

Alle URLs wurden zuletzt am 23. 08. 2019 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift