Diese Arbeit wurde vorgelegt am
Lehr- und Forschungsgebiet
Informatik 3
Softwarekonstruktion

der Fakultät für Mathematik,
Informatik und
Naturwissenschaften

Masterarbeit

Implementierung von Continuous Delivery mittels BPMN

Implementation of Continuous Delivery through BPMN

von

Niels Willig

Aachen, 7. Mai 2018

Prüfer

Prof. Dr. rer. nat. Horst Lichter

Prof. Dr. rer. nat. Bernhard Rumpe

Betreuer

Dipl.-Inform. Andreas Steffens

Eidesstattliche Versicherung

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Masterarbeit mit dem Titel

Implementierung von Continuous Delivery mittels BPMN

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Aachen, 7. Mai 2018 (Niels Willig)

Belehrung

§ 156 StGB: Falsche Versicherung an Eides Statt
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicher ung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt
(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen.

Aachen, 7. Mai 2018 (Niels Willig)

Danksagung

An dieser Stelle möchte ich allen Personen danken, die mich im Rahmen dieser Arbeit unterstützt haben.

Als erstes möchte ich Prof. Dr. rer. nat. Horst Lichter dafür danken, dass ich diese Arbeit an seinem Lehrstuhl schreiben konnte. Zusätzlich möchte ich mich bei meinem Zweitprüfer Dr. rer. nat. Bernhard Rumpe bedanken.

Ein besonderer Dank gilt meinem Betreuer Dipl.-Inform. Andreas Steffens, der mich während des gesamten Zeitraums immer sowohl fachlich als auch persönlich unterstützt hat.

Darüber hinaus möchte ich mich bei Tamara Ostwald und Marcel Carlé für ihre Unterstützung bedanken.

Als letztes möchte ich mich bei meiner Freundin Melanie Abel bedanken, die mich mit viel Geduld moralisch unterstützt hat.

Niels Willig

Kurzdarstellung

Deutsch

Die Anwendung von Continuous Delivery verspricht zuverlässige und kurze Release-Zyklen. Aus diesem Grund implementieren immer mehr Unternehmen Delivery Prozesse für ihre Software-Produkte. Dabei stoßen sie auf eine Vielzahl von Hürden. Viele dieser Probleme kann das im Rahmen einer Masterarbeit am Lehrstuhl *Softwarekonstruktion* der RWTH Aachen entwickelte Software Delivery System JARVIS überwinden.

In dieser Arbeit werden aktuelle Sprachen und Systeme zur Implementierung von Continuous Delivery untersucht und wichtige Eigenschaften abgeleitet. Neben der theoretischen Eignung von BPMN als Modellierungssprache für Delivery Prozesse wird die Sprache in das Software Delivery System JARVIS integriert. Die Implementierung zeigt das sich BPMN sehr gut zur Modellierung von Delivery Prozessen eignet. Durch die Einbindung eines Modellierungswerkzeugs werden Benutzer bei der Erstellung von Modellen unterstützt. Die Arbeit wird durch die Anwendung von, aus BPM bekannten, Methoden zur Analyse von Prozessen komplettiert.

English

The application of Continuous Delivery promises reliable and short release cycles. Therefore more and more companies are implementing delivery processes for their software products. In doing so, they face a lot of different challenges. As part of a master thesis at the reasearch group *Software Contruction* of RWTH Aachen University, the software delivery system JARVIS was implemented to overcome various of these challenges.

This thesis examines state-of-the-art continuous delivery languages and systems and identifies important characteristics of both. Beside the theoretical qualification of BPMN as a modeling language for delivery processes, the language is integrated into the software delivery system JARVIS. The implementation illustrates the well suitability of using BPMN as a delivery process modeling language. During modeling time, the users are supported by an integrated modeling tool. The thesis is concluded by the application of analysis methods originated from BPM.

Inhaltsverzeichnis

1. Einführung **1**
1.1. Struktur der Arbeit 2

2. Hintergrund **3**
2.1. Business Process Management 3
2.2. Business Process Model and Notation 9
2.3. Camunda BPM 19
2.4. Value Streams 20
2.5. Software Delivery Prozess 20
2.6. Continuous Delivery 22

3. Motivation **33**
3.1. Forschungsfragen 34

4. Modellierung von CD mittels BPMN **35**
4.1. Untersuchung der Eigenschaften von CD Sprachen und Systemen 35
4.2. BPMN 57
4.3. Zusammenfassung 63

5. Implementierung von CD mittels BPMN **65**
5.1. BPMN als Delivery Prozess Modellierungssprache 65
5.2. Konzept zur Integration in JARVIS 68
5.3. BPM Konzepte zur Analyse und Optimierung von Delivery Prozessen . . 84
5.4. Zusammenfassung 85

6. Integration in JARVIS **87**
6.1. Umfang und Technologien 87
6.2. Realisierung des BPMN Modelers 88
6.3. Generieren von Vorlagen 91
6.4. Datenmodellierung 93
6.5. Umgang mit BPMN Modellen 94
6.6. Prozess-Reporting und BAM 97

7. Diskussion **99**
7.1. Implementierung von CD mittels BPMN 99
7.2. BPM Konzepte zur Analyse und Optimierung von Delivery Prozessen . . 103
7.3. Selbstorganisierung 103
7.4. Integriertes Modellierungswerkzeug 104

8. Zusammenfassung und Ausblick **105**
8.1. Zusammenfassung 105
8.2. Ausblick 106

A. Anhang **109**
A.1. Beispiel einer Skript-Aufgabe in BPMN 109
A.2. Spezifikation der GIT-Checkout Aktivität 109
A.3. Vorlage der GIT-Checkout Aktivität 111
A.4. Vollständiges Modell 114

Literatur **119**

Akronyme **127**

Tabellenverzeichnis

4.1. Ergebnis der Analyse für Jenkins 44
4.2. Ergebnis der Analyse für GoCD 47
4.3. Ergebnis der Analyse für Codeship Pro 49
4.4. Ergebnis der Analyse für Concourse 51
4.5. Ergebnis der Analyse für Wercker 53
4.6. Zusammenfassung der Analyse-Ergebnisse 56

Abbildungsverzeichnis

2.1. PDCA-Regelkreis 5
2.2. BPMN: Token-Konzept 9
2.3. BPMN: Basiselemente 10
2.4. BPMN: Aufgaben 11
2.5. BPMN: Markierungen an einer Aufgabe 12
2.6. BPMN: Ereignis-Typen 12
2.7. BPMN: Konkrete Ereignis-Typen 14
2.8. BPMN: Gateways 15
2.9. BPMN: Flüsse und Assoziationen 16
2.10. BPMN: Daten und Datenablagen 17
2.11. BPMN: Kollaborationen 18
2.12. BPMN: Konversation 18
2.13. BPMN: Choreographie 19
2.14. Camunda BPM Architektur 20
2.15. Statische Relationen u.a. zwischen SDS, Prozess und Modell 22
2.16. Allgemeiner Delivery Prozess 24

4.1. Venn-Diagramm: Verteilung der Eigenschaften 36
4.2. Delivery Prozess Domäne 37
4.3. Jenkins-Elemente der Domäne 45
4.4. GoCD-Elemente der Domäne 48
4.5. Codeship-Elemente der Domäne 50
4.6. Concourse-Elemente der Domäne 52
4.7. Wercker-Elemente der Domäne 54
4.8. BPMN Domäne (vereinfacht) 57
4.9. Angeheftete Zwischenereignisse 60
4.10. Artefakt-Transfer zwischen zwei Aktivitäten 61
4.11. Artefakt-Storage in BPMN 62

5.1. BPMN Delivery Prozess Domäne 66
5.2. Unterschiedliche Situationen von logischen Abhängigkeiten 66
5.3. Architektur zur Erweiterung von JARVIS 69
5.4. Architektur des BPMN Modelers aus der Server-Perspektive 70
5.5. Architektur des BPMN Modelers aus der Client-Perspektive 71
5.6. Architektur des BPMN Model Services 72
5.7. Architektur des Pipeline Planners 73
5.8. Typischer Ablauf der Modellierung von BPMN Pipelines (Sequenzdiagramm) 76

5.9. Öffnen eines BPMN Modells (Sequenzdiagramm) 77
5.10. Generierung von Vorlagen (Sequenzdiagramm) 78
5.11. Importieren von Modellen (Sequenzdiagramm) 81
5.12. Vorbereitungen zur Ausführung (Sequenzdiagramm) 82
5.13. Umwandlung des BPMN Modells in das interne Datenmodell (Sequenzdiagramm) 83

6.1. Oberfläche des BPMN Modelers 89
6.2. Benutzerunterstützung beim Modellieren 90
6.3. Oberfläche des Token Simulators 90
6.4. Properties Panel mit ausgewählter Vorlage 91
6.5. Umwandlung einer Spezifikationen in eine Vorlage 92
6.6. Ansicht der externen Modelle 94
6.7. Importieren von BPMN Modellen 95
6.8. KPIs-Tab eines Modells 97

7.1. Beispiel BPMN Modell 100
7.2. Ausführungsplan des Modells 100

Liste der Quelltexte

2.1. Beispiel einer Scripted Pipeline 28
2.2. Beispiel einer Declarative Pipeline 29

4.1. Beispiel eines bedingten Sequenzflusses 59

5.1. Spezifikation der BPMN Command API 74

6.1. Beispiel einer Aktivität im XML-Format 93

7.1. Vergleich der Modellierung von funktionalen Abhängigkeiten 101

A.1. Beispiel einer Skript-Aufgabe 109
A.2. Spezifikation der GIT-Checkout Aktivität 109
A.3. Vorlage der GIT-Checkout Aktivität 111
A.4. Beispiel einer YAML-Pipeline 114

1. Einführung

Inhalt

1.1. Struktur der Arbeit 2

In der Softwareentwicklung existieren eine Reihe von Ansätzen die sich mit der Identifikation und Implementierung von neuen Anforderungen beschäftigen. Der vollständige Prozess wird als Software Development Zyklus bezeichnet und umfasst in der Regel die sechs Phasen: *Anforderungen identifizieren, Design ausarbeiten, Implementieren, Testen, Installieren und Überprüfen.* Für die konkrete Umsetzung der einzelnen Phasen existieren verschiedene Konzepte. Dazu zählt zum Beispiel das Requirements Engineering [Val+13] das die *Erhebung, Analyse und Dokumentation* von Anforderungen umfasst.

Die Umsetzung des Software Development Zyklus erfolgt mittels eines Vorgehensmodells. Dabei werden agile Vorgehensmodelle immer wichtiger, da sie für schnelles Feedback sorgen und somit kürzere Reaktionszeiten ermöglichen. Dies wird unter anderem durch eine iterative Anwendung des Zyklus erreicht. Mit dem steigenden Einsatz von agilen Modellen, wurden Methoden wie *Continuous Integration (CI)* immer wichtiger [Rod+12], um mit den immer kürzer werdenden Auslieferungszyklen zurecht zu kommen.

Um die Auslieferungszeiten weiter zu verkürzen hilft die Betrachtung des Prozesses als *Value Stream.* Das Problem bei der bisherigen Umsetzung von Auslieferungsprozessen in der Softwareentwicklung war, dass zwar für die einzelnen Phasen konkrete Konzepte und Methoden existieren, jedoch das Zusammenspiel aller Teil-Prozesse bisher unzureichend betrachtet wurde. Durch die Betrachtung des Zyklus als *Value Stream* wird eine ganzheitliche Sichtweise auf den Prozess möglich und es entstehen viele Ansatzpunkte um den *Value Stream* effizienter und zuverlässiger zu gestalten [HF10].

Ausgehend von dieser Idee wurde der *Continuous Delivery (CD)* Ansatz entwickelt [HF10, S.3]. Nach einer Studie [Per15] aus dem Jahr 2015 wird CD immer mehr in Unternehmen angewendet.

Zur Implementierung von CD existieren zahlreiche Systeme und Modellierungssprachen. Diese Sprachen ermöglichen das Modellieren des Delivery Prozesses auch Pipeline genannt, der sich mit dem *Bauen, Testen und Veröffentlichen* einer Software befasst. Aktuell hat sich der Ansatz *Pipeline-as-Code (PaC)* etabliert. Dabei wird ein Prozess-Modell in einer einzelnen Datei in einer Versionskontrolle abgelegt. Durch die Betrachtung eines Modells als Teil des *Quelltextes* der Software, werden alle Änderungen am Prozess sichtbar und nachvollziehbar.

Ein großes Hindernis bei diesem Ansatz ist, dass aktuell noch keine Editoren existieren, die das Modellieren des Prozesses unterstützen. Des Weiteren bietet jedes der CD-Systeme

eine eigene Modellierungssprache an. Diese sind oft nicht ausgereift und erschweren die Implementierung von CD. Aus diesem Grund stellt sich die Frage, ob eine ausgereifte und oft verwendete Modellierungssprache, wie die *Business Process Modell and Notation (BPMN)*, sich zur Modellierung von CD eignet. Die Verwendung von BPMN bietet den Vorteil, dass sowohl eine grafische, als auch eine auf *Extensible Markup Language (XML)* basierende Repräsentation, spezifiziert ist. Dadurch können die Vorteile einer grafischen Modellierung mit dem PaC Ansatz kombiniert werden.

Da BPMN aus dem Bereich des *Business Process Management (BPM)* stammt, stellt sich die Frage ob Methoden oder Konzepte aus diesem Bereich auf CD anwendbar sind.

1.1. Struktur der Arbeit

Um das Thema der Arbeit behandeln zu können, müssen zunächst alle relevanten Konzepte, Methoden und Begriffe eingeführt werden. Dazu werden die beiden unterschiedlichen Themenkomplexe BPM und CD in Kapitel 2 vorgestellt. Im Anschluss wird das Thema der Arbeit in Kapitel 3 anhand aktueller Probleme motiviert. Zusätzlich wird der Zusammenhang zwischen der Modellierungssprache BPMN und CD über *Value Streams* hergestellt. Das Kapitel wird mit der Einführung der Forschungsfragen beendet.

Als nächstes werden in Kapitel 4 insgesamt 15 Eigenschaften von CD Sprachen und Systeme aus verschiedenen Konzepten abgeleitet. Um herauszufinden, welche Eigenschaften von aktuellen Implementierungen erfüllt werden, werden fünf unterschiedliche Systeme und deren Sprachen auf diese Eigenschaften untersucht. Anschließend wird geprüft, ob BPMN die Eigenschaften einer Sprache für CD theoretisch erfüllen kann.

Nachdem eine grundsätzliche Eignung von BPMN als Modellierungssprache für CD attestiert wurde, wird in Kapitel 5 ein Konzept für die Verwendung von BPMN vorgestellt. Dieses Konzept ist in zwei Teile gegliedert. Als erstes wird die allgemeine Verwendung von BPMN als Modellierungssprache für CD betrachtet. Im zweiten Teil wird ein Konzept zur Integration der Sprache in das System JARVIS vorgestellt. Zusätzlich wird beschrieben, wie BPM-Methoden zur Analyse und Optimierung von Modellen in JARVIS genutzt werden können.

Das Konzept wird in Kapitel 6 umgesetzt und in Kapitel 7 werden die Ergebnisse diskutiert. Die Arbeit wird mit Kapitel 8 komplettiert. Darin werden alle Erkenntnisse aus der Arbeit zusammengefasst und ein Ausblick gegeben.

2. Hintergrund

Inhalt

2.1. Business Process Management 3
2.1.1. BPM-Regelkreis 5
2.1.2. Fachliche und technische Prozessmodelle 6
2.1.3. Business-Process-Management-Systeme 7
2.1.4. Analyse und Optimierung von Prozessen 8
2.2. Business Process Model and Notation 9
2.2.1. Token-Konzept 9
2.2.2. Basiselemente 10
2.2.3. Prozess-Diagramme 10
2.2.4. Weitere Diagrammtypen der BPMN 17
2.2.5. Alternative Prozess-Modellierungssprachen 18
2.3. Camunda BPM 19
2.4. Value Streams 20
2.5. Software Delivery Prozess 20
2.6. Continuous Delivery 22
2.6.1. Grundlagen 22
2.6.2. Techniken und Konzepte 24
2.6.3. Modellierungssprachen für Delivery Prozesse 27
2.6.4. Continuous Delivery Implementierungen 27

2.1. Business Process Management

Der Begriff BPM wird in der Literatur sehr unterschiedlich definiert. Daher wird hier die Definition 2.1 der *European Association of Business Process Management (EABPM)* verwendet, die auch in verschiedenen Werken [FR14; Kom11] Anwendung findet.

Definition 2.1 (Business Process Management)**.** *„Business Process Management (BPM) ist ein systematischer Ansatz, um sowohl automatisierte als auch nicht-automatisierte Prozesse zu erfassen, zu gestalten, auszuführen, zu dokumentieren, zu messen, zu überwachen und zu steuern und damit nachhaltig die mit der Unternehmensstrategie abgestimmten Ziele zu erreichen.“[EAB14, S.62]*

Die Definition unterscheidet explizit zwischen automatisierten und nicht automatisierten Prozessen, um zu verdeutlichen, dass BPM sich nicht nur auf die Unternehmensstruktur bezieht, sondern auch auf die Systemlandschaft. Aus diesem Grund reicht zur Optimierung der Prozesse weder eine Umorganisierung, noch eine Erweiterung der IT aus [FR14].

BPM erstreckt sich vielmehr auf beide Bereiche und nimmt dadurch nicht nur Einfluss auf die Unternehmensorganisation, sondern auch auf die im Unternehmen eingesetzten Systeme. Die Einführung von BPM in Unternehmen wird meist anhand der folgenden Anforderungen motiviert:

1. Existierende Prozesse sollen verbessert werden
2. Existierende Prozesse sollen dokumentiert werden
3. Neue Prozesse sollen entwickelt und angewendet werden

Im Rahmen der Unternehmensdigitalisierung sollen neue verbesserte und vor allem IT gestützte Prozesse, veraltete und ineffiziente Vorgänge ablösen. Verbesserungen können zum Beispiel durch direkte Anbindungen von Systemen geschaffen werden, da damit manuelle Schritte, wie das „Abtippen“ von Formularen, wegfallen.

Die Dokumentation von bestehenden Prozessen kann aufgrund von Zertifizierungen und Standards notwendig sein oder sie soll als Referenz für (neue) Mitarbeiter dienen.

Eine Neueinführung von Prozessen ist im Vergleich zu den ersten beiden Fällen eher selten [FR14]. Durch eine geeignete Prozess-Modellierungssprache können neue Prozesse einfach modelliert, diskutiert und optimiert werden.

Bei einer Studie [Kom11, S. 20] wurde der Nutzen von BPM untersucht und dabei fielen unter anderem die folgenden drei Aspekte besonders positiv auf:

- Transparenz der Prozesse
- Verbesserung der Qualität
- Erhöhung der Produktivität

Zusätzlich zeigt die Studie, dass der Nutzen von BPM besonders in der Verbesserung und Dokumentation von bestehenden Prozessen liegt.

Prozesse

Prozesse stellen einen integralen Bestandteil im BPM dar. Definition 2.2 verdeutlicht, dass ein Prozess immer einem oder mehreren Zielen dient. Dabei ist das vordergründige Ziel dem Kunden und nachgelagert dem Unternehmen einen bestimmten Wert zu liefern.

Bei der Betrachtung von Prozessen wird nicht der Fokus auf die einzelnen Aktivitäten und Aufgaben gesetzt, sondern der Prozess wird im Ganzen betrachtet, bewertet und schließlich auch optimiert. Zusätzlich verdeutlicht diese Definition, dass die Tätigkeiten von Menschen oder auch Maschinen ausgeführt werden können [FR14].

Die Begriffe „Prozess“ und „Geschäftsprozess“ können synonym verstanden werden [EAB14, S. 525]. Durch die Bezeichnung eines Prozesses als Geschäftsprozess soll der Wert-liefernde Aspekt des Prozesses unterstrichen werden.

Definition 2.2 (Prozess)**.** *„Als Prozess wird eine Reihe von festgelegten Aktivitäten verstanden, die von Menschen oder Maschinen ausgeführt werden, um ein oder mehrere Ziele zu erreichen.“[EAB14, S.61]*

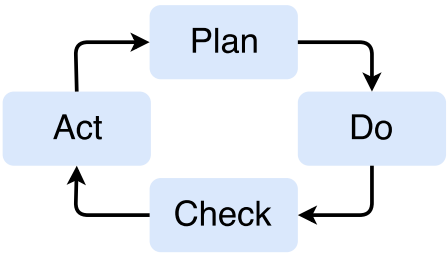

Abbildung 2.1.: PDCA-Regelkreis (auch Demingkreis genannt)

2.1.1. BPM-Regelkreis

BPM stellt kein Vorgehensmodell dar, sondern „ist eine wiederkehrende Management-Aufgabe, welche die sukzessive Optimierung und Anpassung der Prozesslandschaft an die sich ändernden Bedarfe des Unternehmens beinhaltet“ [Bar+10]. Daher lassen sich die Aufgaben des BPM als Regelkreis darstellen. Dieser besteht abhängig der zu Grunde liegenden Literatur aus einer verschiedenen Anzahl an Phasen. Allgemein lässt sich der Regelkreis als *Plan-Do-Check-Act (PDCA)*-Zyklus, wie in Abbildung 2.1 zu sehen, darstellen [EAB14, S.69 ff.]. Im Folgenden wird auf die verschiedenen Phasen eingegangen, um die Funktionsweise und Anwendung von BPM zu verdeutlichen. Ein Zyklus betrachtet im Allgemeinen immer alle Prozesse im Unternehmen. Eine Anpassung einzelner Prozesse ist dennoch möglich.

Die *Plan-Phase* betrachtet zwei verschiedene Aspekte. Als erstes wird das Unternehmensumfeld untersucht, mit dem Ziel Abhängigkeiten zwischen den Prozessen und dem Umfeld zu identifizieren. Dazu sollen unter anderem folgende Fragen beantwortet werden:

- Welche Kunden sind am Prozess beteiligt?
- Was ist der Kundennutzen?
- Welche Abhängigkeiten existieren zwischen den Prozessen?
- Welche Auslöser für Prozesse existieren?
- Was sind die *Key Performance Indicators (KPIs)*?

Nachdem das Umfeld der Prozesse untersucht wurde, können als zweites die einzelnen Prozesse genauer spezifiziert werden. Bei der Spezifikation stehen besonders die folgenden Aspekte im Vordergrund:

- Lieferobjekte (*Deliverables*)
- Beteiligte Systeme
- Beteiligte Rollen, Personen und Unternehmenseinheiten
- Prozess-Performance-Kennzahlen und deren Messpunkte im Prozess

Basierend auf den Performance-Kennzahlen kann später Optimierungspotenzial erkannt werden.

In der *Do-Phase* werden die in der vorherigen Phase erstellten Spezifikationen in Prozesse überführt und produktiv eingesetzt. Die Einführung eines neuen Prozesses erfordert oft weitere Anpassungen, wie zum Beispiel das Schaffen neuer Rollen oder die Weiterentwicklung der Unternehmensorganisation.

Damit eine kontinuierliche Verbesserung der Prozesse erfolgen kann, werden in der *Check-Phase* die erfassten Metriken ausgewertet. Diese Metriken sollen sicherstellen, dass sowohl Kundenbedürfnisse und -erwartungen erfüllt, als auch Zeiten und Kosten eingehalten werden.

Die *Act-Phase* ermöglicht die Optimierung der Prozesse und eine ständige Anpassung an neue Gegebenheiten. Basierend auf den Ergebnissen aus der vorherigen Phase werden die Daten in der Act-Phase analysiert und ein neuer Soll-Zustand der Prozesse erstellt. Dieser stellt die Grundlage für die anschließende *Plan-Phase* dar.

2.1.2. Fachliche und technische Prozessmodelle

Im Kontext von BPM wird grundsätzlich zwischen fachlichen und technischen Modellen unterschieden. Beide Arten unterscheiden sich im Wesentlichen im Detaillierungsgrad der Modellierung und der Zielgruppe [Kom11].

Ein fachliches Modell ist vor allem ein Kommunikationswerkzeug, um Prozesse dokumentieren, analysieren und optimieren zu können. Fachliche Modelle können als Basis für Arbeitsanweisungen im Unternehmen zum Einsatz kommen und werden in der Regel von einer Fachabteilung erstellt. Abhängig von der Zielgruppe, für die das Modell konzipiert ist, können Details im Modell weggelassen werden, wie zum Beispiel komplexe Ausnahmebehandlungen. Analysen und Optimierungen werden normalerweise auf Basis des fachlichen Modells durchgeführt. Es ist keine hundertprozentige Exaktheit des Modells nötig, da es von Menschen betrachtet wird und nicht durch eine Maschine ausgeführt werden soll. Aus diesem Grund kann das fachliche Modell als Abstraktion und Simplifizierung des technischen Modells verstanden werden.

Das technische Modell wird in der Regel auf Basis eines fachlichen Modells entwickelt und ermöglicht die Ausführung des Prozesses mittels einer Process Engine (siehe Abschnitt 2.1.3). Für die Ausführung des Prozesses muss dieses exakt bzw. detailliert genug beschrieben werden. Daher wird beim Erstellen eines technischen Modells auch von grafischer Programmierung gesprochen. Die Entwicklung des technischen Modells wird in der Regel von einer Entwicklungsabteilung übernommen. Ein wichtiger Unterschied zwischen technischen und fachlichen Modellen ist, dass technische Modelle oft Process Engine spezifische Implementierungsdetails beinhalten.

Es ist üblich, dass ein Prozess zuerst durch ein fachliches Modell abgebildet wird und später in ein technisches Modell überführt wird. Dabei wird das fachliche Modell ausgearbeitet, implementierungs-spezifische Details hinzugefügt und schrittweise in ein exaktes technisches Modell überführt. Da das technische Modell für die Analyse irrelevante Details enthält wird das fachliche Modell weiter für die Analyse benötigt und somit müssen beide Modelle parallel gepflegt werden.

Ein Idealzustand wäre, wenn das fachliche Modell dem technischen Modell entsprechen würde. Dies ist aber aus den bereits genannten Gründen nicht möglich. Dennoch kann, zum Beispiel durch das Modellieren mit einer gemeinsamen Sprache, die Synchronisierung beider Modelle vereinfacht werden. Dabei wird von *round-tripping* gesprochen, da ein und dasselbe Modell im kompletten Regelkreis verwendet werden kann. Ein weiterer Ansatz ist der hierarchische Aufbau eines Modells, wobei das fachliche Modell die Basis bildet und immer weiter verfeinert wird, um schließlich das technische Modell zu erhalten.

2.1.3. Business-Process-Management-Systeme

Das Managen von Prozessen und deren automatisierte Ausführung wird durch den Einsatz eines *Business Process Management System (BPMS)* erreicht. Dieses wird wie in Definition 2.3 festgelegt, als System verstanden, dass beliebige Prozessmodelle verwalten kann.

Definition 2.3 (BPMS)**.** *„Ein Business-Process-Management-System (BPMS) ist ein generisches IT-System, das in der Lage ist, auf Basis expliziter Prozessmodelle operative (Geschäfts-)Prozesse umzusetzen, ablaufen zu lassen und diese zu managen.“[EAB14, S.366]*

Ein BPMS erfüllt mindestens folgende grundlegenden Funktionen:

- Prozessausführung mittels Process Engine
- Funktionen zur Unterstützung/Abdeckung des BPM-Regelkreises
- Management der Prozesse (z.B. CRUD-Operationen)

Ausführung von Prozessen mittels Process Engine

Für die Automatisierung von Prozessen wird eine sogenannte „Process Engine“ verwendet [FR14]. Diese benötigt als Eingabe ein technisches Prozessmodell und kann darauf basierende Prozess-Instanzen starten. Das Prozessmodell kann Komponenten beinhalten, die nicht vollständig automatisierbar sind, wie zum Beispiel manuelle Überprüfungsschritte. Damit manuelle Aktivitäten abgebildet werden können, besitzt die Process Engine eine Benachrichtigungsfunktion, um Prozessbeteiligte über anstehende Aufgaben zu informieren. Die Prozessbeteiligten können ihre Aufgaben abarbeiten und dies der Process Engine mitteilen. Diese verarbeitet im Anschluss das Ergebnis (Human Workflow Management). Zusätzlich kann die Process Engine andere IT-Systeme oder Services aufrufen (Service-Orchestrierung).

Eine weitere Aufgabe der Process Engine ist die Entscheidungsfindung, welche Aufgaben oder Services wann und wie aufgerufen werden. Als Entscheidungsgrundlage verwendet die Engine das technische Prozessmodell, die Ergebnisse aus einer manuellen Aufgabe oder eines Service-Aufrufs. Durch diese Dynamik im Ablauf des Prozesses ist dieser nicht starr, sondern kann auf Ergebnisse der Aufrufe reagieren.

Neben der Ausführungsfunktionalität von Modellen besitzt eine Process Engine typischerweise die folgenden Funktionen:

- Versionierung von Modellen
- Datensammlung für Auswertungen
- Überwachung und Administration von Prozess-Instanzen

Die Versionierungen von Modellen ist eine sehr wichtige Funktionalität einer Process Engine um mit Anpassungen an den Prozessmodellen umgehen zu können. Dies erlaubt die kontinuierliche Verbesserung von Prozessen innerhalb des BPM-Regelkreises. Durch eine Versionierung der Modelle werden laufende Prozess-Instanzen nicht ungültig und es entfällt eine Migration. Neue Prozess-Instanzen werden automatisch mit dem neuen Prozessmodell ausgeführt und alte Instanzen können unverändert auslaufen.

Auch die Datensammlung stellt eine wichtige Funktionalität einer Process Engine dar, da sie den Grundstein für Auswertungen und Optimierungen legt. Die Process Engine dokumentiert dabei jegliche Ereignisse innerhalb der Prozess-Instanzen. Durch das ständige Sammeln von Daten kann der Ablauf von Prozessen überwacht, analysiert und ausgewertet werden. Die Ergebnisse dieser Auswertungen stellen die Grundlagen für Optimierungen an den Prozessen in der Act-Phase dar (vgl. Abschnitt 2.1.1).

2.1.4. Analyse und Optimierung von Prozessen

Die kontinuierliche Beobachtung und Verbesserung von Prozessen ist ein wichtiger Grundgedanke im BPM.

Prozess-Überwachung und Auswertung

Es wird zwischen zwei Konzepten unterschieden [EAB14, S. 238, S. 402 ff.]:

- Prozess-Reporting
- *Business Activity Monitoring (BAM)*

Beim Prozess-Reporting werden die verschiedene Kennzahlen einer Prozess-Instanz zu *Prozesstypkennzahlen* aggregiert und darauf basierend Auswertungen erstellt. Solche Kennzahlen können unter anderem die sogenannten KPI sein. Diese bilden die Prozessleistung ab und beziehen sich beispielsweise auf die folgenden Kennzahlen:

- Durchlaufzeit eines Prozesses
- Fehlerraten des Prozesses
- Kosten der Prozessdurchführung

Auf Basis der Prozesstypkennzahlen können Unregelmäßigkeiten während der Prozessausführung identifiziert werden, damit entsprechend reagiert werden kann. Durch die regelmäßige Aufzeichnung der Kennzahlen können Trends oder Abweichungen von Mittelwerten erkannt werden. Diese Methode wird als BAM bezeichnet.

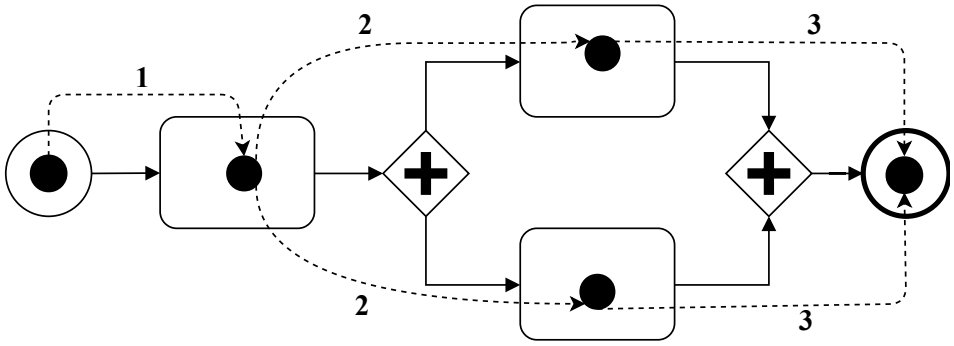

Abbildung 2.2.: BPMN: Token-Konzept

2.2. Business Process Model and Notation

Die BPMN wurde im Jahr 2011 von der *Object Management Group (OMG)* in der Version 2.0 verabschiedet [Obj11]. Diese hat unter anderem die *Unified Modeling Language (UML)* spezifiziert. Eine wesentliche Neuerung in der BPMN 2.0 Spezifikation stellt die Einführung eines Metamodells dar, welches eine direkte Ausführung der Modelle erlaubt. Dies hat zur Folge, dass sowohl fachliche, als auch technische Modelle in einer gemeinsamen Sprache entwickelt werden können. Dadurch kann der angestrebte round-trip mittels BPMN entscheidend unterstützt werden.

Um die verschiedenen Elemente der BPMN zu veranschaulichen wird zunächst das Token-Konzept vorgestellt.

2.2.1. Token-Konzept

Das Token-Konzept ist ein theoretisches Konzept mit dessen Hilfe der Fluss innerhalb eines Prozessmodells visualisiert und das Verhalten einer Komponente im Modell definiert werden kann [Obj11, S.27].

Die Funktionsweise ist in Abbildung 2.2 veranschaulicht. Am Startereignis wird das Token erzeugt und es folgt dem Sequenzfluss zur nächsten Aktivität im Prozess (1). Nachdem die Aktivität beendet wurde, wandert das Token weiter zum parallelen Gateway. Dieses sorgt dafür, dass das Token sich dupliziert und jeweils ein Token einem der ausgehenden Flüsse folgt (2). Zum Schluss werden beide Token im letzten Gateway zusammengeführt und eines der Beiden verschwindet. Das Endereignis sorgt für das Verschwinden des letzten aktiven Tokens und der Prozess ist somit beendet (3). Eine Prozess-Instanz befindet sich erst in einem Endzustand, wenn es keine aktiven Token mehr gibt.

Das Konzept vereinfacht das Verständnis der Funktionsweise der unterschiedlichen Elemente und kann auch intern von einer Process Engine verwendet werden. Dies ist aber nicht durch die BPMN Spezifikation vorgeschrieben [Obj11, S.27].

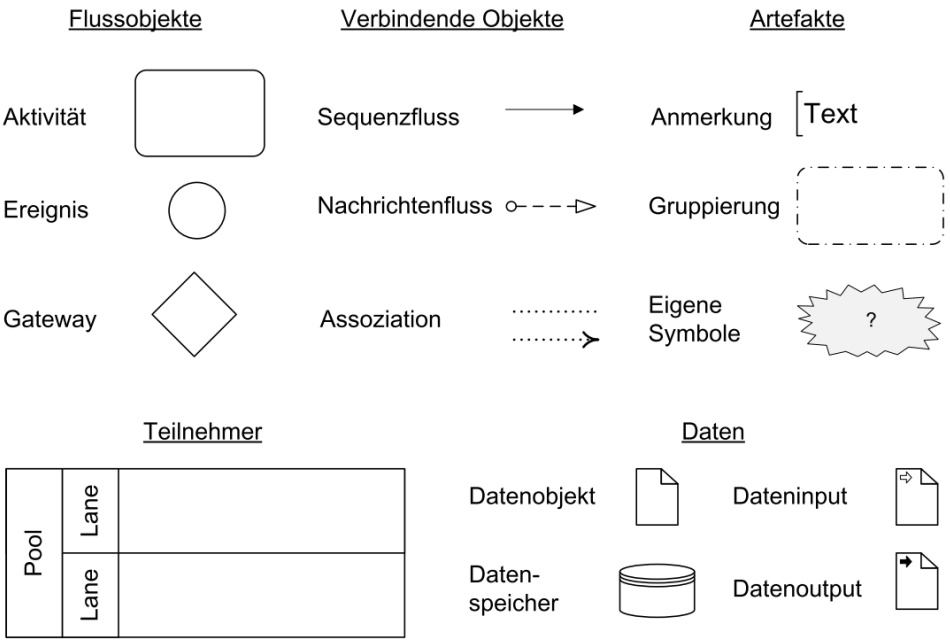

Abbildung 2.3.: BPMN: Basiselemente [FR14, S. 21]

2.2.2. Basiselemente

Für die Modellierung stellt die BPMN 2.0 Spezifikation verschiedene Basiselemente zur Verfügung [FR14]. Abbildung 2.3 zeigt die grafische Darstellung dieser Elemente.

Ein Prozess lässt sich mittels *Flussobjekten* und *Verbindende Objekten* darstellen. Zusatzinformationen zum Prozess geben *Artefakte* und *Daten*. Eine Strukturierung der Diagramme ist mittels *Teilnehmer* möglich. Die Darstellung von Teilnehmern wird durch *Lanes* und *Pools* erreicht. Mit Lanes können verschieden Personen, Rollen oder Anwendungen innerhalb eines Pools voneinander abgegrenzt werden. Eine Verschachtelung von Lanes ist möglich. Die genaue Semantik von Lanes ist in der BPMN 2.0 Spezifikation nicht definiert. Ein Pool umfasst verschiedene Lanes und repräsentiert die Grenzen des Prozesses. Zusätzlich tritt der Pool als *Orchestrator* auf und ist sowohl für die Steuerung und Orchestrierung des Prozesses als auch der einzelnen Elemente verantwortlich.

2.2.3. Prozess-Diagramme

Der Diagramm-Typ *Prozess* dient der Modellierung eines Prozesses. Daher ist hier offiziell keine Verwendung von Pools und Lanes vorgesehen. Die zur Verfügung stehenden Modellierungswerkzeuge bieten dennoch häufig beide Elemente an.

Im Folgenden werden die eingeführten Basiselemente weiter spezifiziert und deren Eigenschaften vorgestellt.

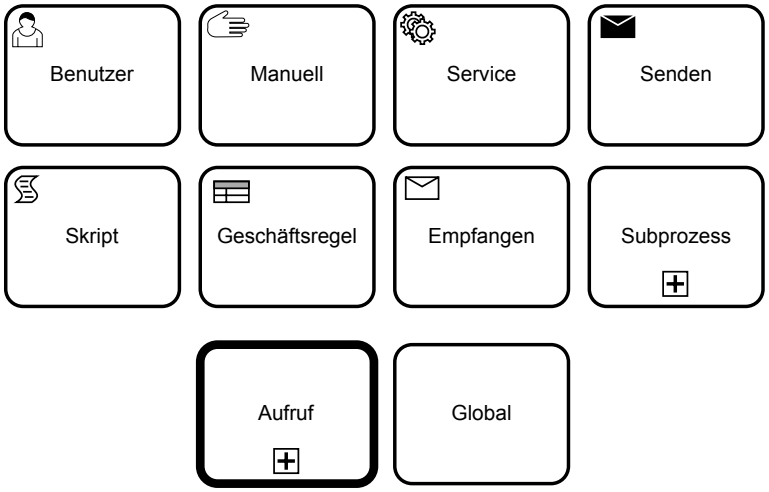

Abbildung 2.4.: BPMN: Aufgaben

Aktivitäten

Zu Flussobjekten zählen Aktivitäten, Ereignisse und Gateways. Aktivitäten werden unter anderem zur Modellierung von Aufgaben verwendet. Eine Aufgabe ist in BPMN eine atomare Aktivität, die nicht weiter heruntergebrochen werden kann [Obj11, S. 112 ff.]. Die möglichen Aufgaben der BPMN sind in Abbildung 2.4 zu sehen.

Die *Benutzer-Aufgabe* ist eine von einem Menschen zu erledigende Aufgabe, die explizit beauftragt wird und deren Ausführung von einer Process Engine überwacht wird.

Eine *Manuelle-Aufgabe* ist ähnlich der Benutzer-Aufgabe. Sie unterscheiden sich nur dadurch voneinander, dass die Manuelle-Aufgabe nicht von einer Process Engine überwacht wird. Daher eignet sich diese Aufgabe vor allem dann, wenn die zu erledigende Aufgabe abseits der Engine stattfindet, wie zum Beispiel das manuelle Transferieren von Dateien in sicherheitskritische Bereiche.

Eine *Service-Aufgabe* kann von einer Software automatisiert ausgeführt werden. Im Allgemeinen wird eine Service-Aufgabe mit Hilfe eines Webservices realisiert. Eine abweichende Implementierung der Service-Aufgabe ist erlaubt, muss aber von der Process Engine entsprechend unterstützt werden. Oft wird zusätzlich zu Webservices die Ausführung von Quellcode ermöglicht.

Weitere automatisierte Aufgaben sind die *Senden- und Empfangen-Aufgaben*. Sie werden verwendet um einen asynchronen Nachrichtenaustausch zu modellieren.

Durch eine *Skript-Aufgabe* kann Code ausgeführt werden. Dabei gibt die Process Engine die unterstützten Sprachen vor.

Durch die *Geschäftsregel-Aufgabe* können Geschäftsregeln in Kombination mit einem Gateway verwendet werden. Dadurch kann das Gateway basierend auf den Geschäftsregeln Entscheidungen treffen.

Das Metamodell definiert neben Aufgaben noch weitere Aktivitäten, wie zum Beispiel *Subprozesse*, *globale Aufgaben* und *Aufruf-Aktivitäten* [Obj11, S. 151].

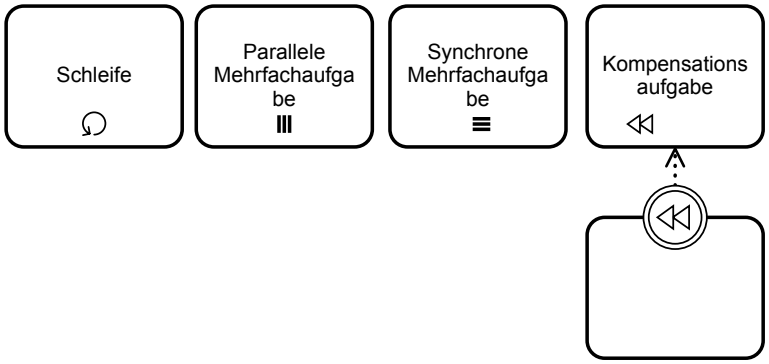

Abbildung 2.5.: BPMN: Markierungen an einer Aufgabe

Subprozesse dienen der Gliederung des Prozesses und mittels Aufruf-Aktivitäten können globale Aufgaben innerhalb eines Prozesses aufgerufen werden. Diese Modellierungstechnik ist sinnvoll, wenn an einer Aufgabe verschiedene Teilnehmer beteiligt sind. Die Aufgabe wird einmalig als globale Aufgabe modelliert und jeder Teilnehmer ruft an entsprechender Stelle diese Aufgabe mittels Aufruf-Aktivität auf.

Jede Aktivität kann zusätzlich, wie in Abbildung 2.5 zu sehen, eine Markierung erhalten. Die Markierung einer Aufgabe als *Schleife* bedeutet, dass die Aufgabe so lange wiederholt wird, bis eine bestimmte Bedingung eintritt. Eine *Mehrfachaufgabe* repräsentiert eine Aufgabe, die *parallel* oder *synchron* von mehreren Beteiligten erledigt werden muss. Falls eine Aufgabe unter bestimmten Bedingungen wieder rückgängig gemacht werden muss, kann Sie als *Kompensation* markiert werden. Wird mit der assoziierten Aufgabe ein Kompensations-Ereignis ausgelöst, wird eine Kompensationsaufgabe ausgelöst. Dadurch können Transaktionen realisiert werden.

Ereignisse

Ereignisse finden innerhalb eines Prozesses statt und nehmen Einfluss auf den Prozessablauf. Es wird zwischen den folgenden drei allgemeinen Ereignis-Typen unterschieden (vgl. Abbildung 2.6):

1. Startereignis
2. Zwischenereignis
3. Endereignis

Abbildung 2.6.: BPMN: Ereignis-Typen

Das *Startereignis* (1) stellt einen möglichen Startpunkt eines Prozesses dar. Daher besitzt es auch keine eingehenden Pfade.

Im Gegensatz dazu kann ein *Zwischenereignis* (2) auch eingehende Pfade besitzen. Manche Zwischenereignisse lassen sich an Aktivitäten „anheften“. Dadurch kann auf Ereignisse während der Ausführung einer Aktivität reagiert werden. Ein angeheftetes Zwischenereignis kann zusätzlich in einer unterbrechenden (3) und einer nicht-unterbrechenden (4) Variante vorkommen. Ein unterbrechendes Ereignis sorgt zum sofortigen Abbruch der Aktivität, wohingegen ein nicht-unterbrechendes Ereignis zu einer Duplizierung des Tokens führt. Ein Token verbleibt an der Aktivität, wobei das andere Token dem mit dem Ereignis assoziierten Pfad folgt. Ist die Aktivität beendet folgt das verbliebene Token dem regulären Pfad. Unterbrechende Ereignisse werden mit einem durchgehenden Kreis und nicht-unterbrechende Ereignisse mit einem gestrichelten Kreis dargestellt. Eine Mehrfachauslösung des Ereignisses ist möglich und sorgt jeweils für eine Duplizierung des Tokens, jedoch nur solange ein Token in der Aktivität vorhanden ist. Zwischenereignisse können auf viele verschiedene Arten und Weisen eingesetzt werden. Sie können den Standard-Fluss der Prozesse unterbrechen und somit eine Ausnahmebehandlung anstoßen oder sie können zur Modellierung von Verzögerungen und Zeitüberschreitungen verwendet werden. Zusätzlich können sie anzeigen, dass an einer bestimmten Stelle im Prozess Nachrichten erwartet werden oder dass innerhalb einer Transaktion zusätzliche Arbeit für die Kompensation im Fehlerfall notwendig ist.

Das Ende eines Prozess-Flusses wird mit Hilfe eines *Endereignisses* (5) modelliert. Dabei ist es wichtig, dass der Prozess nur beendet wird, wenn alle sich im Umlauf befindenden Token ein Endereignis erreicht haben.

Die allgemeinen Ereignis-Typen existieren jeweils in zwei verschiedenen Varianten:

- Eingetretenes Ereignis (catching event)
- Ausgelöstes Ereignis (throwing event)

Ein *eingetretenes Ereignis* besitzt einen Auslöser und tritt ein, wenn dieser Auslöser „gefeuert“ wird. Somit ist ein Startereignis immer ein eingetretenes Ereignis. Ein *ausgelöstes Ereignis* hingegen stellt selbst den Auslöser dar und sorgt dafür, dass ein Ereignis erzeugt wird. Trifft ein Token am Ereignis ein, wird dieses Ereignis gefeuert. Ein Endereignis ist somit immer ein ausgelöstes Ereignis.

Zusätzlich zu den bisher vorgestellten abstrakten Typen und deren Varianten gibt es zusätzlich zehn konkrete Ereignistypen. Im Folgenden werden die wichtigsten vier Typen, die Abbildung 2.7 zeigt, vorgestellt. Der Vollständigkeit halber sei hier auf die ausführliche Definition in der BPMN 2.0 Spezifikation [Obj11] verwiesen.

Innerhalb von Prozessen muss häufig Kommunikation zwischen Teilnehmern, zum Beispiel per E-Mail, modelliert werden. Dafür eignet sich das *Nachrichten-Ereignis*. Ein ausgefüllter Briefumschlag steht für das Senden einer Nachricht und ein unausgefüllter Umschlag stellt einen Nachrichten-Empfang dar. Wichtig ist, dass ein Senden-Ereignis keinen Prozess starten und ein Empfangen-Ereignis keinen Prozess beenden kann.

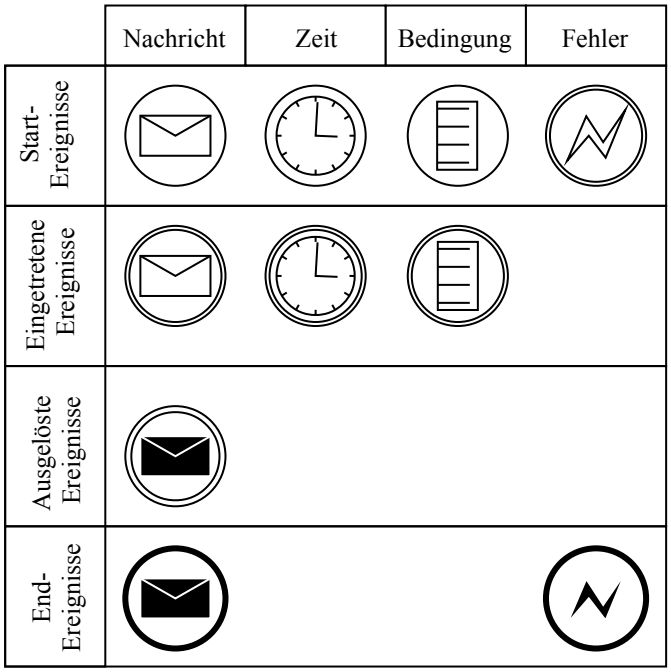

Abbildung 2.7.: BPMN: Konkrete Ereignis-Typen

Ein weiteres wichtiges Ereignis ist das *Zeit-Ereignis*. Es eignet sich zur Modellierung von vielen verschiedenen Situationen, wie zum Beispiel:

- Starten eines Prozesses in bestimmten Intervallen oder zu bestimmten Uhrzeiten
- Prozess eine bestimmte Zeit lang blockieren
- Zeitüberschreitungen einer Aktivität

Das Zeit-Ereignis ist immer ein eingetretenes Ereignis.

Zur Modellierung von Fehlern kann das *Fehler-Ereignis* eingesetzt werden. Die BPMN erlaubt die Verwendung des Ereignisses nur als angeheftetes Zwischenereignis oder als Endereignis. Das Ereignis unterbricht immer sofort den Ablauf. Befindet es sich innerhalb eines Subprozesses wird dieser beendet und der Fehler muss im übergeordneten Prozess behandelt werden.

Das Ereignis *Bedingung* wird ausgelöst, wenn eine bestimmte Bedingung wahr wird. Dies erlaubt zum Beispiel das Warten auf eine Grenzwertüberschreitung. Wird eine Bedingung als angeheftetes Ereignis verwendet, kann es zu einer Mehrfachauslösung kommen, indem die Bedingung nach einer Auslösung zwischenzeitlich nicht mehr erfüllt war und anschließend wieder erfüllt wird.

Abbildung 2.8.: BPMN: Gateways

Gateways

Soll eine Aktivität nur unter bestimmten Bedingungen ausgeführt werden, kann ein Gateway verwendet werden. Ein Gateway kann ein oder mehrere eingehende und ausgehende Pfade besitzen und kann in fünf unterschiedlichen Ausprägungen vorkommen (vgl. Abbildung 2.8).

Das erste Gateway ist ein sogenanntes *datenbasiertes exklusives Gateway* (1). Es erlaubt das Verzweigen eines Prozesses abhängig von Daten. Beim Verwenden dieses Gateways läuft das Token den anhand der Daten ausgewählten Pfad weiter zum nächsten Element. Zusätzlich ermöglicht es, verzweigte Prozess-Pfade wieder zusammenzuführen.

Parallel ablaufende Aktivitäten oder Subprozesse können mittels *parallelem Gateway* (2) realisiert werden. Dieses Gateway sorgt dafür, dass das ankommende Token dupliziert wird und jeder abgehender Pfad durch ein Token besucht wird. Eine Synchronisierung von parallelen Pfaden ist mittels dem gleichen Element möglich. Dabei wird auf alle Token gewartet und nachdem alle Token angekommen sind, verschwinden diese bis auf eines und dieses folgt im Anschluss dem ausgehenden Pfad.

Eine Kombination beider Gateways stellt das *datenbasierte inklusive Gateway* (3) dar. Es erlaubt die Ausführung von einem oder mehreren Prozess-Pfaden. Abhängig von den Daten wird ein eingehendes Token gegebenenfalls mehrfach dupliziert und an entsprechende Pfade weitergereicht. Wie beim parallelen Gateway kann dieses Gateway zur Synchronisierung dieser Pfade verwendet werden. Auch in diesem Fall „wartet“ das Gateway auf alle Token der eingehenden Pfade.

Ein eher selten verwendetes Element in BPMN ist das *komplexe Gateway* (4). Es wird für die Synchronisierung von Pfaden verwendet, hat aber die Eigenheit, dass es nicht auf alle aktiven Tokens warten muss, um ein Token weiter zu reichen. Dadurch lassen sich *m-out-of-n*-Bedingungen realisieren. Das bedeutet, dass wenn das Gateway n eingehende Pfade besitzt und bereits m Token am Gateway angekommen sind, dann wird genau ein Token weitergereicht.

Das letzte Gateway ist das *Ereignis-Gateway* (5). Dieses erlaubt es auf auftretende Ereignisse zu warten. Abhängig von dem eingetretenen Ereignis kann das Token einen ausgehenden Pfad wählen.

Wird ein datenbasiertes Gateway (1, 3) verwendet und keiner der ausgehenden Pfade kommt auf Basis der Daten in Frage kann ein *Standard-Fluss* verwendet werden. Das Token nimmt diesen Pfad, wenn kein Anderer ausgewählt werden kann.

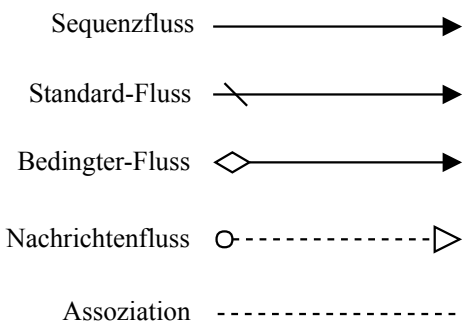

Abbildung 2.9.: BPMN: Flüsse und Assoziationen

Verbindende Objekte

Die zur Verfügung stehenden verbindenden Objekte sind in Abbildung 2.9 dargestellt. Flussobjekte werden mit den verbindenden Objekten *Sequenzfluss* oder *Nachrichtenfluss* miteinander verbunden. Der Unterschied zwischen Sequenzflüssen und Nachrichtenflüssen besteht in der Lokalität der miteinander verbundenen Flussobjekte. Befinden sich beide Objekte innerhalb eines Pools wird der Sequenzfluss verwendet, da die Objekte zu ein und dem selben Prozess gehören. Kommunizieren zwei Objekte aus unterschiedlichen Pools miteinander, handelt es sich um einen Nachrichtenaustausch zwischen zwei unterschiedlichen lokalen Prozessen und daher werden beide mit einem Nachrichtenfluss miteinander verbunden.

Der *Standard-Fluss* wird, wie bereits bei der Beschreibung der Gateways eingeführt, in Kombination mit einem Gateway verwendet. Handelt es sich zum Beispiel um ein datenbasiertes Gateway, wandert das Token über den Standard-Fluss, wenn kein anderer Pfad in Frage kommt.

Ein selten verwendeter Fluss ist der *bedingte Fluss* [FR14, S. 185]. An ihm können direkt Bedingungen geknüpft werden, wann dieser Pfad genommen werden soll.

Die allgemeine *Assoziation* kann verwendet werden um Daten oder Artefakte mit anderen Elementen zu verbinden. Bei Daten muss eine gerichtete Assoziation verwendet werden, um die Richtung des Datenflusses zu definieren.

Artefakte

Die grafische Darstellung von Artefakten wurde schon in Abschnitt 2.2.2 mit Abbildung 2.3 eingeführt. Sie ermöglichen die Anreicherung der Prozesse um zusätzliche Informationen, wie zum Beispiel die Ergänzung der Flussobjekte um Anmerkungen. Flussobjekte und Artefakte können mittels Assoziationen miteinander verbunden werden. Artefakte können aber auch unabhängig von Flussobjekten modelliert werden.

Ein *Hinweis-Artefakt* kann sich auf ein beliebiges Element im Prozess beziehen und hat keine Auswirkung auf den Prozess-Fluss. Es ermöglicht die Verwendung von informellen Hinweisen, die dem Betrachter des Prozesses helfen sollen. Das Artefakt *Gruppe* ermöglicht

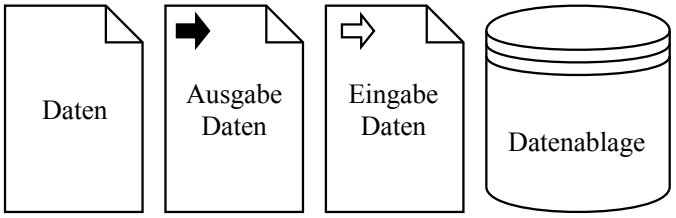

Abbildung 2.10.: BPMN: Daten und Datenablagen

eine informelle Strukturierung von Elementen. Dessen Verwendung hat keine Auswirkung auf die Ausführung eines Prozesses.

Daten und Datenablagen

Daten-Elemente ermöglichen die Modellierung von Informationen, die während des Prozesses benötigt oder erzeugt werden. Da Daten vor allem für Aktivitäten relevant sind, werden Datenobjekte mittels Assoziation mit diesen verbunden. Sie können aber auch mit Sequenzflüssen gekoppelt werden. Neben der expliziten Darstellung von Daten, wie in Abbildung 2.10 zu sehen, können Daten auch als Eingabe und Ausgabe von Aktivitäten genutzt werden. Diese sind nicht in der grafischen Repräsentation des Prozesses sichtbar, sondern werden als Eigenschaften im Modell eingebettet. Sollen diese Daten explizit modelliert werden, kann das Daten-Symbol mit einem schwarzen Pfeil als Ausgabe-Objekt oder mit einem weißen Pfeil als Eingabe-Objekt verwendet werden.

Die bisher vorgestellten Datenobjekte existieren nur innerhalb einer Prozess-Instanz, wohingegen Datenablagen instanzübergreifend existieren. Sie können zum Beispiel für die Modellierung von Datenbanken verwendet werden.

2.2.4. Weitere Diagrammtypen der BPMN

Neben dem bereits vorgestellten Diagramm-Typ *Prozess* definiert die BPMN Spezifikation weitere Diagramm-Typen. Für die Abbildung von Interaktionen zwischen Prozessen werden die Typen *Kollaboration, Choreographie* und *Konversation* verwendet [Obj11, S. 145].

Kollaborationen

Der Diagramm-Typ *Kollaboration* ermöglicht das Darstellen von Nachrichtenaustauschen zwischen verschiedenen Beteiligten [All09b]. Anstatt des Begriffs *Beteiligte* wird oft der Begriff *Partner* oder *Teilnehmer* verwendet. Alle Aktivitäten eines Teilnehmers werden in einem Pool dargestellt und die Nachrichten werden gesondert gekennzeichnet (vgl. Abbildung 2.11).

Mit Kollaborationen wird ein neues Element zur Modellierung eingeführt. Zur Darstellung einer Kommunikation wird ein *Konversationslink* verwendet (vgl. Abbildung 2.12).

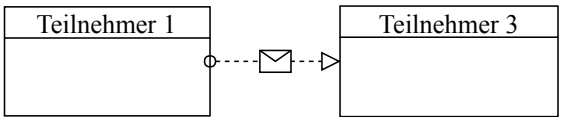

Abbildung 2.11.: BPMN: Kollaborationen

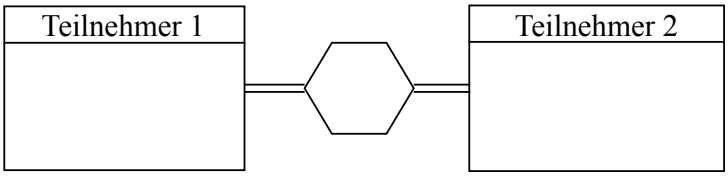

Abbildung 2.12.: BPMN: Konversation

Besteht ein Kollaborationsdiagramm im Wesentlichen nur aus Pools und Kommunikationslinks, wird oft von einem *Konversationsdiagramm* gesprochen [All09b]. Konversationen bieten einen Blick aus der Vogelperspektive auf verschiedene Kommunikationen.

Choreographien

Um die Kommunikation zwischen verschiedenen Unternehmen, auch *Business-to-Business (B2B)*-Kommunikation genannt, abbilden zu können, wurde der Diagramm-Typ *Choreographie* eingeführt [All09b]. Anders als bei einem *Kollaborationsdiagramm* liegt hier der Fokus auf dem Nachrichtenaustausch. Dazu wird der neue Aktivitätstyp *Choreographie-Aktivität* eingeführt. Diese Aktivität repräsentiert einen Nachrichtenaustausch (vgl. Abbildung 2.13). Der Sender der Nachricht wird im weiß hinterlegtem Feld angegeben und der Empfänger im Dunklen. Durch diese übersichtlichere Darstellung der Kommunikation tritt die Reihenfolge der Nachrichten in den Vordergrund und somit kann dieser Diagramm-Typ helfen Schnittstellen zwischen verschiedenen Partnern zu identifizieren und zu optimieren. Vom Kollaborationsdiagramm unterscheidet sich das Choreographie-Diagramm im Wesentlichem durch die Choreographie-Aktivität und der Darstellung der Nachrichtenaustausche unabhängig der internen Prozesse der Teilnehmer, da diese nicht mit abgebildet werden.

2.2.5. Alternative Prozess-Modellierungssprachen

Die BPMN ist aktuell die populärste Modellierungssprache für Prozesse [All09a] und wird von den meisten BPMS-Systemen unterstützt [LAS15]. Vor der Veröffentlichung der BPMN-Spezifikation war die Notation der *Ereignisgesteuerten Prozesskette (EPK)* sehr weit verbreitet. Diese wird heutzutage vor allem bei der fachlichen Modellierung von Geschäftsprozessen verwendet [All09a]. Sie hat aber mittlerweile an Bedeutung verloren.

Neben der im vorherigen Abschnitt vorgestellten BPMN gibt es zwei weitere Sprachen, die eine ähnliche Popularität besitzen. Zum einen die *WS-Business Process Execution Language (BPEL)* und zum anderen die *XML Process Definition Language (XPDL)*.

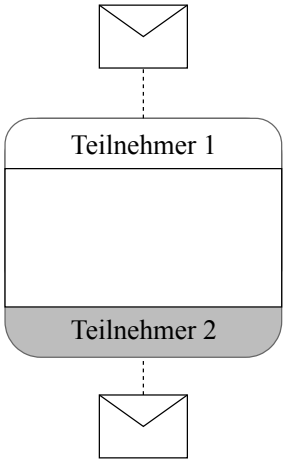

Abbildung 2.13.: BPMN: Choreographie

Beide Modellierungssprachen eignen sich zur Ausführung von Prozessen mittels Process Engines. Sie eignen sich aber weniger für die Visualisierung der Prozesse, da eine grafische Notation fehlt.

2.3. Camunda BPM

Camunda BPM ist ein leichtgewichtiges BPMS, das die Ausführung von Prozessen mittels eigener Process Engine erlaubt [camb]. Es basiert auf verschiedenen Komponenten, wie Abbildung 2.14 zeigt. BPMN-Diagramme können mit der Desktop-Anwendung *Camunda Modeler* oder mit einem anderen Modellierungswerkzeug, wie zum Beispiel dem Eclipse Plugin *Eclipse Modeler*, erstellt werden. Der round-trip soll mittels der Komponente *Camunda Cycle* unterstützt werden, indem sie bei der Synchronisation zwischen fachlichen und technischen Modellen hilft. Die *Tasklist* ist eine Webanwendung, die den Prozess-Teilnehmern ihre aktuellen Benutzer-Aufgaben zeigt und zusätzlich deren Abarbeitung mit Hilfe von Eingabemasken ermöglicht. Dafür kommuniziert sie über eine *Representational State Transfer (REST)*-API mit der Process Engine. Durch diese offene Schnittstelle können auch eigene Implementierungen an die Engine angeschlossen werden. Neben BPMN unterstützt Camunda BPM auch weitere Standards, wie zum Beispiel die *Case Management Model and Notation (CMMN)* und die *Decision Model and Notation (DMN)*. Zur Überwachung der Prozess-Instanzen wird die Komponente *Camunda Cockpit* verwendet. Sie ermöglicht die Überwachung der Prozesse während der Laufzeit und kann somit auch zur Analyse derer verwendet werden. Das Administrationstool *Camunda Admin* ermöglicht die Konfiguration von Benutzern und Gruppen, die Zugriff auf die Komponenten des Systems erhalten sollen.

Neben den bereits vorgestellten Open Source Komponenten stellt Camunda BPM noch verschiedene Enterprise-Komponenten zur Verfügung. Im Unterschied zu anderen BPMS ist Camunda BPM leichtgewichtig und sehr offen aufgebaut. Dadurch lässt es sich einfach erweitern und in eigene Systeme integrieren.

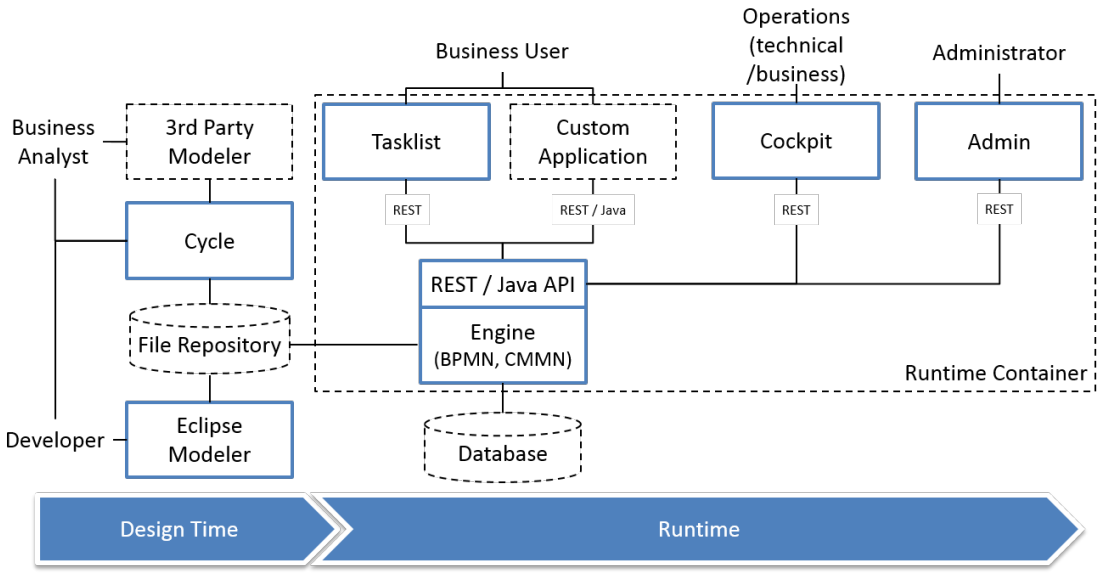

Abbildung 2.14.: Camunda BPM Architektur [camb]

2.4. Value Streams

Das Konzept des Value Streams [RS09; Kim+16], im deutschen auch Wertstrom genannt, kommt ursprünglich aus dem Bereich der „schlanken Produktion“ und wurde maßgeblich von Toyota geprägt. Ein Value Stream umfasst alle Aktivitäten, die während des Herstellungsprozesses eines Produktes ausgeführt werden müssen. Der Herstellungsprozess setzt sich aus vielen verschiedenen Teilprozessen zusammen, ausgehend von der Verarbeitung von Rohmaterialien bis hin zur Auslieferung des angefertigten Produktes an den Kunden.

Die Visualisierung und Analyse von Value Streams erlaubt die Identifikation und Minimierung von nicht wertschöpfenden Tätigkeiten („Muda“). Eine solche Modellierungssprache stellt *Value Stream Mapping (VSM)* dar. Durch die Darstellung wird die Untersuchung und Optimierung der Value Streams vereinfacht. Bei der Optimierung steht die Reduzierung von Verzögerungen im Vordergrund. Dazu werden nicht die einzelnen Prozesse innerhalb des Value Streams betrachtet, sondern das Zusammenspiel aller Prozesse.

2.5. Software Delivery Prozess

Der Software Delivery Prozess kann als Teilprozess des Software Development Zyklus verstanden werden. Dieser wird auch *Software Development Lebenszyklus* genannt und beginnt mit der Entscheidung eine Software zu entwickeln und endet nachdem diese ausgeliefert wurde [Ger+91, S. 67]. Ein Zyklus umfasst in der Regel die sechs Phasen: *Anforderungen identifizieren, Design ausarbeiten, Implementieren, Testen, Installieren und Überprüfen.* Eine konkrete Umsetzung der Phasen erfolgt innerhalb eines Vorgehensmodells, wie zum Beispiel dem Wasserfallmodell [LL13].

Der Software Delivery Prozess bezieht sich auf die Phasen Implementieren, Testen und Installieren des allgemeinen Lebenszyklus. Er lässt sich, wie in Definition 2.4 angegeben, als Prozess verstehen, der unterschiedliche Aktivitäten beinhaltet, um den Lebenszyklus zu unterstützen.

Definition 2.4 (Software Delivery Prozess)**.** *Der Software Delivery Prozess umfasst die Aktivitäten Build, Deploy, Test und Release und unterstützt damit den Software Development Zyklus, mit dem Ziel ein Software Produkt von der Entwicklung bis hin zur Installation in Produktivumgebungen zu unterstützen.[Dör18; HF10]*

Unabhängig von den Vorgehensmodellen ermöglichen diese Aktivitäten stabile Software an Kunden auszuliefern. Durch konkrete Umsetzungstechniken liefert der Software Delivery Prozess im Gegensatz zu Vorgehensmodellen direkt anwendbare Lösungsmöglichkeiten für verschiedene Probleme in der Entwicklung von Software. Wichtig ist, dass es keinen standardisierten Software Delivery Prozess gibt, sondern jedes Software Produkt benötigt seinen eigenen Prozess, der an die eingesetzten Frameworks und Rahmenbedingungen angepasst ist. Durch diese hohe Variabilität wird ein System benötigt, dass verschiedene Software Delivery Prozesse abbilden kann. Solch ein System wird *Software Delivery System (SDS)* genannt (siehe Definition 2.5). Es stellt neben der Ausführung der Prozesse weitere Funktionalitäten, wie Benachrichtigungen oder Darstellung von historischen Daten bereit.

Definition 2.5 (Software Delivery System)**.** *Ein Software Delivery System ist ein Software System, dass verschiedene Komponenten und Konfigurationsmöglichkeiten bereitstellt, um unterschiedliche Software Delivery Prozesse abbilden zu können. [Dör18]*

Damit das SDS einen Software Delivery Prozess ausführen kann, muss dieser als Software Delivery Modell vorliegen. Ein solches Modell beschreibt einen Software Delivery Prozess und definiert die einzelnen Aktivitäten und Abläufe innerhalb des Prozesses. Es dient als Konfiguration für ein SDS.

Die Modellierung des Prozesses erfolgt mit Hilfe einer Software Delivery Beschreibungssprache. Im Folgenden wird der Begriff *Modellierungssprache* synonym verstanden. Aktuelle Beschreibungssprachen folgen der DevOps-Technik *infrastructure-as-code* [LIL15] und bilden den Prozess als Code ab (PaC). Der Zusammenhang zwischen den hier eingeführten Begriffen wird in Abbildung 2.15 zusammengefasst. Im Folgenden werden die Begriffe *Modell* und *Delivery Modell* synonym für *Software Delivery Modell* verwendet. Anstatt *Software Delivery Prozess* werden die Begriffe *Delivery Prozess* und *Pipeline* bzw. wenn es aus dem Kontext ersichtlich ist nur *Prozess* verwendet.

Alle Techniken und Best-practices die den Delivery Prozess unterstützen, können unter dem Begriff *Continuous Delivery* zusammengefasst werden. Im folgenden Abschnitt werden diese vorgestellt.

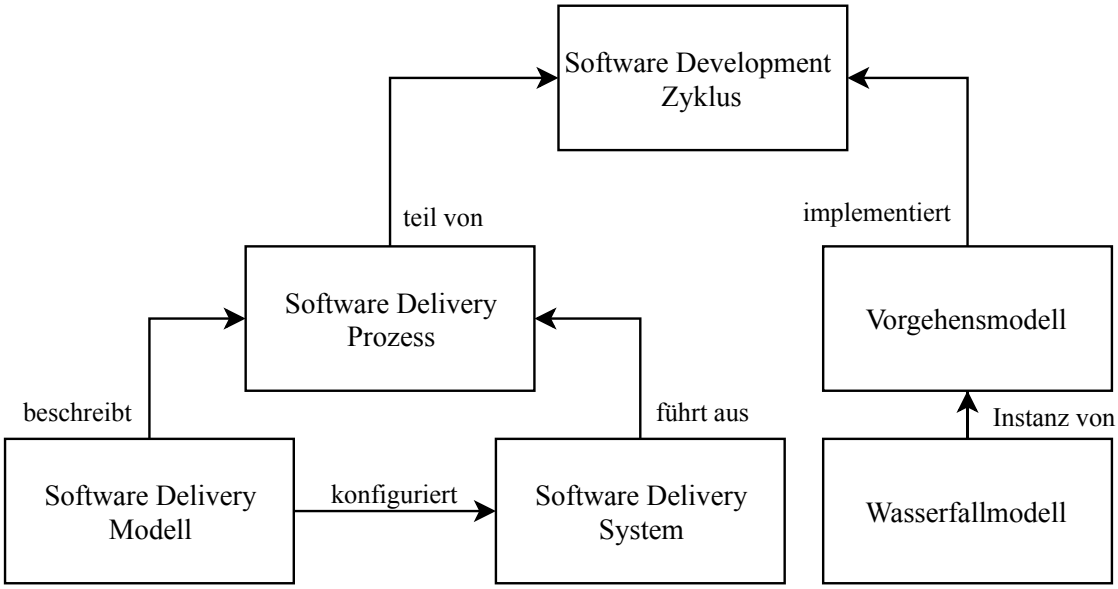

Abbildung 2.15.: Statische Relationen zwischen Software Delivery System, Prozess, Modell, Lebenszyklus und Vorgehensmodellen

2.6. Continuous Delivery

Zunächst werden die Grundlagen und Basiskonzepte definiert. Nachdem die verschiedenen Techniken und Konzepte von CD eingeführt wurden, werden die in dieser Arbeit betrachteten Software Delivery Beschreibungssprachen und Systeme näher gebracht. Komplettiert wird dieser Abschnitt mit einer Betrachtung von Analyse- und Optimierungskonzepten.

2.6.1. Grundlagen

Der Begriff Continuous Delivery wurde maßgeblich von Humble und Farley [HF10] geprägt und fasst verschiedene Techniken und Vorgehensweisen für die Unterstützung des Software Delivery Prozesses zusammen. In ihrem Buch *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation* [HF10, S. 3] stellen die Autoren fest, dass bisherige Ansätze zwar die betreffenden (Teil-) Prozesse abdecken, aber nicht deren Zusammenspiel. Dadurch fehlen entscheidende Teile in der Betrachtung der Value Streams, wodurch eine effiziente und verlässliche Umsetzung der Prozesse erschwert wird.

Die Implementierung von CD bietet verschiedene Vorteile. Durch eine Reduzierung der sogenannten *Cycle Time* kann schnell auf Fehler oder Veränderungen reagiert werden. Die Cycle Time gibt an, wie lange es braucht bis ein Feature, beginnend von der Idee, einem Kunden ausgeliefert werden kann. Die verschiedene Praktiken von CD führen dazu, dass Fehler früh im Sofware Development Prozess erkannt und behoben werden können. Dadurch sinken im Allgemeinen die Kosten und zusätzlich wird die Verlässlichkeit neuer Software-Versionen gesteigert. Vor allem die Automatisierung von Prozessen steigert die

Produktivität der Entwickler, da dadurch wiederkehrende Aufgaben „per Knopfdruck“ erledigt werden können.

Im Folgenden werden die Konzepte [HF10] vorgestellt, die die Basis für CD bilden.

Automatisierung

Mittels Automatisierung können menschliche Fehler verhindert werden und der Software Delivery Prozess ist jederzeit wiederholbar. Die Automatisierung hat den weiteren Vorteil, dass automatisierte Aktivitäten leicht parallelisiert werden und somit die Laufzeiten der Pipelines drastisch verkürzt werden können.

Arbeiten in kleinen Einheiten

Dieses Konzept bezieht sich auf die Anzahl an Änderungen, die zwischen zwei aufeinanderfolgenden Versionen an einer Software vorgenommen werden. Durch wenige Änderungen zwischen unterschiedlichen Versionen sinkt das Risiko von neu eingeführten Fehlern und ein Zurückrollen auf eine vorherige Version ist einfacher möglich. Zusätzlich führen kurze Release-Zyklen zu schnellerem Feedback, da Kunden früher neue Features testen können.

Eingebaute Qualität

Dieses Prinzip stammt aus der Schlanken Produktion, die bereits in Abschnitt 2.4 eingeführt wurde. Das Ziel ist es möglichst schnell Feedback zu erhalten, um rechtzeitig auf Probleme reagieren zu können.

Methoden für eingebaute Qualität sind im Wesentlichen CI und die intensive Verwendung von automatisierten Tests.

Jeder ist verantwortlich für den Delivery Prozess

Dieses Prinzip ist eines der zentralen Prinzipien der DevOps-Bewegung. Im Idealfall kennt jeder Mitarbeiter die Ziele eines Unternehmens und alle arbeiten gemeinsam an der Umsetzung dieser Ziele. In der Realität wird hingegen oft ein Inseldenken praktiziert, das zu einer Abgrenzung unter den Abteilungen führt. Dies hat zur Folge, dass die einzelnen Abteilungen nicht optimal zusammenarbeiten und ein gegenseitiges Beschuldigen bei Fehlern oder Problemen stattfindet (*Blame Game*).

Dadurch, dass jeder verantwortlich für den Delivery Prozess ist, wird die Kommunikation zwischen den Mitarbeiter und Abteilungen gefördert. Dies wiederum sorgt für eine schnellere Problembeseitigung.

Kontinuierliche Verbesserung

Jedes Softwaresystem und jeder Prozess unterliegt einer kontinuierlichen Veränderung, da sich mit der Zeit Anforderungen ändern (vgl. [HF10, S. 28 f.]). Aus diesem Grund

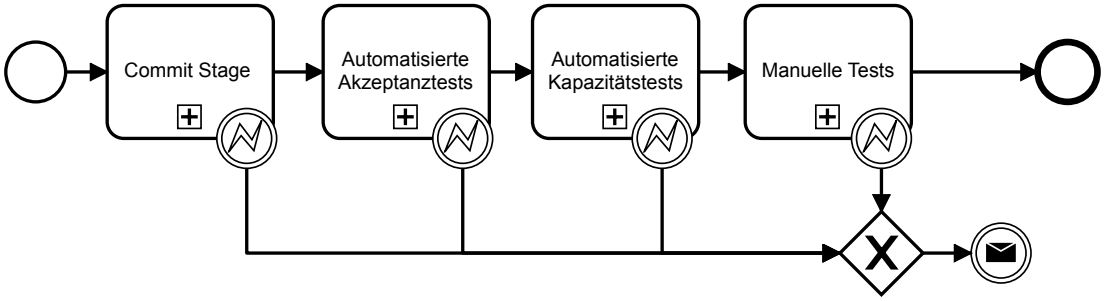

Abbildung 2.16.: Allgemeiner Delivery Prozess

sollte regelmäßig der Delivery Prozess überprüft und bei Bedarf angepasst werden. Kontinuierliche Verbesserung kann, wie der in Abschnitt 2.1.1 vorgestellte BPM-Regelkreis, als Demingkreis dargestellt werden.

2.6.2. Techniken und Konzepte

Neben den Basiskonzepten werden unter dem Begriff CD verschiedene Techniken und Konzepte zusammengefasst [HF10].

Deployment Pipelines

Mit CD werden verschiedene Konzepte eingeführt, die den Software Delivery Prozess unterstützen und ihn robuster gestalten sollen. Das zentrale CD-Konzept ist die *Deployment Pipeline*.

Im Allgemeinen werden unter dem Begriff *Deployment Pipeline* drei unterschiedliche Konzepte zusammengefasst (vgl. Abschnitt 2.5):

1. Software Delivery Prozess
2. Software Delivery Modell
3. Software Delivery System

Der Delivery Prozess besteht mindestens aus den in Abbildung 2.16 dargestellten Aktivitäten. Durch die Automatisierung von immer wiederkehrenden monotonen Aufgaben, kann eine große Fehlerquelle in den betreffenden Phasen des Lebenszyklus ausgeschlossen werden. Der Einsatz einer Deployment Pipeline bringt die folgenden Vorteile mit sich:

- Der Softwareauslieferungsprozess wird explizit modelliert und somit auch sichtbar für alle Beteiligten
- Sie verbessert das Feedback und unterstützt damit die frühzeitige Beseitigung von Problemen
- Sie erlaubt das vollautomatisierte Einrichten der Software in jeder Umgebung zu jeder Zeit

Continuous Integration

Der Begriff CI stammt ursprünglich aus dem *Extreme Programming* [Fow06] und bedeutet, dass Änderungen am Quellcode einer Software kontinuierlich in den Hauptzweig der Software integriert werden. Für eine erfolgreiche Integration wird der modifizierte Quellcode kompiliert und mit Hilfe von automatisierten Tests überprüft. Diese Überprüfungen finden sowohl vor dem Einfließen der Änderungen in den Hauptzweig, als auch im Hauptzweig selbst statt, wobei dort nicht nur die eigenen Änderungen integriert werden, sondern es fließen alle bis zu diesem Zeitpunkt gemachten Änderungen ein. Das Integrieren des Hauptzweiges in einer dedizierten Integrationsumgebung stellt sicher, dass die Software auch in einer produktiv-ähnlichen Umgebung korrekt läuft. Dadurch wird sichergestellt, dass von unterschiedlichen Entwicklern eingebrachte kollidierende Änderungen schnell erkannt und behoben werden. Das automatisierte Testen des Hauptzweiges wird von einem CI-System durchgeführt. Diese Tests stellen die Basis für die meisten der aktuellen CD-Systeme dar.

Baue ein Artefakt einmalig

Das einmalige Erstellen von Software-Artefakten bietet den Vorteil, dass die verschiedenen Tests innerhalb der Deployment Pipeline immer die gleichen Artefakte testen. Konnte ein Artefakt in den verschiedenen Phasen der Pipeline erfolgreich getestet werden, ist dieses prinzipiell auslieferbar. Im Gegensatz dazu sind Artefakte, die in jeder Phase neu erstellt werden, nicht vertrauenswürdig, da prinzipiell Unterschiede zwischen den Artefakten entstehen können. Ursachen können zum Beispiel unterschiedliche Umgebungen sein, in denen die Phasen ausgeführt werden.

Verteile deine Software immer gleich

Durch Verwendung des gleichen Auslieferungsprozesses der Software für jede Umgebung, kann sichergestellt werden, dass die Auslieferung der Software nicht nur in Testumgebungen funktioniert, sondern auch in Produktivumgebungen.

Verwende Deployment-Strategien

Obwohl die Deployment Pipeline nicht für die kontinuierliche Auslieferung der Software auf Produktiv-Systemen gedacht ist, muss ein SDS unterschiedliche Deployment-Strategien unterstützen, da sonst das Prinzip *Verteile deine Software immer gleich* nicht angewendet werden könnte. Im Folgenden werden kurz mögliche Strategien vorgestellt.

Die Verwendung von sogenannten *Zero-Downtime Releases* [HF10; Kim+16] ermöglicht den Austausch von unterschiedlichen Software-Versionen, ohne das Kunden dies bemerken, da kein Ausfall der Software stattfindet. Ein weiterer Vorteil bei der Anwendung einer einheitlichen Strategie ist, dass der Deployment-Prozess nachvollziehbar und wiederholbar ist. Beispiele solcher Strategien sind *Blue-Green Deployments* und *Canary Releasing* (siehe [Kim+16]).

Versioniere alles in der Versionskontrolle

Wird nicht jegliche Konfiguration der Deployment Pipeline versionert, kommt es zu sogenannten *Snowflake-Systemen* [Fow12]. Das bedeutet, dass jedes SDS einzigartig ist und Unterschiede zwischen den Systemen schwer nachvollziehbar sind.

Jede Konfiguration sollte daher in einer Versionskontrolle hinterlegt werden. Dazu zählen nicht nur die Delivery Modelle, sondern auch jegliche Konfiguration des SDSs. Dadurch sind Änderungen an der Konfiguration der Deployment Pipeline nachvollziehbar, wiederholbar und die Bildung von Snowflake-Systemen wird verhindert.

Verwende Smoke-Tests für die Verteilung deiner Software

Die Verwendung von *Smoke-Tests* stellt sicher, dass neu ausgelieferte Software und deren Komponenten sich in einem stabilen Zustand befinden. Dazu werden oberflächliche Tests durchgeführt, die eine grundsätzliche Funktionstüchtigkeit der Software überprüfen.

Frühes und schnelles Feedback

Dieses Prinzip bezieht sich auf den kompletten Delivery Prozess und soll sicherstellen, dass Probleme und Fehler einer Software frühzeitig erkannt und behoben werden können. Daher ist in allen Phasen des Delivery Prozesses Feedback vorgesehen.

Fail Fast

Fail Fast bedeutet, dass das SDS die Ausführung eines Delivery Prozesses abbricht, wenn etwas Unerwartetes auftritt. Dies ist besonders für schnelles Feedback wichtig. Tritt ein Fehler zum Beispiel während der Ausführung von Tests auf, sollte das System die Ausführung am Ende der Phase abbrechen und die Benutzer entsprechend informieren. Das Abbrechen des Prozesses ist vor allem dann wichtig, wenn der Fehler Auswirkungen auf nachfolgende Phasen im Prozess hat oder die Ausführung des Prozesses sehr lange dauert. Durch das Abbrechen des Prozesses erst am Ende einer Phase werden alle erkannten Fehler aggregiert erfasst und können somit zusammen behoben werden [HF10].

Neben dem Erkennen von Fehlern sollte das SDS auch Timeouts unterstützten. Das bedeutet, dass das System die Ausführung abbricht, sobald eine Aktivität im Prozess ungewöhnlich lange ausgeführt wird.

Parallelisiere den Delivery Prozess

Neben der Automatisierung ist die Parallelisierung des Delivery Prozesses einer der wichtigsten Maßnahmen zur Minimierung der Cycle Time [HF10; Mug15]. Dies geschieht, indem unabhängige Phasen oder Aktivitäten parallel ausgeführt werden. Zusätzlich unterstützt das Parallelisieren von langlaufenden Tests frühes Feedback.

2.6.3. Modellierungssprachen für Delivery Prozesse

Die aktuellen Sprachen können zwei grundlegenden Sprach-Typen zugeordnet werden. Diese werden im Folgenden vorgestellt.

Imperative Modellierungssprachen

Imperativ bedeutet hier, dass genau beschrieben wird *wie* die Pipeline aussieht und welche Aktionen ausgeführt werden müssen [Arm16]. Das imperative Modellieren erfordert in der Regel die Verwendung einer Skriptsprache oder einer internen *Domain Specific Language (DSL)*. Häufig werden auch Teilmengen einer Skriptsprache verwendet.

Das Modellieren erfolgt dabei „Schritt-für-Schritt“, d.h. jeder Schritt muss exakt definiert werden. Der Ablauf der Pipeline ist stark von der konkreten Implementierung abhängig. Veränderungen in der Software oder der Umgebung führen oft zu Anpassungen an der Pipeline.

Ein Vorteil von imperativen Sprachen ist, dass ein Softwareentwickler in der Regel mit der Sprache vertraut ist oder die Lernkurve für eine DSL annehmbar ist. Problematisch ist, dass die kontinuierliche Erweiterung der Skripte häufig zu sehr komplexen Pipelines führt, die unübersichtlich und schwer zu warten sind.

Deklarative Modellierungssprachen

Im Gegensatz zur imperativen Modellierung ist der Fokus bei den deklarativen Modellierungssprachen auf dem *was* in der Pipeline gemacht werden soll [Arm16]. Dabei wird nicht festgelegt *wie* der Ablauf konkret ist. Das ausführende System analysiert die Pipeline und bestimmt selbst *wie* sie die Pipeline umsetzt. Das hat den Vorteil, dass Implementierungsdetails vor dem Entwickler versteckt werden. Zusätzlich wird viel Logik aus der Pipeline-Definition in das ausführende System verschoben und die Definition wird kürzer und lesbarer. Ein weiterer Vorteil ist, dass aufgrund der besseren Lesbarkeit, die deklarative Modellierung die Identifikation von Fehlern vereinfacht. Ein Nachteil der deklarativen gegenüber der imperativen Modellierungssprache ist, die starke Systemabhängigkeit bei der Umsetzung der Pipelines. Durch den hohen Aufwand bei der Erstellung deklarativer Modellierungssprachen, sind diese meist deutlich weniger mächtig.

2.6.4. Continuous Delivery Implementierungen

In diesem Abschnitt werden die in der Arbeit betrachteten Systeme kurz vorgestellt. Eine vollständige Einführung in alle Systeme würde den Rahmen dieser Arbeit sprengen. Aus diesem Grund wird nur Jenkins detailliert vorgestellt. Weitere Informationen sind in den angegebenen Quellen zu finden.

Die untersuchten Systeme werden bereits seit einiger Zeit verwendet und können daher als „erwachsen“ angesehen werden. Neben diesen Systemen wird in dieser Arbeit das an der RWTH Aachen entwickelte SDS JARVIS betrachtet. Die aktuelle Implementierung ist allerdings noch ein Prototyp.

```
#!/usr/bin/env groovy

node {
    def workspace = pwd()
    echo "Building Job at ${workspace}"

    stage('Shell') {
        try {
            sh returnStdout: true, script: 'doSomething.sh'
        }
        catch (exc) {
            echo 'Something failed!'
        }
    }
}
```

Quelltext 2.1: Jenkinsfile1.groovy

Jenkins

Jenkins ist ein im Jahr 2011 erschienenes Automatisierungstool, das sich durch seine große Flexibilität und Erweiterbarkeit auszeichnet. Es basiert auf dem 2005 erschienenem Hudson, wurde in Java implementiert und unterstützt verschiedene Plattformen, wie zum Beispiel Java oder .NET [Sof].

Die große Flexibilität von Jenkins wird durch dessen Plugin-Architektur erreicht. Jenkins implementiert selbst nur die Basisfunktionalitäten und kann durch eine große Anzahl an bereits existierender Plugins erweitert werden. Eines der wichtigsten Plugins im Rahmen von CD stellt das *Pipeline Plugin* dar. Dieses fasst verschiedene Plugins zusammen, die die Implementierung von CD unterstützen.

Der Delivery Prozess wird bei Jenkins [Jd] als PaC mit Hilfe eines sogenannten *Jenkinsfiles* abgebildet. Dieses wird mit dem Source Code der zu bauenden Software in die Versionskontrolle eingecheckt. Damit eine Pipeline ausgeführt werden kann, muss über die Jenkins-Oberfläche ein entsprechender *Job* konfiguriert werden. Für die Implementierung der Pipeline stehen aktuell zwei verschiedene DSLs zur Verfügung. Zum einen die imperative *Scripted Pipeline* DSL und zum anderen die *Declarative Pipeline* DSL.

Eine Beispiel-Pipeline in der *Scripted Pipeline* DSL ist in Quelltext 2.1 zu finden. Die *Scripted Pipeline* DSL basiert auf Groovy und besitzt daher eine sehr große Flexibilität. Dies hat aber den Nachteil, dass die Entwickler der Pipeline mit Groovy vertraut sein müssen oder eine entsprechend steile Lernkurve bewältigen müssen.

Aus diesem Grund wurde die *Declarative Pipeline* DSL entwickelt. Diese schränkt den Funktionsumfang ein und legt eine feste Struktur der Pipeline fest. Dadurch können diese übersichtlicher und entsprechend einfacher implementiert werden. Um dennoch allen Anforderungen gerecht zu werden, ist die Verwendung von Skript-Teilen innerhalb der Pipeline möglich. Die Einbettung von *Scripted Pipeline* DSL Code ist bei komplexeren

```
1  #!/usr/bin/env groovy
2
3  pipeline {
4      agent any
5      stages {
6          stage("build") {
7              steps {
8                  retry(3) { build("build-job") }
9              }
10         }
11         stage("tests") {
12             steps {
13                 parallel (
14                     "unit test" : {
15                         build("unit-test-job")
16                     },
17                     "component test" : {
18                         build("component-test-job")
19                     }
20                 )
21             }
22         }
23     }
24 }
```

Quelltext 2.2: Jenkinsfile2.groovy

Pipelines auch zwingend notwendig, da viele Aktivitäten (noch) nicht von Jenkins ausgeführt werden können. Ein weiteres Problem besteht in der Notwendigkeit bestehende Plugins so zu erweitern, dass diese mit der *Declarative Pipelin* DSL kompatibel sind. Dadurch stehen aktuell viele Plugins in der deklarativen Pipeline nicht zur Verfügung. Weitere Probleme ergeben sich daraus, dass Jenkins für die Umsetzung der Pipeline verantwortlich ist. Da bei der strikten Verwendung dieser DSL zum Beispiel keine Integrationen mit anderen Software-Systemen per Skript möglich ist, wird die Ausdruckskraft der Pipeline stark eingeschränkt. Ein Beispiel in der DSL ist in Quelltext 2.2 zu finden. Die Pipeline besteht aus den Phasen „build“ und „tests“. Um die Tests des Projekts ausführen zu können wird zunächst in der „build“ Phase das Projekt gebaut. Anschließend werden die Test-Schritte „unit test“ und „component test“ parallel ausgeführt.

Die Visualisierung der Pipeline wird mittels *Pipeline Stage View Plugin* ermöglicht. Dieses bietet einen einfachen und schnellen Überblick über die letzten Durchläufe einer Pipeline. Aktuell gibt es keine Visualisierung von parallelen Aktivitäten (vgl. Quelltext 2.2 Zeile 13).

GoCD

GoCD oder Go ist erstmalig unter dem Namen Cruise im Jahre 2007 erschienen [Abe14]. Es ist Open Source und wurde als Alternative zu bereits existierenden CI Servern von ThoughtWorks entwickelt, da diese nur CI betrachten und nicht CD [Fow14].

Das Erstellen von Delivery Modellen erfolgt in der Regel über Dialoge in einer Web-Oberfläche. Die Modelle werden im Hintergrund als XML-Dokumente persistiert. Um einen einfachen Umgang mit den Modellen zu ermöglichen, können durch Plugins neue Sprachen hinzugefügt werden. Aktuell existieren Plugins für *JavaScript Object Notation (JSON)* und *YAML Ain't Markup Language (YAML)*.

Das Datenmodell von GoCD sieht vor, dass eine Pipeline aus Stages besteht und diese wiederum aus Jobs. Ein Job fasst beliebig viele Tasks zusammen. Diese Tasks können entweder vordefinierte Kommandos wie *maven compile* sein oder beliebige Befehle die in einer Shell ausgeführt werden.

Codeship

Das Unternehmen Codeship wurde 2011 gegründet und bietet seit dem die CD-Lösung Codeship an. Codeship wurde 2018 von CloudBees aufgekauft [Lab18].

Die Software wird als *Software as a Service (SaaS)*-Lösung zum einen als Basic- und zum anderen als Pro-Variante angeboten. Im Gegensatz zur Basic-Variante unterstützt die Pro-Variante Docker und PaC mittels einer YAML-DSL. Alternativ kann JSON verwendet werden.

Eine Pipeline besteht in Codeship aus beliebig vielen Steps die jeweils in einem eigenen Docker-Container gestartet werden. Das Parallelisieren von Steps ist möglich.

Concourse

Concourse wurde 2014 von Pivotal Software, Inc. als Open Source Software veröffentlicht und erlaubt das deklarative Modellieren von Delivery Prozessen [Piva]. Ein Prozess besteht bei Concourse aus Jobs, die wiederum aus einzelnen Tasks bestehen. Diese werden, wie bei Codeship, jeweils in einem eigenen Docker-Container ausgeführt. Des Weiteren erfolgt die Modellierung in einer YAML-DSL.

Ein zentrales Ziel von Concourse ist die Vermeidung von Snowflake-Systemen [Kum16]. Die Konfiguration des Systems erfolgt statisch und kann nicht zur Laufzeit des Systems verändert werden.

Wercker

Wercker wird vom gleichnamigen Unternehmen seit 2011 entwickelt und wird nur als SaaS zur Verfügung gestellt [Ora]. Im Jahre 2017 wurde das Unternehmen von Oracle akquiriert. Im Gegensatz zu den bereits vorgestellten Lösungen unterscheidet Wercker zwischen *Workflows* und *Pipelines*. Dabei orchestriert ein Workflow beliebig viele Pipelines.

Auch Wercker bietet eine native Unterstützung von Docker-Containern und Pipelines können in einer YAML-DSL modelliert werden. Neben dem Ausführen von Shell-Befehlen können auch vordefinierte Aktionen ausgeführt werden.

JARVIS

JARVIS wurde 2017 im Rahmen einer Masterarbeit entwickelt [Dör18]. Es basiert auf einer Microservice-Architektur und kann über Docker ausgeführt werden.

Das System unterscheidet, ähnlich wie GoCD, zwischen einem internen und externen Delivery Modell. Dadurch können Delivery Prozesse mit Hilfe von unterschiedlichen Modellierungssprachen umgesetzt werden. Wichtig dabei ist, dass die externen Modelle automatisch in ein internes Modell überführt werden können.

Des Weiteren besitzt JARVIS die Eigenschaft der *Selbstorganisierung*. Dies bedeutet, dass das Modell keinen konkreten Ablauf des Delivery Prozesses vorgeben muss, sondern das System ermittelt anhand der zur Verfügung stehenden Informationen den Ablauf. Dabei ermittelt JARVIS nicht nur die Reihenfolge der Aktivitäten, sondern es untersucht zusätzlich die Ein- und Ausgabe-Spezifikationen der Aktivitäten und ermittelt Datenübergabepunkte zwischen den Aktivitäten.

3. Motivation

Inhalt

3.1. Forschungsfragen 34

Die Techniken und Konzepte von CD bieten viele verschiedene Vorteile, wie zum Beispiel kürzere *Time-to-Market* Zeiten oder die Steigerung der Zuverlässigkeit von Releases [Rod+17]. Dennoch ist die Einführung von CD mit Schwierigkeiten verbunden [CSA15], da vor allem die aktuellen Werkzeuge keine vollständige Umsetzung von CD ermöglichen [Che17].

Im Rahmen eines Literatur Reviews wurde zusätzlich festgestellt, dass aktuelle Delivery Modelle zu kompliziert und komplex sind [LIL17]. Dadurch entsteht ein erheblicher Wartungsaufwand für den Delivery Prozess. Untersuchungen haben ergeben, dass 27% des Entwicklungsaufwands und 44% des Aufwands für Tests auf *Build Maintenance* entfallen [McI+11]. Aus diesem Grund ist es wichtig, dass die Komplexität der Delivery Modelle reduziert wird. Zusätzlich sollten spezielle Werkzeuge den Benutzer bei der Modellierung der Delivery Prozesse unterstützen.

Aktuelle CD-Systeme stellen jeweils eigene Modellierungssprachen für Delivery Prozesse bereit. Diese sind nicht standardisiert und besitzen alle unterschiedliche Eigenschaften. Im Gegensatz dazu ist BPMN eine standardisierte Sprache die sich bereits gegen andere Prozess-Modellierungssprachen durchsetzen konnte [Nat11]. Daher stellt sich die Frage, ob sich BPMN zur Modellierung von Delivery Prozessen eignet und ob die Sprache Vorteile gegenüber den proprietären DSLs bietet.

Da BPMN aus dem Bereich BPM stammt, muss zunächst ein Zusammenhang zwischen beiden Themen hergestellt werden. Aus dem vorherigen Kapitel 2 ist bereits bekannt, dass BPM der Verbesserung von Geschäftsprozessen dient. Diese Geschäftsprozesse liefern Kunden bestimmte Services oder Produkte. Damit ein solcher Prozess untersucht werden kann, muss dieser zuerst mit Hilfe einer Modellierungssprache abgebildet werden. Anschließend kann das Modell analysiert oder ausgeführt werden. Eine solche Modellierungssprache stellt BPMN dar.

Ein großer Vorteil von BPMN ist, dass sie für die Modellierung von beliebigen Prozessen verwendet werden kann. Aus diesem Grund wurde BPMN bereits für die Darstellung von Value Streams verwendet [Zor10]. Ein solcher Value Stream umfasst alle Geschäftsprozesse und Aktionen die für die Herstellung und Auslieferung eines Produkts notwendig sind[Kim+16] und kann mit Hilfe von VSM dargestellt werden (vgl. Abschnitt 2.4). VSM und BPMN unterscheiden sich in ihrer Intention. Das Ziel von VSM ist die Identifikation von nicht wertschöpfenden Tätigkeiten, wobei bei BPMN die Identifikation der einzelnen

Aktivitäten und des gesamten Informationsfluss im Vordergrund steht.

Value Streams können mittels VSM dargestellt werden und ein Mapping zwischen den Symbolen von VSM und BPMN ist möglich, da beide Notationen zur Darstellung von Prozessen verwendet werden [Zor10].

Auch die Deployment Pipeline wird als Value Stream oder Teil eines Value Streams betrachtet [HF10; MFS14; Kim+16]. Dies ist möglich, da die Deployment Pipeline den Delivery Prozess abbildet und somit ein Teil des Value Streams ist, der den Kunden neue Software und Features liefert.

Nach [Zor10] müsste es somit möglich sein Delivery Prozesse mit Hilfe von BPMN zu modellieren.

3.1. Forschungsfragen

Diese Arbeit soll verschiedene Forschungsfragen beantworten. Im Folgenden werden diese Fragen vorgestellt.

RQ1: Welche Eigenschaften besitzen aktuelle Sprachen und Systeme für CD und inwieweit unterstützt BPMN diese?

Die Eigenschaften aktueller Sprachen und Systeme geben Aufschluss über die Anforderungen neuer Systeme und Sprachen und stellen daher einen integralen Bestandteil dieser Arbeit dar. Die identifizierten Eigenschaften und konkreten Funktionen können gegen BPMN geprüft werden.

RQ2: Kann BPMN zur Implementierung von CD verwendet werden?

Um diese Frage beantworten zu können, werden die identifizierten Eigenschaften gegen BPMN geprüft. In diesem Kapitel wurde bereits festgestellt, dass auch aktuelle Sprachen nicht alle Anforderungen erfüllen. Daher sind die im Folgenden eingeführten Eigenschaften nicht alle notwendig, sondern sie vereinfachen die Implementierung von CD. Neben der theoretischen Eignung von BPMN soll die Sprache in das SDS JARVIS integriert werden. Dies erlaubt, neben einer theoretischen Bewertung, eine praktische Überprüfung von BPMN als Modellierungssprache für Delivery Prozesse.

RQ3: Existieren BPM Konzepte die sich zur Analyse und Optimierung von Delivery Prozessen eignen?

In diesem Kapitel wurde bereits eine Verbindung von CDund BPMN hergestellt. Da BPMN eine Sprache aus der BPM Domäne ist, stellt sich die Frage, ob BPM weitere nützliche Methoden oder Konzepte für CD bietet. Für diese Forschungsfrage werden die in Kapitel 2 eingeführten Konzepte zur Analyse und Optimierung von Prozessen auf CD angewendet und in das SDS JARVIS integriert.

4. Modellierung von CD mittels BPMN

Inhalt

4.1. Untersuchung der Eigenschaften von CD Sprachen und Systemen 35
4.1.1. Eigenschaften 35
4.1.2. Bewertung 44
4.1.3. Ergebnis 44
4.1.4. Übersicht der Ergebnisse 55
4.2. BPMN 57
4.3. Zusammenfassung 63

In diesem Kapitel werden zunächst wichtige Eigenschaften von Sprachen und Systemen für CD herausgearbeitet. Im Anschluss werden fünf verschiedene Systeme und deren Sprachen auf die erarbeiteten Eigenschaften untersucht. Dadurch können wichtige Defizite existierender Lösungen identifiziert werden.

Im Anschluss werden die festgestellten Eigenschaften aktueller Systeme einer CD Implementierung auf Basis von BPMN gegenübergestellt.

4.1. Untersuchung der Eigenschaften von CD Sprachen und Systemen

Um wichtige Eigenschaften von CD Sprachen und Systemen ermitteln zu können, muss zunächst festgelegt werden, welche Eigenschaften geprüft werden sollen. Diese werden im Folgenden vorgestellt.

4.1.1. Eigenschaften

Die folgenden Eigenschaften dienen als Basis für die Evaluation der Software-Produkte und stellen das Leistungsspektrum dar. Aufgrund der hohen Abhängigkeiten zwischen den Sprachen und Systemen ist eine eindeutige Unterscheidung zwischen Eigenschaften, die sich auf das System beziehen und Eigenschaften, die sich auf die Sprache beziehen, nicht immer möglich. Die Verteilung der Eigenschaften ist in Abbildung 4.1 zusammengefasst.

E.1 Delivery Prozess Domäne

Beschreibung Die Delivery Prozess Domäne ist eine Teildomäne der SDS Domäne und wird in Abbildung 4.2 dargestellt. Basierend auf dem Domänenmodell besteht ein Delivery Prozess aus Phasen (*Stages*), welche wiederum Aktivitäten (*Activities*)

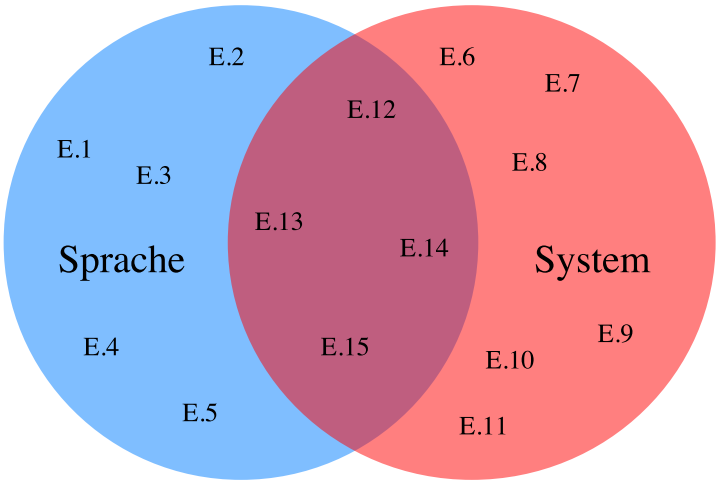

Abbildung 4.1.: Venn-Diagramm: Verteilung der Eigenschaften

beinhalten können. Diese Aktivitäten werden nur unter bestimmten Voraussetzungen (*Execution Preconditions*) ausgeführt und haben *logische* und *funktionale Abhängigkeiten*.

Logische Abhängigkeiten definieren den Zusammenhang zwischen zwei Aktivitäten und beziehen sich auf den Kontrollfluss. Besitzt eine Aktivität A eine logische Abhängigkeit von einer anderen Aktivität B, muss A nach B ausgeführt werden. Eine Erweiterung der logischen Abhängigkeit stellt die funktionale Abhängigkeit dar. Diese bezieht sich auf den Datenfluss und drückt aus, dass eine Aktivität zusätzlich bestimmte Daten einer anderen Aktivität benötigt. Funktionale Abhängigkeiten dienen daher zur Spezifikation der Eingabe-Daten einer Aktivität und indirekt der Ausgabe-Daten der referenzierten Aktivität.

Das Modell legt drei unterschiedliche Aktivitätstypen fest. *Transformation-Aktivitäten* erhalten als Eingabe ein oder mehrere Artefakte und produzieren ein neues Artefakt. Die Artefakte dienen als Eingabe für *Assessment-Aktivitäten*, welche diese untersuchen und einen Bericht erstellen. Das Auswerten der Berichte findet in *Quality-Gate-Aktivitäten* statt. Anhand eines Regelwerkes werden die Berichte ausgewertet und die betroffenen Artefakte können akzeptiert oder abgelehnt werden.

Bewertung Die Sprache muss alle Konzepte zur Abbildung der Domänen-Elemente zur Verfügung stellen.

Quelle Die Delivery Prozess Domäne wurde im Rahmen einer Masterarbeit als Teil der SDS Kerndomäne identifiziert [Dör18]. Diese wurde in der Arbeit zur Implementierung des eigenen SDS JARVIS verwendet.

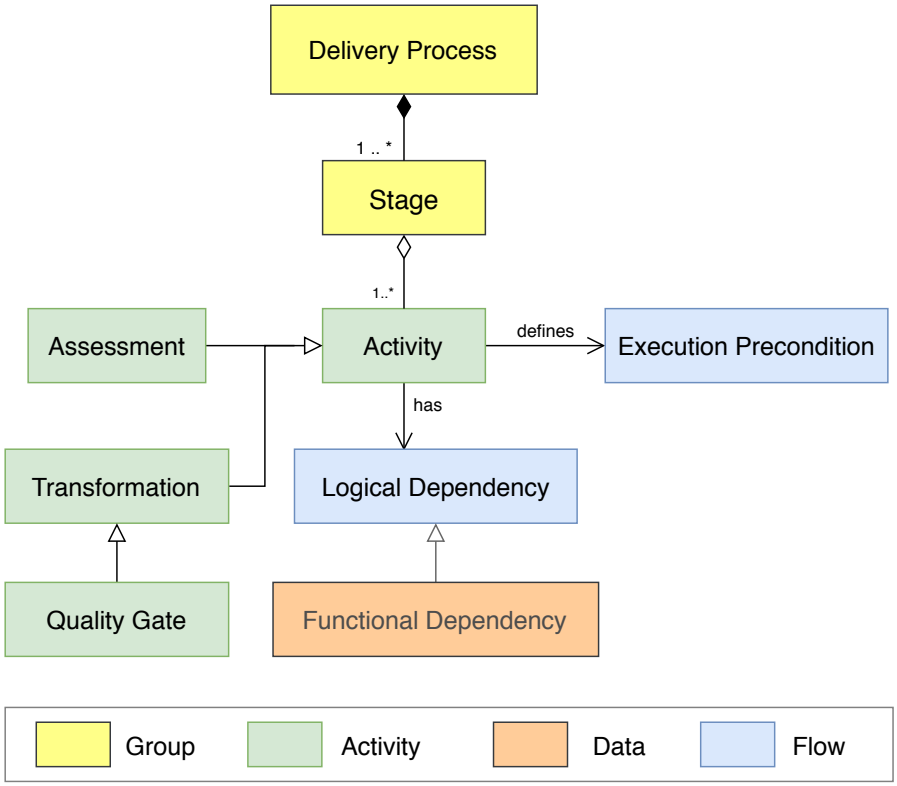

Abbildung 4.2.: Delivery Prozess Domäne

E.2: Alles als Code

Beschreibung Die Beschreibung und Handhabung von Infrastruktur und Delivery Modell als Quellcode führt dazu, dass beides zusammen mit dem Quellcode dem gleichen Software Development Zyklus unterliegen.

Die Anwendung von *Infrastructure as Code (IaC)* unterstützt den *Shift-Left* Ansatz [Wea16], indem das Testen in Produktionsnahen Umgebungen erleichtert wird.

Die damit mögliche Infrastrukturautomatisierung erlaubt das Erstellen von einheitlichen Umgebungen, in denen die Software betrieben und auch getestet wird. Ein verbreitetes System für diesen Zweck ist Docker [Doc].

Bewertung Damit diese Eigenschaft erfüllt ist, muss die Sprache Schnittstellen zur Docker DSL zur Verfügung stellen. Zusätzlich müssen die Delivery Modelle als Quellcode vorliegen.

Quelle Die Eigenschaft wird für die Automatisierung des Delivery Prozess benötigt. Diese ist eines der Basiskonzepte von CD (siehe Abschnitt 2.6.1).

E.3: Wiederverwendbarkeit von Komponenten

Beschreibung Die Wiederverwendung von Komponenten vereinfacht die Implementierung von neuen Deployment Pipelines. Eine Template-Funktionalität ermöglicht das Erstellen von Vorlagen, welche eine Grundstruktur einer Pipeline vorgeben und an das jeweilige Softwareprojekt angepasst werden können. Dies erlaubt es Deployment Pipelines bei ähnlichen Produkten zu standardisieren und damit schneller umzusetzen.

In manchen Situationen ist es nötig, dass eine Pipeline mit unterschiedlichen Parametern ausgeführt wird. Beispielsweise könnte in der Pipeline ein Release-Prozess abgedeckt werden. Da nicht immer nach einem erfolgreichen Durchlauf der Pipeline eine neue Version der Software veröffentlicht werden soll, muss die Pipeline entweder dupliziert und leicht verändert oder ein Delivery Modell muss entsprechend parametrisiert werden können. Aus diesem Grund muss die Sprache Variablen in den Modellen erlauben. Diese werden vor jedem Pipeline-Durchlauf individuell festgelegt.

Bewertung Die Sprache muss die Modellierung von Vorlagen ermöglichen. Zusätzlich muss ein Delivery Modell mit Hilfe von Variablen parametrisierbar sein.

Quelle Die Wiederverwendung von Softwarekomponenten wirkt sich unter anderem positiv auf die Qualität, Zuverlässigkeit und Entwicklungszeit der Komponenten aus [Sam97]. Daher ist sie bereits seit langem ein wichtiges Konzept in der Softwareentwicklung [McI68].

E.4: Ausführung von Code innerhalb einer Pipeline

Beschreibung Da eine Deployment Pipeline sehr speziell auf die betreffende Software zugeschnitten ist, kann es vorkommen, dass die Ausdruckskraft der Sprache nicht ausreicht, um alle Anforderungen umsetzen zu können. Um dennoch den entsprechenden Leistungsumfang abdecken zu können, müssen Skript-Sprachen in die eigentlichen Sprachen eingebettet werden können.

Bewertung Damit diese Eigenschaft erfüllt ist, muss mindestens eine Skript-Sprache in die jeweilige Sprache eingebettet sein.

Quelle Diese Eigenschaft kann keinem Konzept von CD zugeordnet werden. Dennoch findet sie sich in vielen CD Sprachen wieder [Ora; Sof; Tho; Piva; Clo].

E.5: Flusssteuerung und Entscheidungen

Beschreibung Mit Hilfe von Entscheidungen können alternative Flüsse für Release-Prozesse oder Fehlerbehandlungen abgebildet werden. Dabei ist der Ablauf in der Pipeline nicht im Voraus vorgegeben, sondern Entscheidungen werden basierend auf Ereignissen während des Pipeline-Durchlaufs getroffen. Dabei ist besonders wichtig, dass Entscheidungen auf aktuellen Daten aus der Pipeline getroffen werden können. Dadurch können

zum Beispiel Benachrichtigungen erstellt werden, wenn die Code-Abdeckung der Tests unter einen bestimmten Schwellwert fällt.

Im Gegensatz dazu gibt es Entscheidungen, die nur manuell durchgeführt werden können. Bei solchen Entscheidungen handelt es sich meist um manuelle Überprüfungsschritte, wie zum Beispiel das Ignorieren von fehlgeschlagenen Test-Fällen im Ausnahmefall [HF10, S. 15].

Bewertung Damit diese Eigenschaft als erfüllt gilt, muss die Sprache automatisierte und manuelle Entscheidungen abbilden können.

Quelle Die Sprache dient der Modellierung von speziellen Prozessen und daher sollte sie die oben genannten Prozess-typischen Eigenschaften erfüllen (vgl. Abschnitt 2.1).

E.6: Unterstützung von Verwaltungssystemen für Quellcode

Beschreibung Quellcode kann mit unterschiedlichen Systemen versioniert werden. Die bekanntesten sind GIT, *Mercurial (HG)* und *Subversion (SVN)*. Neben der einfachen Versionierung bieten Produkte und Dienste wie GitHub, GitLab und Bitbucket weitere Funktionalitäten, wie das Durchsuchen von Quellcode oder die grafische Darstellung der Quellcode-Historie. Ein wichtiges Unterscheidungsmerkmal ist die Bereitstellung der Verwaltungssysteme. Bei GitHub handelt es sich zum Beispiel um einen Online-Dienst, welcher nicht auf eigenen Servern bereitgestellt wird. GitLab hingegen kann auf beliebigen Servern installiert werden.

Bewertung Ein System muss die oben genannten Versionsverwaltungssysteme unterstützen. Dabei sollten sowohl On-Premise, als auch SaaS Installationen unterstützt werden.

Quelle Diese Eigenschaft kann direkt aus dem Konzept „Versioniere alles in der Versionskontrolle“ abgeleitet werden (vgl. Abschnitt 2.6.2).

E.7: Integration in Systemlandschaft

Beschreibung Die Anbindung des SDS an andere Systeme ist ein wichtiger Inkubator für einen voll-automatisierten Delivery Prozess. Durch eine nahtlose Integration des SDS in die bestehende Systemlandschaft werden manuelle Schritte im Prozess überflüssig und die Cycle Time kann verringert werden (sie Abschnitt 2.6.1).

Eine mögliche Schnittstelle zwischen den Systemen kann über ein etabliertes Protokoll, wie das *Hypertext Transfer Protocol (HTTP)*, implementiert werden. Neben einer solchen generischen Schnittstelle, sollte das SDS Schnittstellen etablierter Systeme direkt implementieren, da nicht alle eine einfach zu nutzende *Application Programming Interface (API)* zur Verfügung stellen.

Bewertung Das SDS muss ein HTTP-API zur Verfügung stellen, mit dem verschiedene Funktionalitäten, wie zum Beispiel das Starten einer Pipeline, ermöglicht wird. Zusätzlich muss es möglich sein, andere HTTP-APIs anzusprechen.

Quelle Diese Eigenschaft ist abgeleitet aus den Konzepten „Automatisierung“ und „Frühes und schnelles Feedback“ (vgl. Abschnitt 2.6.1 und 2.6.2).

E.8 Benutzerunterstützung

Beschreibung Das Modellieren des Delivery Prozesses ist ein wichtiger Arbeitsschritt bei der Entwicklung einer Deployment Pipeline. Daher ist es besonders wichtig, dass ein SDS den Benutzer bei der Entwicklung neuer Modelle unterstützt. Eine solche Unterstützung kann mit Hilfe eines eigenen Modellierungswerkzeugs (Editor) für die Modellierungssprache umgesetzt werden. Dieses Werkzeug kann potenzielle Fehler in Modellen schon während der Entwicklung aufdecken oder sogar verhindern, indem entweder die Modelle direkt validiert werden oder der Editor nur die Verwendung von im aktuellen Kontext erlaubten Elementen zulässt. Eine weitere Möglichkeit der Benutzerunterstützung ist die Einbindung von vordefinierten Elementen der Sprache. Dadurch wird eine korrekte Verwendung der Elemente unterstützt.

Bewertung Das System muss ein Modellierungswerkeug integrieren, mit dem Delivery Modelle entwickelt werden können.

Quelle Die erfolgreiche Anwendung von jeglichen DSLs hängt maßgeblich von den zur Verfügung stehenden Modellierungswerkzeugen ab [PP08].

E.9: Benutzerfreundlichkeit

Beschreibung Auch nachdem eine Pipeline entwickelt wurde, gibt es Funktionalitäten, welche die Ausführung der Pipelines unterstützen. Zum Beispiel vereinfacht eine Wiederholungsfunktion, die einen Pipeline-Durchlauf bei Bedarf wiederholt, den Umgang mit unerwarteten Fehlern.

Ein *Command Line Interface (CLI)*-Tool erleichtert den Umgang mit einem SDS und Validierungen von Modellen können direkt auf dem Rechner des Entwicklers vorgenommen werden. Des Weiteren kann ein solches Tool eine einfache Schnittstelle zum SDS bereitstellen, über die zum Beispiel Pipeline-Modelle hochgeladen werden können. Im Gegensatz zu einem Editor ist ein CLI-Tool deutlich weniger komfortabel und unterstützt den Benutzer nicht direkt bei der Modellierung.

Weitere Merkmale beziehen sich auf die Oberfläche des SDS. Durch die Darstellung einer Pipeline-Historie können Veränderungen in den Durchlaufzeiten der einzelnen Phasen einfacher erkannt werden. Eine Ansicht für laufende Pipeline-Durchläufe gibt Aufschluss über den aktuellen Zustand der Pipeline und Fehler können direkt für den Benutzer in aufbereiteter Form dargestellt werden. Dabei sollte jegliche Ausgabe eines

Pipeline-Durchlaufes archiviert und einfach zugänglich gemacht werden. Dazu zählen auch Test-Dokumentationen, wie sie zum Beispiel von JUnit generiert werden.

Bewertung Das SDS muss eine Wiederholungsfunktion von Pipeline Durchläufen besitzen und ein CLI-Tool bereitstellen. Zusätzlich werden die folgenden Ansichten benötigt:

- Pipeline-Historie
- Pipeline-Durchlauf
- Generierte Test-Dokumentation

Quelle Die Benutzerfreundlichkeit ist wichtig, da die aktuellen Systeme zu kompliziert und komplex sind [LIL17]. Die verschiedenen Ansichten werden für die Nachvollziehbarkeit benötigt [BWZ15].

E.10: Sicherheitsmerkmale

Beschreibung Die aktuellen SDS bieten im Wesentlichen zwei verschiedene sicherheitsrelevante Merkmale. Als erstes kann zwischen den zwei Bereitstellungsmodellen der SDS unterschieden werden. Kann ein System als On-Premise Installation verwendet werden, bietet das den Vorteil, dass das Unternehmen die volle Kontrolle über Datenflüsse hat und somit vorgeschriebene Compliance-Regeln eingehalten werden können. Wohingegen bei der Bereitstellung der Systeme mittels SaaS oft eine Einhaltung der Compliance-Regeln schwer ist, da das System nicht in einer vom Unternehmen überwachten Umgebung läuft.

Ein zweites Sicherheitsmerkmal stellt die sichere Verwahrung von Zugangsdaten dar. Innerhalb einer Deployment Pipeline werden verschiedene Services und Systeme angesprochen, die vor unbefugtem Zugriff geschützt werden müssen. Daher ist es wichtig, dass das SDS eine Funktionalität besitzt, mit der Zugangsdaten, wie Passwörter oder SSH-Schlüssel, sicher hinterlegt und in der Pipeline verwendet werden können. Dies bietet eine sicherere Alternative zur direkten Vorhaltung, zum Beispiel in Klartext, von sicherheitsrelevanten Daten innerhalb der Pipeline.

Bewertung Es müssen folgende Eigenschaften erfüllt sein:

- On-Premise Installation
- Sichere Aufbewahrung von Zugangsdaten

Quelle Sicherheit ist für jedes Softwaresystem ein wichtiges Thema. Durch die Anbindung an verschiedene Systeme, wie zum Beispiel der Versionsverwaltung, muss die Deployment Pipeline entsprechend abgesichert werden [NCS17].

E.11: Unterstützung von Continuous Integration

Beschreibung Das kontinuierliche Integrieren von Änderungen ist ein wichtiger Bestandteil von CD und muss von den betrachteten Systemen unterstützt werden.

Bewertung Für eine vollständige Erfüllung dieser Eigenschaft muss das System die folgenden Funktionen besitzen:

- Quellcode von einer Versionsverwaltung abfragen und kompilieren
- Test-Fälle automatisiert ausführen

Quelle CI ist ein integraler Bestandteil von CD (vgl. Abschnitt 2.6.2).

E.12: Artefakt-Verwaltung

Beschreibung Um das Konzept *Baue ein Artefakt einmalig* umzusetzen, müssen Artefakte zwischen den einzelnen Phasen transferiert werden (vgl. Abschnitt 2.6.2). Zusätzlich ist die Archivierung von Artefakten eine weitere wichtige Funktionalität, um sicher zu stellen, dass nur erfolgreich überprüfte Artefakte veröffentlicht werden.

Bewertung Artefakte müssen eindeutig identifizier- und archivierbar sein.

Quelle Diese Eigenschaft lässt sich auf das Konzept „Baue ein Artefakt einmalig“ zurückführen (vgl. Abschnitt 2.6.2).

E.13: Unterstützung von Deployment-Strategien

Beschreibung Um Software immer in produktiv-ähnlichen Umgebungen testen und gleich ausliefern zu können, müssen die in der Produktiv-Umgebung angewendeten Deployment-Strategien in der Deployment Pipeline modellierbar sein. Für die Umsetzung der Strategien ist es wichtig, dass diese automatisiert ausgeführt werden.

Bewertung Die Sprache muss Konzepte zur Abbildung von Deployment-Strategien bereitstellen. Diese müssen vom System entsprechend umgesetzt werden können.

Quelle Die Eigenschaft lässt sich direkt aus dem CD Konzept „Verwende Deployment-Strategien“ ableiten (vgl. Abschnitt 2.6.2).

E.14: Frühes und schnelles Feedback

Beschreibung Für frühes Feedback ist es wichtig, dass ein Pipeline-Durchlauf direkt nach dem einchecken von Änderungen am Quellcode angestoßen wird. Dazu muss zum einen die Sprache Konstrukte für die Modellierung von *Build-Triggern* zur Verfügung

stellen und zum anderen muss das System auf Änderungen in der Versionskontrolle reagieren können.

Das Ergebnis eines Pipeline-Durchlaufes sollte für schnelles Feedback möglichst direkt den beteiligten Personen mitgeteilt werden. Daher ist die Verwendung von im Unternehmen etablierten Kommunikationswerkzeugen besonders wichtig. Für eine generische Anbindung an beliebige Benachrichtigungssysteme ist es sinnvoll, dass die Systeme mittels *Webhooks* Nachrichten publizieren können.

Für schnelles Feedback ist es nötig *Timeouts* abbilden zu können. Diese erlauben es Pipeline Durchläufe frühzeitig abzubrechen und dem Benutzer entsprechendes Feedback zu geben.

Bewertung Es werden folgende Benachrichtigungssysteme geprüft:

- E-Mail
- Chat-Programme: HipChat, Slack, Jabber, Mattermost
- Generische Webhooks

Zusätzlich wird geprüft, ob in der Sprache Timeouts modelliert werden können.

Quelle Diese Eigenschaft stellt die Anwendbarkeit des Konzepts „Frühes und schnelles Feedback“ sicher (vgl. Abschnitt 2.6.2).

E.15: Parallelisierter Delivery-Prozess

Beschreibung Um schnell das Ergebnis eines Pipeline-Durchlaufs zu erhalten, sollten voneinander unabhängige Phasen oder Aktivitäten in der Pipeline parallel ausgeführt werden können. Bei besonders langen Pipeline Durchläufen kann dadurch die Cycle Time signifikant verkürzt werden.

Oft treten bei der Parallelisierung Seiteneffekte auf. Um diese zu verhindern, können Docker-Container eingesetzt werden. Einzelne Phasen oder Aktivitäten werden in einzelnen Containern ausgeführt (*Execution Isolation*). Alternativ können sogenannte *Remote-Agents* eingesetzt werden. Dies bedeutet, dass die einzelnen Schritte auf unterschiedlichen Servern ausgeführt werden. Dadurch können verschiedene Umgebungen simuliert werden.

Bewertung Die Sprache muss das parallele Modellieren von Phasen und Aktivitäten ermöglichen. Zusätzlich muss das Konzept *Execution Isolation* mit Hilfe von *Remote-Agents* oder Docker-Containern unterstützt werden.

Quelle *Execution Isolation* ist ein allgemein bekanntes Konzept in der Softwareentwicklung. Das parallele Ausführen von Phasen oder Aktivitäten ermöglicht die Beschleunigung des Prozesses und somit können Fehler schneller erkannt werden [Mug15].

4.1.2. Bewertung

Die Methode legt drei Einstufungen für die Bewertung der Eigenschaften fest:

- Gute Unterstützung (+)
- Partielle Unterstützung (o)
- Unzureichende Unterstützung (-)

Wird eine Eigenschaft vollständig erfüllt, wird die Bewertung (+) vergeben. Ist eine der bewerteten Funktionalitäten nur teilweise umgesetzt oder nur aufwändig abzubilden, wird sie als partiell unterstützt (o) bewertet. Unzureichende Unterstützung (-) wird vergeben, wenn es nicht möglich ist die Funktionalität abzubilden.

4.1.3. Ergebnis

Die Systeme und Sprachen wurden unter Berücksichtigung der vorgestellten Eigenschaften untersucht. Die Ergebnisse werden im Folgenden vorgestellt.

Allgemein kann festgehalten werden, dass alle der folgenden untersuchten Sprachen und Systeme eine gute Unterstützung für CI bieten (E.14).

Jenkins

Die Ergebnisse der Analyse für Jenkins sind in Tabelle 4.1 zusammengefasst [Sof].

Eigenschaft	+	o	-
E.1: Delivery Prozess Domäne		o	
E.2: Alles als Code	+		
E.3: Wiederverwendbarkeit von Komponenten		o	
E.4: Ausführung von Code innerhalb einer Pipeline	+		
E.5: Flusssteuerung und Entscheidungen		o	
E.6: Unterstützung von Verwaltungssystemen für Quellcode	+		
E.7: Integration in Systemlandschaft	+		
E.8: Benutzerunterstützung			-
E.9: Benutzerfreundlichkeit		o	
E.10: Sicherheitsmerkmale	+		
E.11: Unterstützung von Continuous Integration	+		
E.12: Artefakt-Verwaltung	+		
E.13: Unterstützung von Deployment-Strategien		o	
E.14: Frühes und schnelles Feedback	+		
E.15: Parallelisierter Delivery-Prozess	+		

Tabelle 4.1.: Ergebnis der Analyse für Jenkins

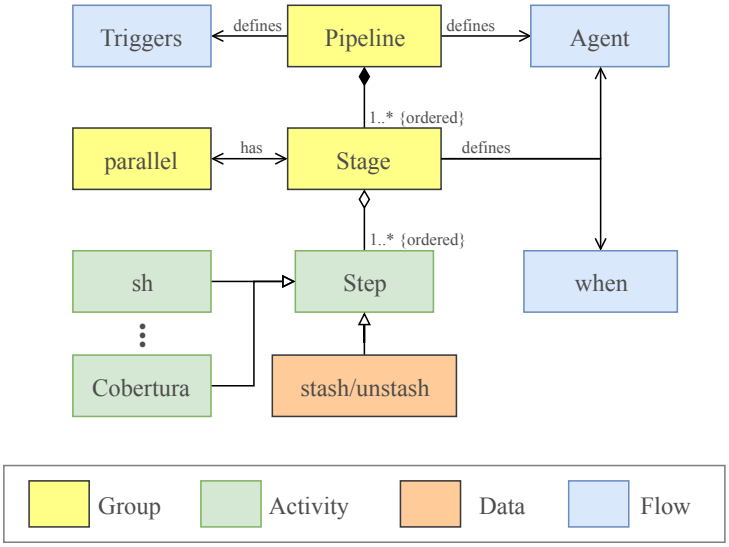

Abbildung 4.3.: Jenkins-Elemente der Domäne

Die Jenkins Pipeline DSL realisiert teilweise die Elemente der Delivery Prozess Domäne (vgl. Abbildung 4.3). Anstatt des Begriffs *Activity* verwendet die DSL *Steps*. *Execution Preconditions* werden mittels *when*-Elementen definiert. Die Pipeline DSL unterscheidet bei Aktivitäten nicht zwischen *Assessments*, *Transformations* und *Quality Gates*. Dennoch können die verschiedenen Schritte den drei Subtypen zugeordnet werden. Logische Abhängigkeiten werden implizit durch die Anordnung der Phasen und über das *parallel*-Element im Modell festgelegt. Die explizite Modellierung von funktionalen Abhängigkeiten ist nicht möglich, da Daten zwischen einzelnen Phasen oder Schritten über gemeinsame Arbeitsbereiche zur Verfügung gestellt werden. Mittels den Operationen *stash* und *unstash* ist ein Zugriff auf solche Arbeitsbereiche möglich.

Die Delivery Modelle können im sogenannten *Jenkinsfile* abgelegt und versioniert werden. Des Weiteren kann jede Phase in einem eigenen Docker-Container ausgeführt werden.

Jenkins stellt keine wirkliche Template-Funktionalität bereit, sondern verwendet das Konzept der *Shared Libraries*. Diese werden in Groovy implementiert und ermöglichen das Aufrufen von Methoden und Funktionen innerhalb der Pipeline. Zusätzlich können Variablen zur Parametrisierung verwendet werden.

Neben der Verwendung von Shared Libraries kann beliebiger Groovy-Code direkt in die Pipeline-Definition eingebettet werden. Alternativ ist die Einbettung von Shell-Befehlen möglich.

Automatisierte Entscheidungen können mit dem Schlüsselwort *when* modelliert werden. Das automatisierte Ignorieren von Schritten ist mittels *Try-Catch*-Block möglich. Dabei können allerdings nur Fehler abgefangen werden, die innerhalb eines Schrittes auftreten. Eine feingranulare Differenzierung ist nicht möglich. Entscheidungen, die auf aktuellen

Daten basieren, sind nur möglich, wenn diese Daten als Variablen in der Pipeline bereit gestellt werden. Manuelle Entscheidungen werden explizit mit einem *input*-Schritt modelliert. Dieser erlaubt sowohl einfache Ja/Nein Abfragen, als auch die Eingabe von beliebigen Texten. Das manuelle Ignorieren von Testfällen ist nicht direkt in der Sprache möglich.

Jenkins kann alle gängigen Versionsverwaltungssysteme ansprechen. Durch seine offene Plugin-Architektur können viele verschiedene Systeme integriert werden. Beispielweise kann Jenkins bestehende JIRA-Tickets schließen oder Neue anlegen. Andere Systeme können Jenkins über eine HTTP-API ansprechen und steuern.

Ein Pipeline Durchlauf kann mittels Retry-Schalter erneut angestoßen werden. Die Validierung einer Pipeline ist nur in Verbindung mit der Oberfläche von Jenkins möglich, da ein eigenständiges CLI-Tool oder ein Editor fehlt. Die Oberfläche von Jenkins stellt alle wichtigen Informationen bereit und Zugangsdaten können zentral hinterlegt und anschließend in der Pipeline referenziert werden. Des Weiteren ist eine Installation von Jenkins auf eigenen Servern möglich.

Artefakte werden mit Hilfe von *stash* und *unstash* Befehlen zwischen Stages transferiert. Zur eindeutigen Identifikation werden *fingerprints* eingesetzt. Damit Artefakte nach einem Pipeline-Durchlauf über die Oberfläche heruntergeladen werden können, müssen diese mit dem *archiveArtifacts*-Schritt archiviert werden.

Jenkins integriert keine direkte Unterstützung für Deployment-Strategien, sondern diese müssen entweder in einem Skript oder mittels Plugins umgesetzt werden.

Für frühes und schnelles Feedback kann ein Pipeline-Durchlauf, sowohl zeitgesteuert als auch über die HTTP-API angestoßen werden. Das zeitgesteuerte Abbrechen eines Durchlaufes kann mittels *Timeouts* festgelegt werden. Zusätzlich ist das Starten einer Pipeline über die Oberfläche möglich. Das Publizieren von Benachrichtigungen kann über alle gängigen Plattformen erfolgen (vgl. Abschnitt 4.1.1).

Das parallele Ausführen von Stages kann modelliert werden. Zusätzlich kann die Ausführung innerhalb eines Docker-Containers oder auf einem anderen Jenkins-Knoten erfolgen.

GoCD

Die Ergebnisse der Analyse für GoCD sind in Tabelle 4.2 zusammengefasst [Tho]. GoCD verwendet ein Domänenmodell, das im Vergleich zu Abbildung 4.2, eingeschränkter ist (vgl. Abbildung 4.4). Das Modell führt neben Phasen und Aufgaben ein weiteres Aggregat ein: den Job. Innerhalb eines Jobs können Aufgaben definiert werden, die immer parallel ausgeführt werden. Dies führt dazu, dass logische Abhängigkeiten implizit über die Semantik der Phasen und Jobs gegeben sind. Eine feingranulare explizite Definition von logischen Abhängigkeiten ist nicht möglich. Tasks können funktionale Abhängigkeiten indirekt über eine *Fetch Artifact*-Aufgabe definieren. Dabei muss jedoch beachtet werden, dass keine direkte Referenz auf einen anderen Task angegeben wird, sondern nur Artefakt-Referenzen. Diese Artefakte müssen vor der Ausführung der jeweiligen Aufgabe von einem vorherigen Job publiziert worden sein, da nur Jobs Artefakte veröffentlichen können. Des Weiteren können nur die einfachen *Execution Preconditions passed, failed, any* verwendet

Eigenschaft	+	o	-
E.1: Delivery Prozess Domäne		o	
E.2: Alles als Code		o	
E.3: Wiederverwendbarkeit von Komponenten	+		
E.4: Ausführung von Code innerhalb einer Pipeline	+		
E.5: Flusssteuerung und Entscheidungen			-
E.6: Unterstützung von Verwaltungssystemen für Quellcode	+		
E.7: Integration in Systemlandschaft		o	
E.8: Benutzerunterstützung			-
E.9: Benutzerfreundlichkeit		o	
E.10: Sicherheitsmerkmale	+		
E.11: Unterstützung von Continuous Integration	+		
E.12: Artefakt-Verwaltung	+		
E.13: Unterstützung von Deployment-Strategien		o	
E.14: Frühes und schnelles Feedback			-
E.15: Parallelisierter Delivery-Prozess	+		

Tabelle 4.2.: Ergebnis der Analyse für GoCD

werden. Auch bei GoCD gibt es keine Unterscheidung von Aufgaben in *Transformations, Assessments* und *Quality-Gates*.

GoCD ermöglicht nicht die direkte Anbindung an Docker. Über die Erweiterungsschnittstelle *Elastic Agents* kann mit Hilfe von Plugins diese Funktionalität dennoch realisiert werden. Solche Plugins werden über XML-Dateien konfiguriert und können daher versioniert werden. Dennoch ist eine zentrale Versionierung mitsamt dem Quellcode nicht möglich. Delivery Modelle können hingegen direkt versioniert werden.

Damit eine Wiederverwendung von Delivery Modellen möglich ist, können Templates und Variablen verwendet werden.

In GoCD ist die Einbettung von Shell-Skripten in die Pipeline möglich. Weitere Skript-Sprachen können nicht direkt verwendet werden.

Entscheidungen sind nur sehr eingeschränkt möglich. Automatisiert kann nur auf Fehler reagiert werden und das Ignorieren von fehlgeschlagenen Test-Fällen ist, sowohl automatisiert, als auch manuell, nicht möglich. Manuelle Ja/Nein-Abfragen sind hingegen über *approval*-Schritte realisierbar.

Alle gängigen Versionsverwaltungssysteme können verwendet werden. Für die Integration in die Systemlandschaft stellt GoCD eine HTTP-API zur Verfügung. GoCD selbst kann jedoch von Haus aus keine Ticket-Systeme ansprechen.

Das erneute Starten einer Pipeline ist über die Oberfläche möglich. Ein eigenständiger Editor oder ein CLI-Tool fehlen für die, als Plugin realisierten, Modellierungssprachen. Alternativ bietet GoCD eine Oberfläche zum Konfigurieren von Pipelines an. Dabei wird aber dem Prinzip PaC widersprochen, da die erstellten Modelle nicht versioniert werden können. Für die auf YAML basierende Sprache gibt es aktuell nur einen einfachen

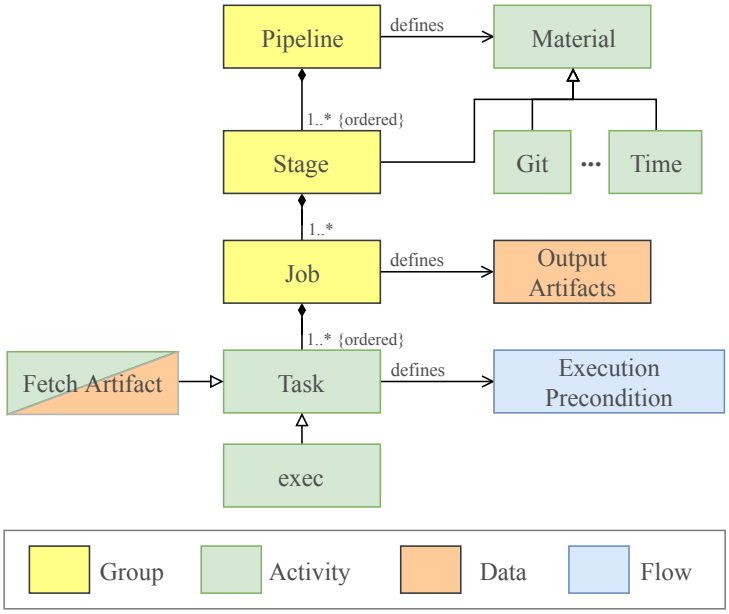

Abbildung 4.4.: GoCD-Elemente der Domäne

YAML-Validator, der allerdings nur die Syntax der Modelle prüft. Die Oberfläche von GoCD stellt alle wichtigen Informationen bereit.

GoCD kann als On-Premise Installation auf eigenen Servern bereit gestellt werden und Zugangsdaten können zentral hinterlegt werden.

Artefakte müssen bei GoCD innerhalb des Projekt-Arbeitsbereichs liegen, um identifiziert und zwischen den Stages transferiert werden zu können. Die Identifikation erfolgt über relative Pfade im Arbeitsbereich des Projekts. Als Artefakt konfigurierte Dateien oder Ordner können über die Oberfläche heruntergeladen werden.

Deployment-Strategien werden nicht direkt von GoCD unterstützt, sondern müssen durch Erweiterungen realisiert werden.

Pipeline Durchläufe können zeitgesteuert oder mit Hilfe der HTTP-API angestoßen werden. Benachrichtigungen sind nur beschränkt möglich. Integrationen mit Jabber oder Mattermost existieren aktuell nicht und eine generische Anbindung an beliebige Systeme fehlt komplett [Ben16]. Die Verwendung von Timeouts ist hingegen möglich.

Die Ausführung von parallelen Schritten oder ganzen Stages in Docker-Containern ist nicht möglich. Als Alternative bietet GoCD jedoch Remote-Agents.

Codeship

Die Ergebnisse der Analyse für Codeship sind in Tabelle 4.3 zusammengefasst [Clo]. Im Domänenmodell von Codeship fehlt das Konzept der Phasen (vgl. Abbildung 4.5). Anstatt Phasen können optional parallele oder sequenzielle *Group-Steps* festgelegt werden. Diese beinhalten wiederum *Steps*. Codeship unterscheidet dabei zwischen *Push-* und *Run-Steps*. Ein Push-Step ermöglicht das Publizieren von Docker-Images und Run-Steps

Eigenschaft	+	o	-
E.1: Delivery Prozess Domäne			-
E.2: Alles als Code	+		
E.3: Wiederverwendbarkeit von Komponenten			-
E.4: Ausführung von Code innerhalb einer Pipeline	+		
E.5: Flusssteuerung und Entscheidungen			-
E.6: Unterstützung von Verwaltungssystemen für Quellcode		o	
E.7: Integration in Systemlandschaft			-
E.8: Benutzerunterstützung			-
E.9: Benutzerfreundlichkeit		o	
E.10: Sicherheitsmerkmale			-
E.11: Unterstützung von Continuous Integration	+		
E.12: Artefakt-Verwaltung		o	
E.13: Unterstützung von Deployment-Strategien		o	
E.14: Frühes und schnelles Feedback		o	
E.15: Parallelisierter Delivery-Prozess	+		

Tabelle 4.3.: Ergebnis der Analyse für Codeship Pro

dienen der Ausführung von Shell-Kommandos. Es wird nicht zwischen *Transformations, Assessments* und *Quality-Gates* unterschieden. Des Weiteren kann die Ausführung von einzelnen Steps nur über Branches bzw. Tags aus der Versionsverwaltung eingeschränkt werden. Weitere *Execution Preconditions* sind nicht möglich. Logische Abhängigkeiten und funktionale Abhängigkeiten können nicht explizit definiert werden. Der Austausch von Artefakten ist nur über einen gemeinsamen Arbeitsbereich möglich.

Codeship Pro bietet offiziell *native Docker-Unterstützung* und Delivery Modelle werden als YAML-Datei versioniert.

Die Wiederverwendbarkeit von Komponenten ist bei Codeship nur sehr eingeschränkt möglich. Eine Template-Funktionalität fehlt komplett, die Parametrisierung ist mit Hilfe von Variablen möglich.

Die Einbettung von Shell-Befehlen in Delivery Modelle ist möglich.

Entscheidungen können in Codeship nicht modelliert werden. Ein weiterer Nachteil von Codeship ist, dass keine lokal installierten Versionsverwaltungssystemen eingebunden werden können, sondern nur Systeme, wie GitHub oder Bitbucket, die Online als SaaS zur Verfügung gestellt werden.

Die HTTP-API ermöglicht eine rudimentäre Interaktion mit Codeship. Ticket-Systeme können nicht über Codeship angesprochen werden.

Das Starten von Pipeline Durchläufen ist über die Oberfläche möglich und eine Validierung eines Delivery Modells ist indirekt über das Ausführen der Pipeline mittels einem CLI-Tool möglich. Codeship stellt keinen eigenen Editor zum Modellieren von Pipelines zur Verfügung. Die Oberfläche von Codeship bietet alle wichtigen Funktionalitäten, bis auf die Darstellung von JUnit-Test Reports in aufbereiteter Form [Rad16].

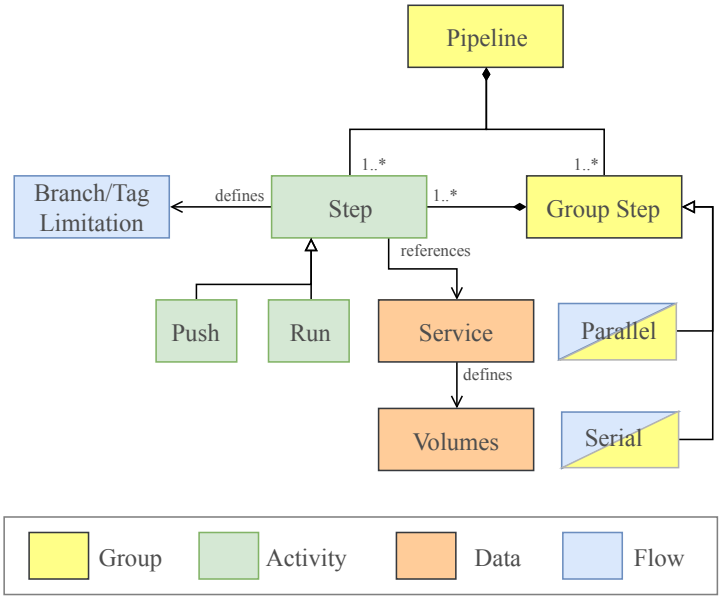

Abbildung 4.5.: Codeship-Elemente der Domäne

Codeship Pro wird nur als SaaS bereit gestellt. Zugangsdaten können jedoch zentral hinterlegt werden. Problematisch bei dieser Kombination ist, dass die Daten nicht auf eigenen Servern gespeichert werden und somit Compliance-Richtlinien schwer eingehalten werden können.

Im Allgemeinen kann auf Artefakte nur innerhalb eines Pipeline-Durchlaufes zugegriffen werden, da sie im Anschluss gelöscht werden. Docker-Images können jedoch an Docker-Registries übertragen werden. Zwischen den Stages können Artefakte über gemeinsame Ordner im Dateisystem transferiert werden. Die Identifikation ist dabei über Ordner- und Dateinamen möglich.

Die Bereitstellung der Artefakte mit Hilfe von Deployment-Strategien ist nur möglich, wenn die Deployment-Umgebung diese direkt unterstützen (vgl. [Shi17]). Dazu bietet Codeship verschiedene Integrationen mit Services, wie zum Beispiel Amazon EC2 [Ama], Heroku [Her] oder Azure Container Service [Mic].

Die zeitgesteuerte Ausführung von Pipeline Durchläufen ist nicht direkt möglich, sondern muss von außerhalb über die HTTP-API angestoßen werden [Raf16]. Das Publizieren von Benachrichtigungen ist über die geforderten Systeme oder über eine generische Schnittstelle möglich. In Codeship können keine expliziten Timeouts für Schritte festgelegt werden, sondern es existiert nur ein impliziter Timeout von 5 Stunden.

Parallelität kann mit Codeship modelliert werden und *Execution Isolation* wird mittels Docker-Containern erreicht.

Concourse

Die Ergebnisse der Analyse für Concourse sind in Tabelle 4.4 zusammengefasst [Piva]. In Abbildung 4.6 sind die relevanten Elemente von Concourse abgebildet. Anstatt dem

Eigenschaft	+	o	-
E.1: Delivery Prozess Domäne		o	
E.2: Alles als Code	+		
E.3: Wiederverwendbarkeit von Komponenten		o	
E.4: Ausführung von Code innerhalb einer Pipeline	+		
E.5: Flusssteuerung und Entscheidungen			-
E.6: Unterstützung von Verwaltungssystemen für Quellcode		o	
E.7: Integration in Systemlandschaft		o	
E.8: Benutzerunterstützung			-
E.9: Benutzerfreundlichkeit		o	
E.10: Sicherheitsmerkmale	+		
E.11: Unterstützung von Continuous Integration	+		
E.12: Artefakt-Verwaltung		o	
E.13: Unterstützung von Deployment-Strategien		o	
E.14: Frühes und schnelles Feedback		o	
E.15: Parallelisierter Delivery-Prozess	+		

Tabelle 4.4.: Ergebnis der Analyse für Concourse

Konzept der Phasen verwendet Concourse sogenannte *Jobs*. Diese werden standardmäßig parallel ausgeführt. Die sequenzielle Ausführung von Jobs ist über die Angabe einer gemeinsamen *serial_group* möglich. Des Weiteren können *Steps* in *Put-, Get-Operationen* und *Tasks* unterteilt werden. Die *Put-, Get-Operationen* ermöglichen den Zugriff auf Artefakte. Eine Unterteilung der Steps in *Transformations, Assessments und Quality-Gates* ist ebenfalls nicht möglich. Die Modellierung von einfachen logischen Abhängigkeiten ist nicht möglich, da sich Concourse auf funktionale Abhängigkeiten beschränkt. Diese werden über *Input-* und *Output-Parameter* definiert. Zusätzlich können die *Execution Preconditions on_success, on_failure, on_abort* und *ensure* verwendet werden. Weitere Bedingungen können nicht modelliert werden.

Concourse erlaubt, sowohl das Erstellen von neuen Docker-Images, als auch das Starten von Docker-Containern innerhalb einer Pipeline. Das Versionieren von Delivery Modellen als YAML-Dateien ist möglich.

In den Delivery Modellen von Concourse können Variablen zur Parametrisierung verwendet werden. Eine Template-Funktionalität ist in Concourse nicht direkt implementiert, wurde aber von einem Drittanbieter [Sta17] als Zusatzsoftware unter MIT Lizenz zur freien Verfügung gestellt.

Das Ausführen von Shell-Befehlen oder Skripten ist in Pipelines möglich. Andere Skriptsprachen können nur verwendet werden, wenn das Skript in einem Docker-Container

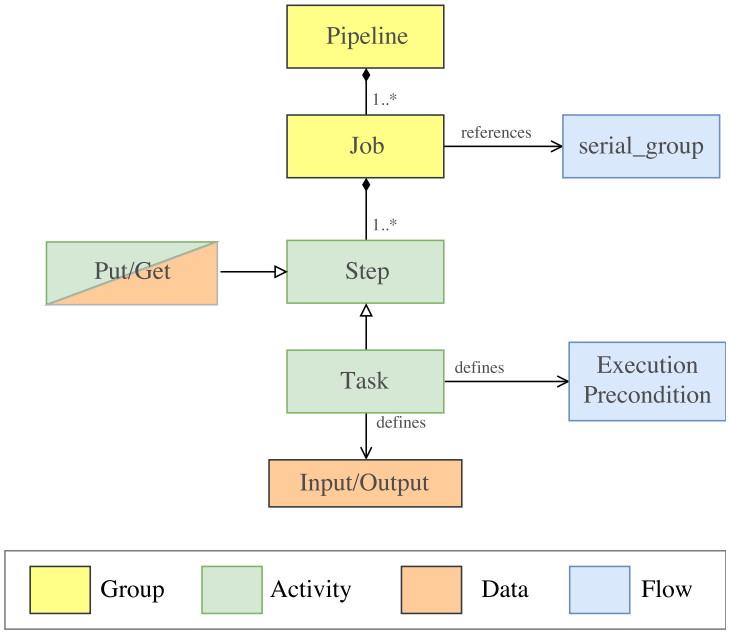

Abbildung 4.6.: Concourse-Elemente der Domäne

läuft, der alle benötigten Laufzeitkomponenten der Sprache bereit stellt.

Das Modellieren von Entscheidungen ist mit Concourse nur sehr rudimentär möglich. Als automatisierte Entscheidungen stehen nur, wie bereits oben genannt, *on_success, on_failure, on_abort* und *ensure* zur Verfügung. Eine Auswertung von Laufzeitvariablen ist nicht möglich. Manuelle Ja/Nein-Abfragen können nicht modelliert werden. Als Alternative bietet Concourse jedoch manuelle Aufgaben. Diese blockieren einen Pipeline-Durchlauf und der Benutzer muss diese über die Oberfläche bestätigen, damit der Durchlauf fortgesetzt wird. Eine manuelle Zweigauswahl, sowie das manuelle oder automatisierte Ignorieren von fehlgeschlagenen Test-Fällen, ist nicht möglich.

Concourse unterstützt die Versionsverwaltungssysteme Git und HG, bietet aber keine native Subversion Unterstützung. Um Subversion verwenden zu können, kann eine eigene Anbindung (vgl. [Rob16]) integriert werden. Anbindungen an Mehrwertdienste, wie GitHub, GitLab oder Bitbucket gehen nicht über die Funktionalitäten der darunter liegenden Versionsverwaltungssystem hinaus.

Zur Integration in die Systemlandschaft stellt Concourse eine HTTP-API zur Verfügung. Diese ist weitestgehend undokumentiert und somit nur bedingt nutzbar [Sti16]. Über eine Erweiterung können fremde HTTP-API angesprochen werden [Blo16]. Zusätzlich existieren verschiedene Erweiterungen um Concourse mit anderen Systemen, wie zum Beispiel Ticket-Systemen, zu verbinden.

Pipeline Durchläufe können manuell über die Oberfläche angestoßen werden. Das CLI-Tool *Fly* ermöglicht das Validieren und lokale Ausführen von Pipelines. Ein eigenes Modellierungstool, wie einen Editor, wird nicht bereit gestellt. Die Oberfläche von Concourse bietet keine allgemeine Pipeline-Historie, sondern bietet für jede Aktivität

innerhalb der Pipeline eine eigene Historie. Test-Dokumentationen von JUnit können nicht in aufbereiteter Form dargestellt werden.

Concourse kann auf lokalen Servern installiert werden und ermöglicht die zentrale Verwaltung von Zugangsdaten.

Artefakte werden in Concourse als Eingaben oder Ausgaben von Aufgaben gesehen. Concourse stellt sicher, dass Artefakte in den Aufgaben zur Verfügung stehen, bei denen sie benötigt werden. Das Archivieren von Artefakten funktioniert durch Erweiterungen und wird nicht direkt von der Sprache unterstützt.

Deployment-Strategien werden nicht von Concourse unterstützt und müssen durch Erweiterungen implementiert werden.

Das Starten eines Pipeline Durchlaufes ist sowohl zeitgesteuert, wie auch über die HTTP-API, möglich. Die Sprache ermöglicht das Modellieren von zeitgesteuerten Abbrüchen von einzelnen Schritten. Concourse kann Benachrichtigungen nicht über Jabber oder Mattermost versenden, jedoch über HipChat und E-Mail.

Durch die implizite Flusssteuerung parallelisiert Concourse automatisch unabhängige Aufgaben und verwendet dazu Remote-Agents und Docker-Container.

Wercker

Die Ergebnisse der Analyse für Wercker sind in Tabelle 4.5 zusammengefasst [Ora].

Eigenschaft	+	o	-
E.1: Delivery Prozess Domäne		o	
E.2: Alles als Code		o	
E.3: Wiederverwendbarkeit von Komponenten			-
E.4: Ausführung von Code innerhalb einer Pipeline	+		
E.5: Flusssteuerung und Entscheidungen			-
E.6: Unterstützung von Verwaltungssystemen für Quellcode		o	
E.7: Integration in Systemlandschaft			-
E.8: Benutzerunterstützung		o	
E.9: Benutzerfreundlichkeit		o	
E.10: Sicherheitsmerkmale			-
E.11: Unterstützung von Continuous Integration	+		
E.12: Artefakt-Verwaltung		o	
E.13: Unterstützung von Deployment-Strategien		o	
E.14: Frühes und schnelles Feedback		o	
E.15: Parallelisierter Delivery-Prozess	+		

Tabelle 4.5.: Ergebnis der Analyse für Wercker

Wercker verwendet zwei unterschiedliche Ebenen bei der Modellierung der Pipelines. Auf der obersten Ebene werden verschiedene Pipelines grafisch zu einem Workflow orchestriert. Auf dieser Ebene erfolgt das Modellieren von logischen Abhängigkeiten und es können

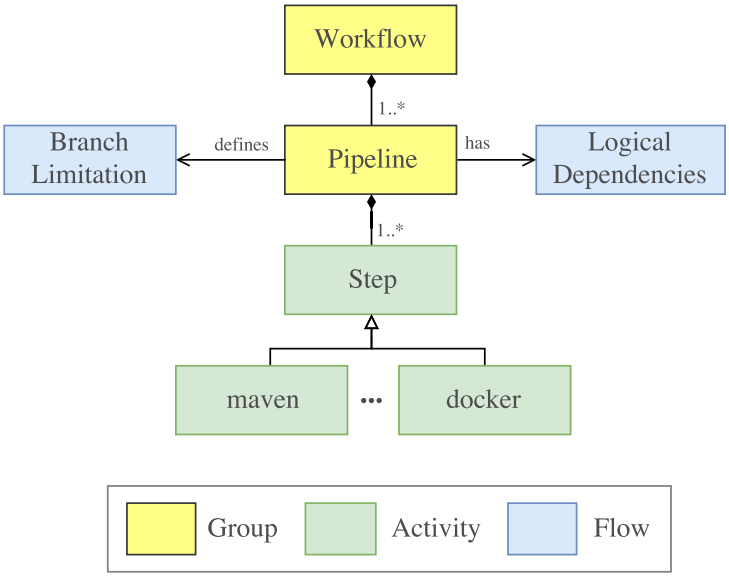

Abbildung 4.7.: Wercker-Elemente der Domäne

sowohl sequenzielle, als auch parallele Abläufe modelliert werden. Wercker ermöglicht nur die Ausführungsbeschränkung von Pipelines anhand von Branch-Namen. Pipelines beinhalten einzelne sequenzielle Steps, die beliebige Aufgaben implementieren können. Es wird keine funktionalen Abhängigkeiten modelliert und der Transfer von Daten muss mittels Steps realisiert werden.

Die Infrastruktur wird bei Wercker mit Hilfe von *Services* modelliert. Services werden über ein Docker-Image definiert und zur Ausführungszeit wird ein entsprechender Docker-Container gestartet. Schritte in einer Pipeline werden auch in einem Docker-Container ausgeführt. Wie bereits in Abschnitt 2.6.4 beschrieben, unterscheidet Wercker zwischen *Workflows* und *Pipelines*. Problematisch dabei ist, dass *Workflows* nicht als Code abgebildet werden, sondern nur über die Oberfläche konfiguriert werden können.

Bei Wercker gibt es keine Template-Funktionalität. Die Verwendung von Variablen in Delivery Modellen ist hingegen möglich.

Um den Funktionsumfang der Sprache zu erweitern können Shell-Befehle ausgeführt werden. Andere Skript-Sprachen müssen über einen entsprechenden Docker-Container zur Verfügung gestellt werden.

Eine Steuerung des Flusses basierend auf manuellen oder automatisierten Entscheidungen ist nicht möglich.

Wie Codeship Pro, wird auch Wercker nur als SaaS zur Verfügung gestellt und kann daher nicht auf lokal installierte Versionsverwaltungssysteme zugreifen. Neben GitHub sind nur die SaaS Varianten von Bitbucket und GitLab nutzbar.

Wercker kann über eine HTTP-API angesprochen werden. Eine Integration in die Systemlandschaft ist nur sehr eingeschränkt möglich, da auf interne Systeme meist nicht von außen zugegriffen werden kann.

Das erneute Starten von Pipeline Durchläufen ist über die Oberfläche möglich. Pipelines können nicht vollständig in einem eigenen Editor entwickelt werden, da nur die Orchestrierung der einzelnen Pipelines über eine Oberfläche möglich ist. Die lokale Ausführung von Pipelines ist über das CLI-Tool *Wercker CLI* möglich. Es bildet vollständig die SaaS Umgebung von Wercker ab und die Validierung von Delivery Modellen ist indirekt über die lokale Ausführung der Pipeline möglich. Die Oberfläche bietet alle wichtigen Informationen, bis auf die Darstellung von Test-Dokumentationen.

Wie auch Codeship, wird Wercker nur als SaaS bereit gestellt und kann nicht auf lokalen Servern installiert werden. Zugangsdaten können bei Wercker in Variablen ausgelagert werden.

Das Konzept der Artefakte wird von Wercker nicht verwendet. In jedem Schritt der Pipeline kann auf den Projekt-Arbeitsbereich zugegriffen werden und somit auch auf die erzeugten Artefakte. Das Speichern von Docker-Images ist über den Schritt *internal/docker-push* in einer Pipeline möglich.

Deployment-Strategien werden in Wercker nicht implementiert, können aber über eigene Erweiterungen realisiert werden.

Die Ausführung von Piplines ist in Wercker nicht zeitgesteuert möglich, sondern nur über die HTTP-API oder über die Oberfläche. Benachrichtigungen können nicht über Jabber und Mattermost versendet werden, jedoch über Slack oder per E-Mail. Die Verwendung von Timeouts ist für einzelne Schritte oder für den gesamten Pipeline Durchlauf möglich.

Die einzelnen Schritte innerhalb einer Pipeline können sequenziell oder parallel erfolgen. Optional kann jeder Schritt in einem anderen Docker-Container ausgeführt werden. Das Konzept der Remote-Agents wird nicht verwendet.

4.1.4. Übersicht der Ergebnisse

Alle Ergebnisse der Analyse sind in Tabelle 4.6 zusammengefasst. Die Analyse der Sprachen und Systeme hat gezeigt, dass viele der Eigenschaften nur teilweise oder gar nicht erfüllt werden. Bei der Untersuchung der Sprachen wurde festgestellt, dass die Delivery Prozess Domäne nicht vollständig mit den Sprachen abbildbar ist, da vor allem das Modellieren von Abhängigkeiten nicht möglich ist. Zusätzlich wird das Konzept der *Execution Preconditions* in den Sprachen nur sehr beschränkt angewendet, da oft nur einfache und vordefinierte Bedingungen verwendet werden können. Alle der untersuchten Sprachen verwenden als kleinste Arbeitseinheit Schritte oder Aufgaben. Es wird aber nicht zwischen *Transformations, Assessments* und *Quality-Gates* differenziert.

Die Versionierung von Delivery Modellen und die Verwendung von Docker wird von allen Sprachen teilweise bis gut unterstützt. Wohingegen die Wiederverwendbarkeit von Komponenten im Allgemeinen nur beschränkt möglich ist. Im Gegensatz dazu ermöglicht jede der untersuchten Sprachen die Ausführung von Quellcode als einzelnen Schritt in der Pipeline. Dies zeigt, dass die Sprachen nicht ausreichend Ausdrucksstark sind, um allen Anforderungen gerecht zu werden und daher auf weitere Sprachen zurückgegriffen werden muss.

Eigenschaft	1	2	3	4	5
E.1: Delivery Prozess Domäne	o	o	-	o	o
E.2: Alles als Code	+	o	+	+	o
E.3: Wiederverwendbarkeit von Komponenten	o	+	-	o	-
E.4: Ausführung von Code	+	+	+	+	+
E.5: Flusssteuerung und Entscheidungen	o	-	-	-	-
E.6: Unterstützung von Versionsverwaltungen für Quellcode	+	+	o	o	o
E.7: Integration in Systemlandschaft	+	o	-	o	-
E.8: Benutzerunterstützung	-	-	-	-	o
E.9: Benutzerfreundlichkeit	o	o	o	o	o
E.10: Sicherheitsmerkmale	+	+	-	+	-
E.11: Unterstützung von Continuous Integration	+	+	+	+	+
E.12: Artefakt-Verwaltung	+	+	o	o	o
E.13: Unterstützung von Deployment-Strategien	o	o	o	o	o
E.14: Frühes und schnelles Feedback	+	-	o	o	o
E.15: Parallelisierter Delivery-Prozess	+	+	+	+	+

Tabelle 4.6.: Zusammenfassung der Analyse-Ergebnisse: Jenkins (1), GoCD (2), Codeship (3), Concourse (4), Wercker (5)

Die Modellierung und Steuerung des Flusses ist bei allen Sprachen, bis auf die Sprache von Jenkins, nicht ausreichend möglich. Bei dieser Eigenschaft haben die untersuchten Sprachen am schlechtesten abgeschnitten.

Die Analyse der Systeme hat gezeigt, dass alle eine mittelmäßige bis gute Unterstützung von Versionsverwaltungssystemen besitzt. Die Systeme bieten nur teilweise Schnittstellen zur Integration in die bestehende Systemlandschaft. Keines der betrachteten Systeme bietet ein vollständiges Modellierungswerkzeug um Delivery Prozesse zu modellieren. Allgemein ist die Benutzerfreundlichkeit bei allen nur beschränkt gegeben. Im Gegensatz dazu bieten die meisten Systeme entsprechende Sicherheitsmerkmale, wie die sichere Verwahrung von Passwörtern.

Neben den Eigenschaften, die eindeutig Sprachen oder Systemen zugeordnet werden können, wurden weitere *Hybride*-Eigenschaften untersucht. Der explizite Umgang mit Artefakten ist teilweise oder vollständig möglich. Deployment-Strategien können nur teilweise oder mit eingebetteten Sprachen realisiert werden und auch das Publizieren von Feedback ist nur auf wenigen unterschiedlichen Wegen möglich. Im Gegensatz dazu bieten alle Sprachen und Systeme die Möglichkeit Pipeline-Komponenten zu parallelisieren.

Bei der Analyse der Sprachen ist vor allem herausgekommen, dass das *Steuern des Flusses und die Modellierung von Entscheidungen (E.5)* eine Eigenschaft ist, die den meisten aktuellen Sprachen fehlt. Die Systeme haben bei der Eigenschaft *Benutzerunterstützung (E.8)* am schlechtesten abgeschnitten, da kein Modellierungswerkzeug zur Verfügung gestellt wird.

4.2. BPMN

Nachdem wichtige Eigenschaften von Sprachen für CD abgeleitet und existierende Sprachen auf deren Erfüllung untersucht wurden, wird nun überprüft, ob BPMN die gleichen Eigenschaften erfüllt, wie die existierenden Sprachen, oder sogar weitere Vorteile mit sich bringt.

Zusätzlich werden die relevanten Bestandteile der BPMN-Domäne identifiziert, damit später eine passende Abbildung von BPMN-Elementen auf die Delivery Prozess Domäne definiert werden kann.

Im Folgenden wird überprüft, ob die abgeleiteten Eigenschaften aus Abschnitt 4.1.1 von BPMN erfüllt werden.

Delivery Prozess Domäne (E.1)

Abbildung 4.8 zeigt einen Ausschnitt aus der vollständigen Definition der BPMN Domäne.

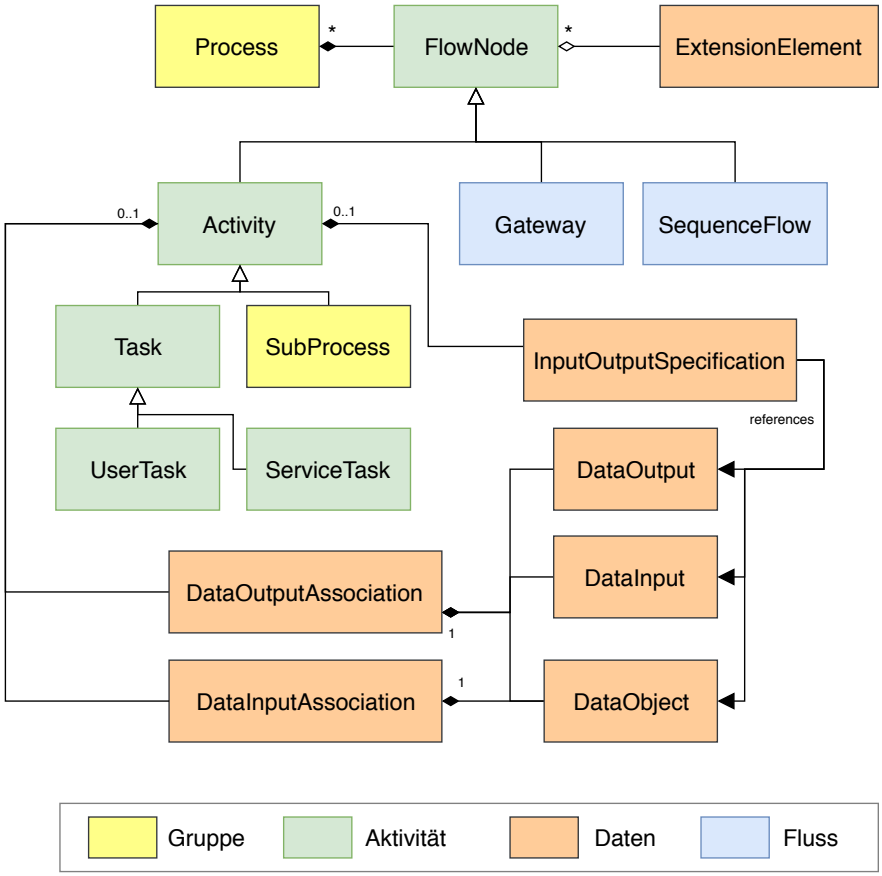

Abbildung 4.8.: BPMN Domäne (vereinfacht)

BPMN bietet viele verschiedene Elemente für die Modellierung von Prozessen. Das zentrale Element stellt dabei der Prozess dar. Dieser kann beliebig viele Flussknoten-Elemente (FlowNodes) beinhalten. Diese können wiederum beliebig viele Erweiterungselemente besitzen. Die Erweiterungselemente dienen der Anreicherung von Elementen um beliebige Datenstrukturen und müssen über *XML Schema Definitions (XSDs)* definiert werden. Das Einbinden von XSDs ist über ein Import-Element möglich. Ein mögliches Anwendungsgebiet ist die Modellierung von speziellen Eingabe- und Ausgabe-Spezifikationen von Aktivitäten. Dadurch können beliebige Datenmodelle und funktionale Abhängigkeiten abgebildet werden.

Flussknoten können entweder Aktivitäten (*Activities*), Gateways oder Sequenzflüsse (*SequenceFlows*) sein. Aufgaben, wie Benutzer-Aufgaben (*UserTask*), Service-Aufgaben (*ServiceTask*) oder auch Subprozesse (*SubProcess*) sind Aktivitäten. Eine Service-Aufgabe kann als *Transformation, Assessment* oder *Quality-Gate* verstanden werden, da die drei Aktivitäten automatisierte Funktionalitäten bereitstellen. Eine genauere Differenzierung zwischen den Typen kann mit Hilfe von zusätzlichen Eigenschaften, die an den Aktivitäten festgelegt werden, erreicht werden.

Zur Spezifikation der Ein- und Ausgaben von Aktivitäten können *InputOutputSpecifications* zusammen mit *DataOutputAssociations* bzw. *DataInputAssociations* verwendet werden. Mit Hilfe der InputOutputSpecification wird das Interface der Aktivität definiert, das bedeutet alle Daten die die Aktivität vor der Ausführung benötigt und alle Daten, die die Aktivität nach der Ausführung zurückliefert. Die Assoziationen stellen eine weitere Möglichkeit für die Modellierung von funktionalen Abhängigkeiten dar und spezifizieren, woher die Daten kommen und wohin diese weitergeleitet werden. Das Modellieren der Daten kann über XSDs erfolgen.

Logische Abhängigkeiten können mittels Sequenzflüssen modelliert werden und Gateways ermöglichen die Modellierung von Bedingungen.

Eine detaillierte Abbildung wird später in Kapitel 5.1.1 gegeben.

Alles als Code (E.2)

BPMN bietet keine direkte Möglichkeit zur Beschreibung der Infrastruktur, da sie sich auf die Beschreibung von Prozessen fokussiert. Aus diesem Grund muss die Modellierung der Infrastruktur extern erfolgen. Dadurch wird eine Trennung von Zuständigkeiten (Separation of Concerns) erreicht und es können für beide Anwendungsgebiete spezialisierte Sprachen verwendet werden. Dieser Ansatz wird auch in den meisten existierenden Sprachen verwendet. Zur Beschreibung der Infrastruktur wird meist Docker verwendet. Dazu bietet zum Beispiel Jenkins die Möglichkeit Docker-Images für einzelne Phasen zu spezifizieren. Mit BPMN kann ein solcher Ansatz leicht realisiert werden, indem entsprechende Werte an Elementen, wie Aktivitäten und Subprozessen, konfiguriert werden. Die Implementierung der Aktivitäten kann diese Parameter auswerten und entsprechende Aktionen ausführen.

Neben Infrastruktur-Modellen müssen auch die Delivery Modelle versioniert werden. Dies ist seit BPMN Version 2.0 direkt möglich, da diese ein allgemeines XML-Format zur

```
<sequenceFlow>
   <conditionExpression xsi:type="tFormalExpression">
     ${test == 'foo'}
   </conditionExpression>
 </sequenceFlow>
```

Quelltext 4.1: sequenceFlowExpressionExample.groovy

Speicherung von BPMN Modellen bietet. Dadurch können modellierte Pipelines direkt in einer Versionskontrolle hinterlegt werden.

Wiederverwendbarkeit von Komponenten (E.3)

Die zentralen Komponenten von BPMN sind Aktivitäten. Diese stehen für eine Aktion oder Aufgabe, die im Prozess durchgeführt werden soll. Neben direkt im Modell eingebetteten Implementierungen der Aktivitäten über zum Beispiel Skript-Aufgaben, können auch externe Services über Service-Aufgaben angesprochen werden. Dadurch können Funktionalitäten aus der Pipeline ausgelagert und wiederverwendet werden.

Zusätzlich können andere Prozesse mittels Aufruf-Aktivität aufgerufen werden. Dadurch können Teilprozesse von mehreren Prozessen verwendet werden.

Des Weiteren ist es möglich Variablen zur Parametrisierung von Prozessen zu verwenden. Wichtig dabei ist, dass BPMN zwar Daten-Elemente zur Verfügung stellt, aber kein eingebautes Datenmodell und keine Abfragesprache („Expression Language“) definiert [Obj11, S. 203]. Die Spezifikation schlägt XSD als Daten-Beschreibungssprache und XPath als Abfragesprache vor, überlässt die Entscheidung aber den BPMS-Entwicklern. Dadurch können theoretisch beliebige Sprachen verwendet werden. Camunda BPM zum Beispiel erlaubt die Verwendung von JSON, XML, Java Objekten und primitiven Datentypen. Als Abfragesprache kann die *Unified Expression Language (EL)* verwendet werden (vgl. Quelltext 4.1).

Ausführung von Code innerhalb einer Pipeline (E.4)

Zur Ausführung von Skripten, bietet BPMN unter anderem Skript-Aufgaben. Dabei sind aus Sicht der Sprache beliebige Skript-Sprachen möglich, da der Quellcode direkt in die Aufgabe eingebettet wird und das BPMS die Ausführung übernimmt. Camunda BPM erlaubt zum Beispiel die Ausführung von JavaScript und Groovy (vgl. Quelltext in Anhang A.1).

Neben Skript-Aufgaben können Skripte auch für Bedingte-Sequenzflüsse verwendet werden. Dies erlaubt zum Beispiel die Auswertung von Variablen zur Laufzeit.

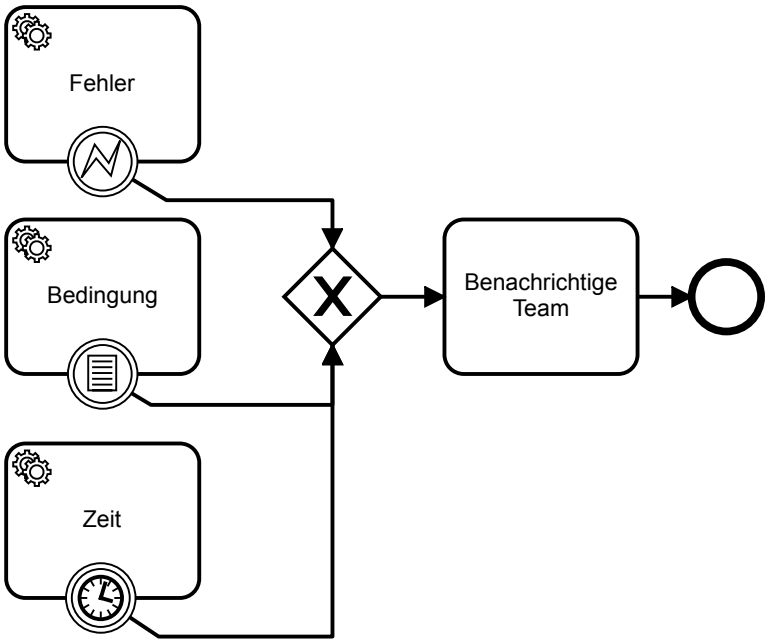

Abbildung 4.9.: Angeheftete Zwischenereignisse

Flusssteuerung und Entscheidungen (E.5)

Ein großer Vorteil von BPMN gegenüber der bereits untersuchten CD Sprachen ist, dass viele verschiedene Elemente zur Flusssteuerung und zur Modellierung von Entscheidungen verwendet werden können. Dies umfasst unter anderem die unterschiedlichen Ereignis-Typen, Gateways, bedingte Sequenzflüsse, Manuelle-Aufgaben und Geschäftsregel-Aufgaben.

Vor allem angeheftete Zwischenereignisse sind hilfreich, da sie zur Modellierung von Ereignissen innerhalb Aktivitäten dienen. In Abbildung 4.9 werden drei unterschiedliche Anwendungsmöglichkeiten gezeigt.

Das Fehler-Ereignis ermöglicht das Modellieren von Fehlern innerhalb von Aktivitäten. Tritt ein Fehler in der Aktivität auf, wird das Ereignis ausgelöst und es kann im Prozess darauf reagiert werden, indem zum Beispiel eine Benachrichtigung versendet wird. Tritt ein Fehler-Ereignis auf wird die beteiligte Aktivität sofort beendet.

Ein allgemeineres Element ist das angeheftete Bedingung-Ereignis. Dieses wertet einen Ausdruck aus, der auch Variablen beinhalten kann. Neben der unterbrechenden Variante, die die Aktivität abbricht, kann das Ereignis auch in der nicht-unterbrechenden Variante verwendet werden. Das bedeutet, dass wenn das Ereignis ausgelöst wird, ein zusätzliches Token erzeugt wird und die Aktivität nicht abgebrochen wird (vgl. Abschnitt 2.2.3). Dadurch können zum Beispiel bei langlaufenden Aktivitäten Benachrichtigungen während der Laufzeit der Aktivität versendet werden.

Ein weiteres wichtiges Ereignis ist das Zeit-Ereignis. Dieses ermöglicht zum Beispiel das Modellieren von Timeouts. Als Startereignis kann es auch die zeitgesteuerte Ausführung

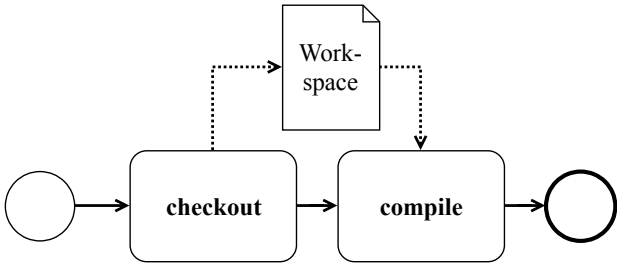

Abbildung 4.10.: Artefakt-Transfer zwischen zwei Aktivitäten

der Pipeline auslösen.

Neben Ereignissen können Gateways zur Steuerung des Flusses verwendet werden (vgl. Abschnitt 2.2.3). Das datenbasierte exklusive Gateway ermöglicht das Verzweigen von Pfaden. Dadurch können zum Beispiel Phasen oder Aktivitäten in Abhängigkeit von Variablen ausgeführt werden. Die Bedingungen werden direkt an den ausgehenden Sequenzflüssen mit Hilfe einer Abfragesprache, wie EL, definiert. Parallelität kann mit Hilfe von parallelen Gateways realisiert werden. Sie ermöglichen die parallele Ausführung von Phasen oder Aktivitäten in einer Pipeline.

Manuelle Entscheidungen können mit einer Kombination aus Benutzer-Aufgaben und datenbasierten exklusiven Gateways modelliert werden. In der Benutzer-Aufgabe können sowohl Ja/Nein-Abfragen, als auch komplexere Formular-Daten abgefragt werden. Basierend auf diesen Daten kann mittels eines darauf folgenden Gateways der Pfad gewählt werden. Dies ist zum Beispiel hilfreich wenn eine Release-Phase nur manuell angestoßen werden soll.

Auf Regeln basierende automatische Entscheidungen können mit einer ähnlichen Kombination modelliert werden. Dazu wird eine Geschäftsregel-Aufgabe vor einem datenbasierten exklusiven Gateway geschaltet. Auch in diesem Fall spezifiziert BPMN nicht die konkrete Implementierung. Camunda BPM ermöglicht zum Beispiel als Implementierung DMN, Java Klassen oder Ausdrücke.

Artefakt-Verwaltung (E.12)

Für die Verwaltung von Artefakten können in BPMN zwei unterschiedliche Ansätze verwendet werden. Wie bereits in Abschnitt 2.2.3 eingeführt, gibt es die Elemente Daten und Datenablagen. Diese ermöglichen das explizite Modellieren von Artefakten und Artefakt-Speichern. Durch das Daten-Element kann der Artefakt-Transfer zwischen mehreren Aktivitäten modelliert werden, indem das Daten-Element mit den entsprechenden Aktivitäten durch Assoziationen verbunden wird (vgl. Abbildung 4.10). Eine ausgehende Assoziation an einer Aktivität zeigt, dass die Aktivität ein Artefakt erzeugt und eine eingehende Assoziation bedeutet, dass die Aktivität das Artefakt als Eingabe benötigt. Die Datenablage kann als Artefakt-Speicher verwendet werden (vgl. Abbildung 4.11). Eine von einer Aktivität ausgehende Assoziation zu einer Datenablage bedeutet, dass die Aktivität ein Artefakt produziert und in den Speicher ablegt. Von der Datenablage

ausgehende Assoziationen bedeuten, dass die entsprechenden Aktivitäten Daten aus der Datenablage ausließt.

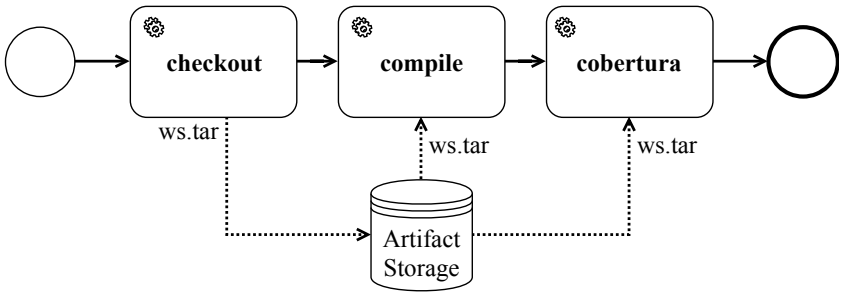

Abbildung 4.11.: Artefakt-Storage in BPMN

Neben dieser grafischen Darstellung von Daten erlaubt BPMN, wie bereits zuvor beschrieben, die Verwendung von Variablen bei der Parametrisierung von Aktivitäten. Theoretisch können Aktivitäten Artefakte oder Artefakt-Referenzen erzeugen und diese in Variablen speichern. Dadurch können Variablen zur Parametrisierung von Aktivitäten verwendet werden. Dieses Konzept ist aber nicht Teil der BPMN Spezifikation, sondern muss von den BPMS-Lieferanten implementiert werden.

Unterstützung von Deployment-Strategien (E.13)

Die BPMN bietet nur allgemeine Elemente zur Modellierung und überlässt die konkreten Implementierungen den BPMS-Lieferanten. Aus diesem Grund gibt es keine vordefinierten Aktivitäten zur Modellierung von Deployment-Strategien. Da eine Deployment-Strategie jedoch auch einen Prozess darstellt, kann er als Teilprozess des Delivery Prozesses angesehen und somit einfach in ein bestehendes Delivery Modell eingebunden werden. Ein großer Vorteil dieses Ansatzes ist, dass ein in BPMN modellierter Deployment-Prozess sowohl in der Pipeline, als auch in Produktiv-Umgebungen verwendet werden kann.

Frühes und schnelles Feedback (E.14)

Damit möglichst schnell Feedback geliefert werden kann, muss eine Pipeline automatisiert gestartet werden können. BPMN Pipelines werden durch Start-Ereignisse gestartet. Neben Blanko-Startereignissen können Nachricht-, Zeit- oder Bedingung-Ereignisse zum Starten eines Prozesses verwendet werden. Das Blanko-Ereignis ermöglicht das manuelle Starten und Nachrichten-Ereignisse das automatisierte Starten der Pipeline über eine Schnittstelle. Als Nachricht könnte zum Beispiel eine Änderung in der Versionskontrolle angesehen werden. Des Weiteren kann die Pipeline mittels Zeit-Ereignis zu beliebigen Zeitpunkten oder in Intervallen ausgeführt werden.

Benachrichtigungen können als Service-Aufgaben modelliert werden. Dadurch können in BPMN beliebige Benachrichtigungsdienste modelliert werden.

Das Modellieren von Timeouts ist, wie bereits zuvor beschrieben, über angeheftete Zeit-Ereignisse möglich.

Parallelisierter Delivery-Prozess (E.15)

Die BPMN stellt parallele Gateways zur Modellierung von Parallelität zur Verfügung. Dadurch können sowohl parallele Phasen, als auch parallele Aktivitäten modelliert werden. Dabei muss beachtet werden, dass Flüsse, die durch ein paralleles Gateway auseinander geführt wurden, auch wieder durch ein weiteres paralleles Gateway zusammengeführt werden müssen.

4.3. Zusammenfassung

In diesem Kapitel wurden wichtige Eigenschaften von CD Sprachen und Systemen abgeleitet (RQ-1). Es wurden fünf Sprachen und Systeme unter Berücksichtigung dieser Eigenschaften untersucht, wobei festgestellt wurde, dass aktuelle Sprachen kaum die Möglichkeiten bieten, um Entscheidungen zu modellieren. Bei der Betrachtung der Systeme wurde die schlechte Benutzerunterstützung während der Modellierung als größtes Problem identifiziert.

Anschließend wurde BPMN als allgemeine Prozess-Modellierungssprache auf diese Delivery Prozess spezifischen Eigenschaften geprüft und es wurde festgestellt, dass BPMN alle Eigenschaften erfüllt. Die in BPMN zur Verfügung stehenden Elemente ermöglichen die Abbildung aller Konzepte der Delivery Prozess Domäne. Zur Modellierung von funktionalen Abhängigkeiten können entweder Assoziationen in Verbindung mit *InputOutputSpecifications* und *DataObjects* oder *ExtensionElements* verwendet werden. Wichtig bei der Verwendung der *DataObjects* ist, dass diese erst über XSDs definiert werden müssen und dadurch eventuell mehr Aufwand bei der Modellerstellung entsteht. Des Weiteren ist dieses Konstrukt komplex, da neben der grafischen Modellierung der Assoziationen und Daten-Objekte eine zusätzliche Komplexität durch die *InputOutputSpecifications* erreicht wird.

Zusammenfassend kann gesagt werden, dass BPMN sich sehr gut für die Modellierung von Delivery Prozessen eignet, da BPMN alle untersuchten Eigenschaften einer CD Sprache besitzt. Anzumerken ist jedoch, dass die Sprache sehr komplex ist und sehr viele unterschiedliche Elemente für die Modellierung von Prozessen bereitstellt. Daher ist ein unterstützendes Modellierungswerkzeug besonders wichtig.

5. Implementierung von CD mittels BPMN

Inhalt

5.1. BPMN als Delivery Prozess Modellierungssprache 65
5.1.1. Abbildung der Delivery Prozess Domäne 65
5.1.2. Datenmodellierung und Konfiguration der Aktivitäten 68
5.2. Konzept zur Integration in JARVIS 68
5.2.1. Überblick 68
5.2.2. Komponenten-Sicht 70
5.2.3. Prozess-Sicht 75
5.3. BPM Konzepte zur Analyse und Optimierung von Delivery Prozessen . . 84
5.4. Zusammenfassung 85

In Kapitel 4 wurden Eigenschaften von SDSs und den dazu gehörigen Sprachen ermittelt. Dieses Kapitel führt verschiedene Konzepte ein, damit BPMN diese Eigenschaften erfüllt. Dazu wird die Sprache in das SDS JARVIS integriert. Zusätzlich wird die Analyse und Optimierung von BPMN Pipelines betrachtet.

5.1. BPMN als Delivery Prozess Modellierungssprache

Im Folgenden werden zwei Konzepte zur Abbildung der Delivery Prozess Domäne mit Hilfe von BPMN vorgestellt. Danach wird gezeigt, wie BPMN das bereits herausgearbeitete Leistungsspektrum erfüllen kann.

5.1.1. Abbildung der Delivery Prozess Domäne

Das Metamodell von BPMN definiert die Entitäten der BPMN-Domäne. Damit Delivery Prozesse mittels BPMN modelliert werden können, muss eine Abbildung zwischen BPMN und der Delivery Prozess Domäne konstruiert werden. Aufgrund der hohen Flexibilität von BPMN können unterschiedliche Abbildungen definiert werden und eine partielle Abbildung ist ausreichend. Als erstes wird eine allgemeine Abbildung vorgestellt, die nicht die funktionalen Abhängigkeiten betrachtet. Die verschiedenen Varianten zur Abbildung von funktionalen Abhängigkeiten werden im Anschluss erläutert und diskutiert.

Allgemeine Abbildung

BPMN lässt sich, wie in Abbildung 5.1 dargestellt, zur Abbildung der Delivery Prozess Domäne verwenden. Das Diagramm zeigt einen Prozess mit einem Subprozess.

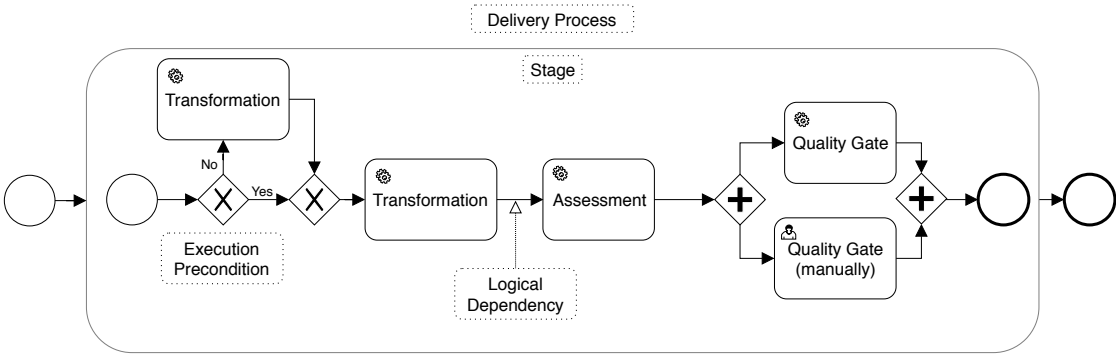

Abbildung 5.1.: BPMN Delivery Prozess Domäne ohne funktionale Abhängigkeiten

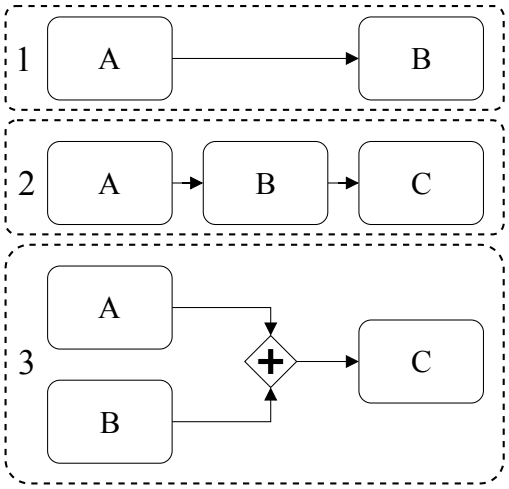

Abbildung 5.2.: Unterschiedliche Situationen von logischen Abhängigkeiten

Dabei kann die Prozess-Entität direkt auf einen Delivery Prozess abgebildet werden. Der Subprozess stellt eine Stage des Delivery Prozesses dar. Es ist sowohl eine sequenzielle, als auch eine parallele Anordnung von Subprozessen möglich.

Transformations, Assessments und Quality-Gates können mit Hilfe von Aufgaben dargestellt werden. Transformations und Assessments sind immer automatisierte Aufgaben und werden daher als Service-Aufgaben modelliert. Da die Bewertung von Assessments durch automatisierte oder manuelle Quality-Gates erfolgen kann, werden diese entweder als Service-Aufgabe oder als Benutzer-Aufgabe abgebildet.

Zur Modellierung von Execution Preconditions können datenbasierte exklusive Gateways verwendet werden. Dadurch können beliebige Bedingungen definiert werden. Das parallele Gateway kann zur Modellierung von Parallelität verwendet werden.

Die Modellierung von logischen Abhängigkeiten wird in BPMN mit Hilfe von Sequenzflüssen realisiert. Dabei kann zwischen drei unterschiedlichen Situationen unterschieden werden (vgl. Abbildung 5.2). Die einfachste Situation ist wenn die Aktivität B direkt

über einen eingehenden Sequenzfluss mit Aktivität A verbunden ist (1). Es kann direkt eine logische Abhängigkeit von A für B gefolgert werden. Des Weiteren ist die logische Abhängigkeit transitiv. Dadurch ist in der zweiten Situation nicht nur B abhängig von A und C abhängig von B, sondern auch C von A. Durch die Modellierung von Parallelität ergibt sich eine weitere Situation (3). Werden die Aktionen A und B parallel ausgeführt und im Anschluss durch ein paralleles Gateway zusammengeführt, besitzt die darauf folgende Aktivität C eine logische Abhängigkeit von A und B.

Funktionale Abhängigkeiten

Funktionale Abhängigkeiten können auf unterschiedliche Arten und Weisen modelliert werden. Jede Aktivität besitzt in BPMN eine InputOutputSpecification. Diese bietet die Möglichkeit die benötigten Eingaben und die erzeugten Ausgaben einer Aktivität zu spezifizieren. In der InputOutputSpecification werden beliebig viele DataInput-Objekte und DataOutput-Elemente definiert. Die Referenzierung erfolgt über das *inputSet* bzw. *outputSet*. Dadurch können Daten-Objekte in verschiedenen Aktivitäten referenziert werden.

Die DataInput- bzw. DataOutput-Elemente werden innerhalb der InputOutputSpecifiaction angelegt. Damit zur Laufzeit entsprechende Daten zur Verfügung stehen, müssen DataInputAssociations für Eingaben und DataOutputAssociations für Ausgaben spezifiziert werden. Diese sind nicht Teil der InputOutputSpecification, sondern werden allgemein für die Aktivität definiert. Sie können als Quelle ein DataInput- bzw. DataOutput-Element und als Ziel ein passendes Element aus der InputOutputSpecification referenzieren. Als Quelle kann auch ein globales DataObject verwendet werden. Stimmen die Typen von Quelle und Ziel nicht über ein, muss eine Transformationsregel angegeben werden.

Eine weitere Möglichkeit um funktionale Abhängigkeiten zu modellieren, ist die Verwendung der *ExtensionElements* [Obj11, S. 64]. Diese können zu jeder Aktivität hinzugefügt werden. Da sie von der BPMN-Spezifikation nicht weiter definiert werden, sind beliebige Erweiterungen möglich. Die Erweiterungen sind Process Engine spezifisch und werden von diesen ausgewertet. Dabei werden unbekannte ExtensionElements ignoriert. Camunda BPM verwendet ExtensionElements um das Interface von Aktivitäten zu modellieren. InputOutputSpecification-Elemente, Daten-Objekte und Assoziationen werden von Camunda BPM nicht ausgewertet. Das Interface einer Aktivität wird bei Camunda BPM mit Hilfe des InputOutput-Elements angegeben. Dieses beinhaltet beliebig viele InputParameter bzw. OutputParameter. Ein Parameter ist ein einfaches *Schlüssel-Werte-Paar*. Der Schlüssel ist immer eine Zeichenkette und der Wert muss von den Typen *Text, Script, Map* oder *List* sein. Wichtig ist, dass ein Wert entweder vordefiniert oder zur Laufzeit bestimmt werden kann. Damit eine Zuweisung zur Laufzeit möglich ist, müssen entsprechende Java-Objekte programmatisch über eine Schnittstelle der Engine hinzugefügt werden. Mittels EL können dann diese Objekte zur Parametrisierung der Elemente verwendet werden.

5.1.2. Datenmodellierung und Konfiguration der Aktivitäten

In der Regel benötigt eine Aktivität verschiedene Eingabe-Daten. Neben einer fixen Konfiguration von Werten, die zur Entwicklungszeit feststehen, müssen Platzhalter zur Modellierung von funktionalen Abhängigkeiten verwendet werden können.

BPMN ermöglicht die in Abschnitt 5.1.1 vorgestellten Varianten. Die Modellierung von Daten erfolgt in beiden Fällen extern über XSDs bzw. Java-Objekte. Problematisch bei der Verwendung von XSDs ist die zusätzliche Komplexität der Datenmodellierung. Des Weiteren ist die Definition der funktionalen Abhängigkeiten wenig komfortabel, da zur Modellierung viele verschiedene Elemente richtig verknüpft werden müssen. Um dieser Komplexität entgegen zu wirken, wurde bei Camunda BPM auf diese Art der Datenmodellierung verzichtet und der Erweiterungspunkt *ExtensionElements* im BPMN Metamodell verwendet, um ein eigenes Datenmodell abzubilden. Ein weiterer Vorteil ist die programmatische Konfigurierbar- und Parametrisierbarkeit von Prozess Instanzen mit Hilfe von Java Objekten. Dies führt im Allgemeinen zu einer höheren Flexibilität des Systems. Ein Nachteil dieser Variante ist, dass die Implementierung der *ExtensionElements* von Camunda BPM nicht in der BPMN Spezifikation festgelegt ist und somit nicht von jedem BPMS unterstützt werden muss.

Dennoch ist die Verwendung der Camunda BPM Variante sinnvoll, da sie durch ihr einfaches Konzept flexibler und praktikabler ist. Neben den oben genannten Vorteilen erleichtert diese Variante die Implementierung von Vorlagen für Aktivitäten, da diese in JARVIS dynamisch zur Laufzeit generiert werden müssen. Dies ist nötig, da nur zur Laufzeit feststeht welche Aktivitäten zur Verfügung stehen. Zusätzlich ermöglicht dies, das dynamische Hinzufügen von neuen Aktivitäten und deren direkte Verwendung in Delivery Modellen. Dadurch muss nicht für jede neue Aktivität die Software neu gebaut und ausgeliefert werden. Es reicht aus, dass ein einzelner Service neu gestartet oder aktualisiert wird.

Die Verwendung von Vorlagen soll den Benutzer bei der Konfiguration der Aktivitäten in einer Pipeline unterstützen.

5.2. Konzept zur Integration in JARVIS

Das Ziel des Konzeptes ist die Erfüllung der Eigenschaft *Benutzerunterstützung* und die Integration von BPMN in das SDS JARVIS.

5.2.1. Überblick

Damit BPMN als externe Modellierungssprache verwendet werden kann, müssen zwei Komponenten hinzugefügt werden (vgl. Abbildung 5.3). Die Komponente *BPMN Modeler* stellt das Modellierungswerzeug dar, welches das Entwickeln neuer BPMN Pipelines ermöglicht. Über das *API Gateway* kann der Modeler mit der neuen Komponente *BPMN Model Service* kommunizieren. Dieser interne Service stellt zum einen Schnittstellen für den Modeler bereit und zum anderen implementiert er Kommandos um externe BPMN Pipeline Modelle bereitzustellen bzw. in das System zu importieren.

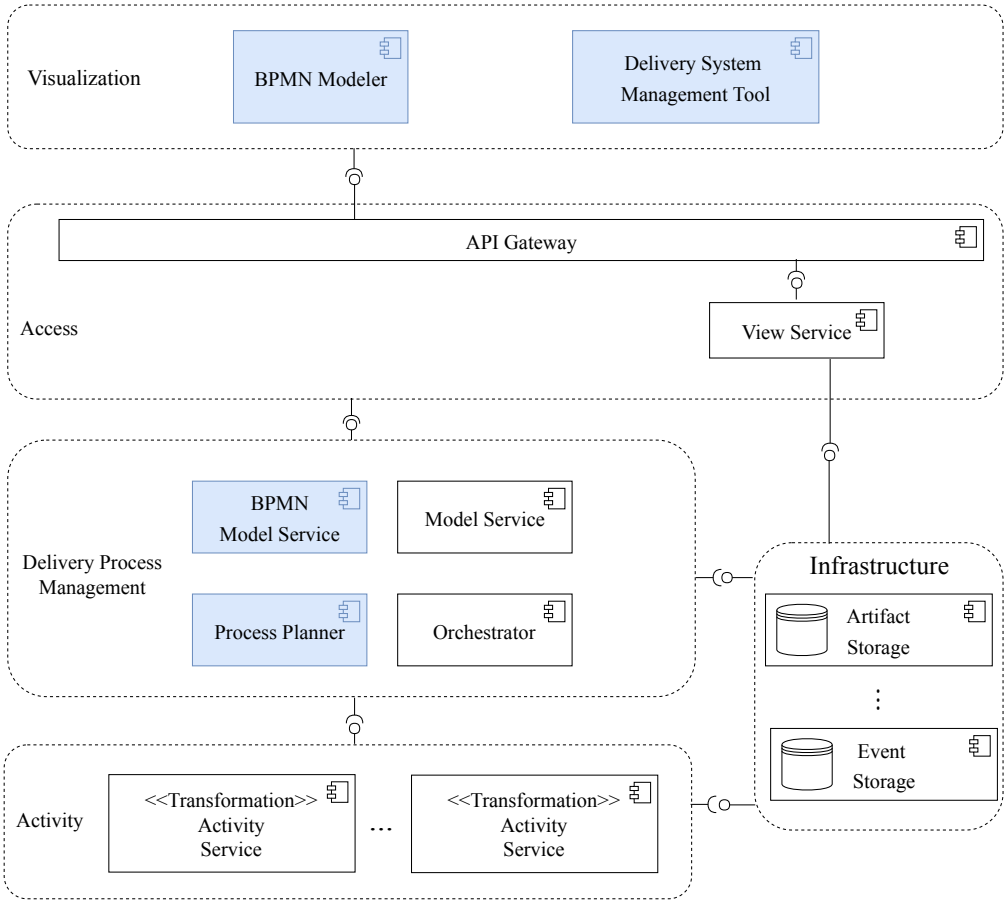

Abbildung 5.3.: Architektur zur Erweiterung von JARVIS (neue bzw. erweiterte Komponenten in blau)

Neben den beiden neuen Komponenten werden bereits existierende Komponenten erweitert. Die Komponente *Delivery System Management Tool* stellt die Oberfläche von JARVIS dar und ermöglicht dem Benutzer das Importieren von externen Delivery Modellen und das Ausführen der importierten Modelle. Damit sich der *BPMN Modeler* nahtlos in das bestehende System einfügen kann, wird die Oberfläche um eine neue Ansicht erweitert, die alle gespeicherten BPMN Modelle darstellt und den Zugriff auf den BPMN Modeler ermöglicht. Das Persistieren der Modelle wird vom *Artifact Storage* übernommen. Dieser verwendet eine mongoDB [Mon] zur Datenhaltung. Bisher konnte der *Artifact Storage* Dateien speichern und einzelne Dateien nur über eine *ArtifactId* abfragen. Um einen einfachen Umgang mit BPMN Modellen zu ermöglichen, muss eine Abfrage mit Hilfe des Dateinamens und eine Auflistung aller BPMN-Modelle implementiert werden.

Zusätzlich muss der *Process Planner* erweitert werden. Dieser ist für die Transformation von externen Modellen in das interne Modell nötig und soll nun BPMN Modelle unterstützen.

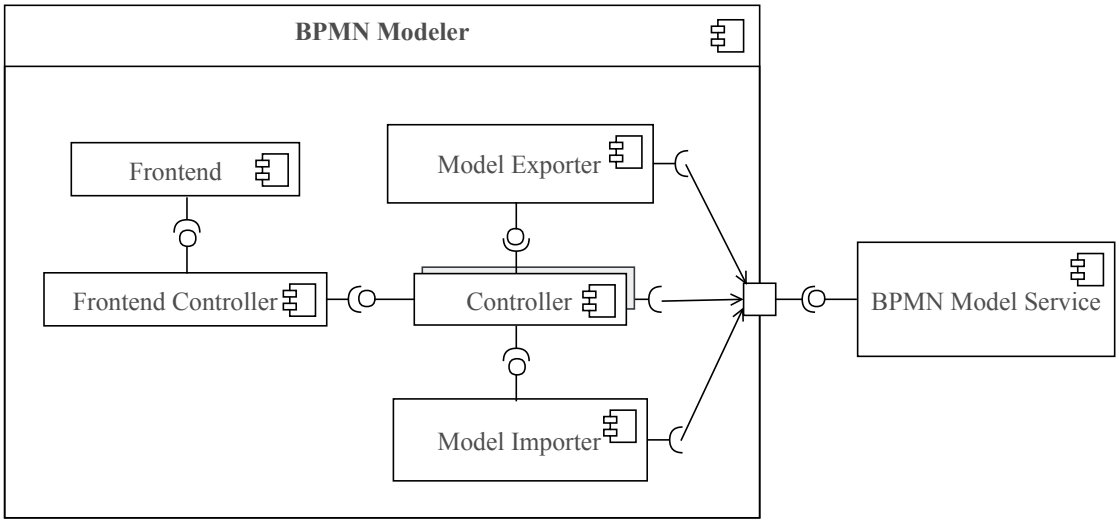

Abbildung 5.4.: Architektur des BPMN Modelers aus der Server-Perspektive

5.2.2. Komponenten-Sicht

Im vorherigen Abschnitt wurde festgestellt, dass zwei neue Komponenten für die Integration von BPMN als externe Modellierungssprache in JARVIS benötigt werden. Im Folgenden wird die Architektur der neuen Komponenten vorgestellt.

BPMN Modeler

Durch die Erweiterung von JARVIS um ein Modellierungswerkzeug soll die *Benutzerunterstützung (E.8)* umgesetzt werden. Der Modeler soll die Entwicklung von Delivery Modellen unterstützen, indem er dem Benutzer eine Oberfläche mit den folgenden Funktionalitäten bereit stellt:

- Grafisches Modellieren von Pipelines
- Unterstützung bei der Modellierung
  - Einfache Platzierung von Elementen
  - Konfiguration von Aktivitäten
- Laden existierender BPMN Modelle
- Speichern neuer Modelle

Um diese Funktionalitäten anbieten zu können, muss der Modeler verschiedene Komponenten beinhalten (vgl. Abbildung 5.4). Der BPMN Modeler kann als klassische Client-Server-Anwendung gesehen werden, die auf der Server-Seite eine Variante des *Model-View-Controller (MVC)*-Musters implementiert. Der *View* wird dabei im Browser durch die Komponente *Frontend* dargestellt und vereint weitere Komponenten.

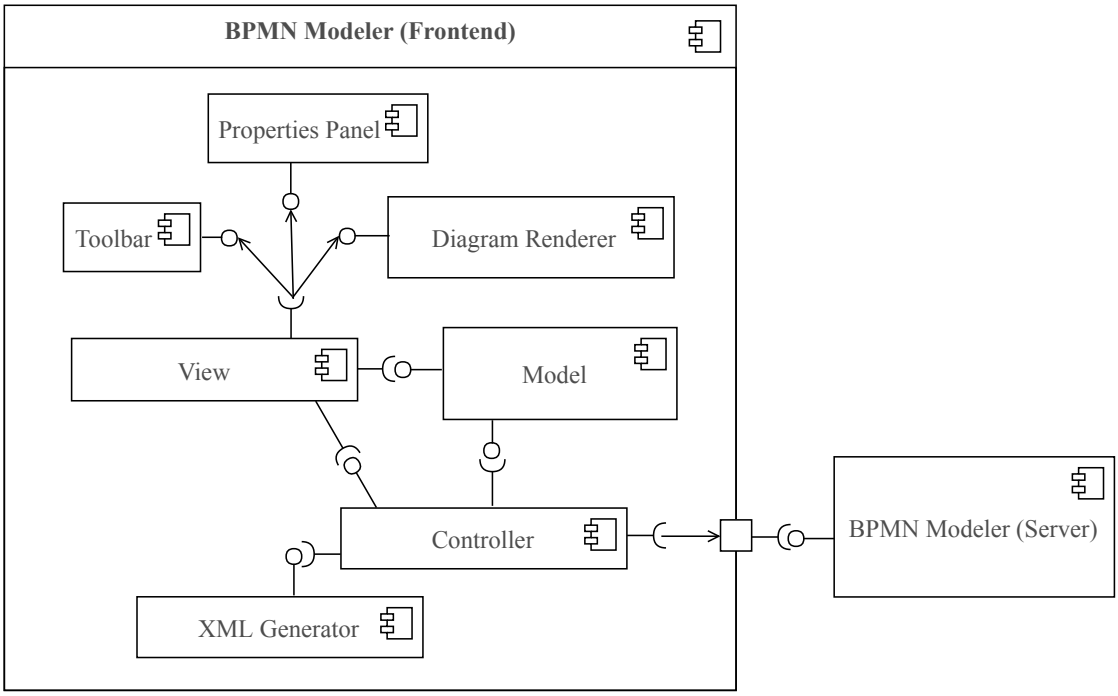

Abbildung 5.5.: Architektur des BPMN Modelers aus der Client-Perspektive

Beide Seiten stellen eigenständige Teilsysteme dar und werden daher getrennt voneinander betrachtet. Zunächst wird die Server-Seite vorgestellt.

Benutzer-Aktionen oder Eingaben werden vom *Frontend* über HTTP an die Server-Komponente *Frontend Controller* weitergereicht. Diese ist für die Verarbeitung der Anfragen und für das Zurückliefern von *Views* zuständig. Dazu leitet er die Anfragen an dedizierte *Controller* weiter. Ein *Controller* erstellt bzw. aktualisiert eigenständig das *Model*. Das neue *Model* wird anschließend vom *Frontend Controller* zur dynamischen Generierung des *Views* verwendet.

Die Komponenten *Model Exporter* und *Model Importer* können von *Controllern* verwendet werden. Der *Model Importer* ermöglicht das Importieren existierender BPMN Modelle. Es ist vorgesehen, dass Dateien per *Drag and Drop* oder per *BPMN Model Service* importiert werden können. Um der Oberfläche diese Modelle zur Verfügung zu stellen wird nach dem Import das *Model* entsprechend aktualisiert.

Zum Speichern der Modelle wird der *Model Exporter* benötigt. Koordiniert durch den *Controller* wird das BPMN Modell aus dem *Model* abgefragt und über den *Model Exporter* an den *BPMN Model Service* weitergereicht.

Wie bereits oben beschrieben, stellt die Client-Seite ein eigenständiges Teilsystem dar und kann deshalb in weitere Komponenten unterteilt werden (vgl. Abbildung 5.5). Die *View*-Komponente stellt eine *Hypertext Markup Language (HTML)*-Seite bereit und verwendet dabei die Komponenten *Toolbar* und *Properties Panel*. Ungültige Benutzer-Eingaben, wie zum Beispiel das unzulässige Platzieren eines Elementes aus der *Toolbar*

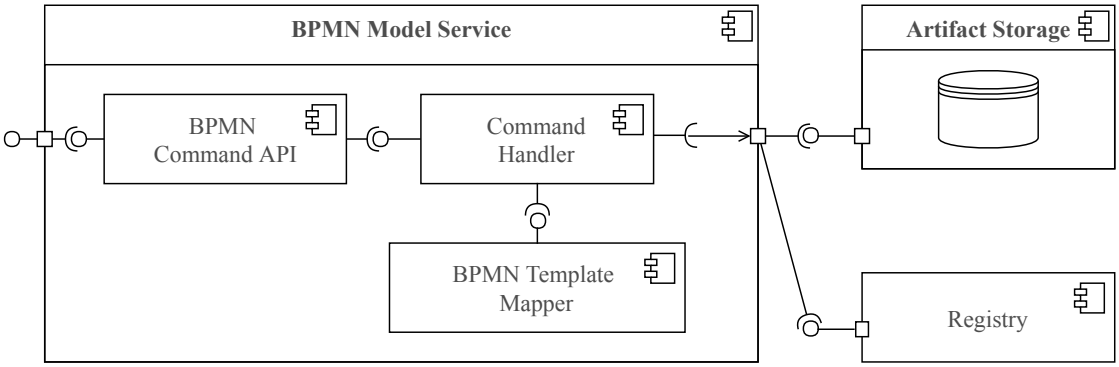

Abbildung 5.6.: Architektur des BPMN Model Services

im Diagramm, werden vom *Controller* erkannt und verhindert. Um den Benutzer bei der Konfiguration zu unterstützen, ruft der *View* über den *Controller* Vorlagen für Aktivitäten ab. Dazu verwendet dieser einmalig beim ersten Aufruf die Server-Seite des *BPMN Modelers* und speichert die Vorlagen im *Model*.

Der Benutzer konfiguriert Aktivitäten über das *Properties Panel*. Wie die *Toolbar*, handelt es sich dabei um eine Teilkomponente der Oberflächen.

Das Speichern von Modellen erfolgt über den *BPMN Model Service*. Dazu extrahiert der *XML Generator* alle BPMN Elemente aus dem internen Datenmodell und wandelt sie anschließend in das standardisierte XML-Datenformat um. Dies wird daraufhin vom *BPMN Model Service* im *Artifact Storage* persistiert.

BPMN Model Service

Der *BPMN Model Service* ist ein interner Service von JARVIS der dem *BPMN Modeler* unterschiedliche Schnittstellen bereitstellt und folgende Funktionalitäten besitzt:

- CRUD-Operationen für BPMN Modelle
- Bereitstellung der Aktivitäts-Vorlagen
- Import von ausführbaren Modellen

Die Architektur von JARVIS erlaubt es, durch wenige Komponenten, neue externe Modellierungssprachen einzubinden (vgl. Abbildung 5.6). Da JARVIS das *Command Query Responsibility Segregation (CQRS)* Muster [Bet13] verwendet sind Kommandos die zentralen Objekte zur Modifizierung des System-Zustandes und werden daher auch zur Interaktion mit der Komponente verwendet. Sie werden vom *Command Handler* ausgeführt, welcher dafür auf den *Artifact Storage* zugreifen kann, um zum Beispiel BPMN Modelle speichern oder abrufen zu können.

Eine weitere Funktionalität der Komponente ist die Bereitstellung von Aktivitäts-Vorlagen. Dazu bezieht der *Command Handler* über die *Registry* alle Spezifikationen der zur Verfügung stehenden Aktivitäten. Im Anschluss wandelt der *BPMN Template Mapper* diese in ein entsprechendes Format für den *BPMN Modeler* um.

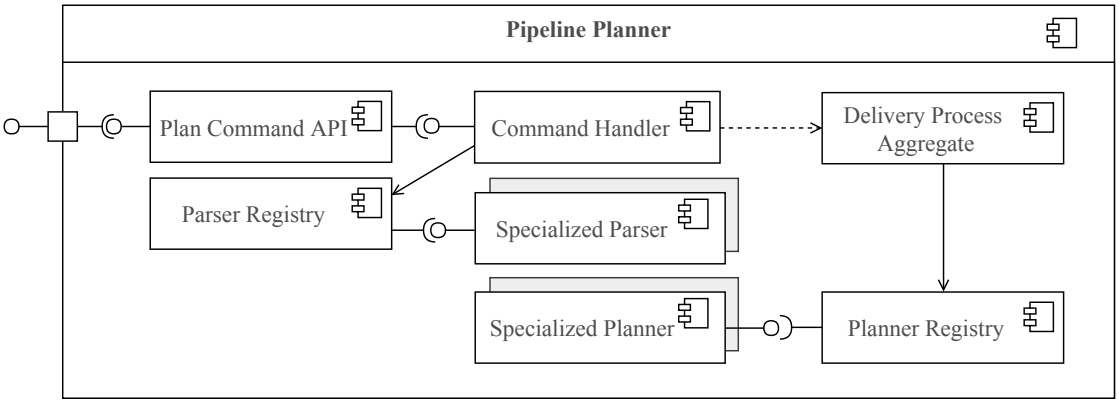

Abbildung 5.7.: Architektur des Pipeline Planners

BPMN Command API

Das *BPMN Command API* ist Teil des *BPMN Model Services* und ermöglicht es anderen Services, wie zum Beispiel dem *Model Service*, Kommandos an die Komponente zu senden. Zur Beschreibung der Schnittstelle wird die Corba IDL [Obj] verwendet (vgl. Quellcode 5.1).

Die Komponente stellt im Wesentlichen zwei unterschiedliche Schnittstellen bereit. Die Schnittstelle *BpmnModelInteraction* bietet insgesamt fünf unterschiedliche Operationen. Dazu zählen die Standard-CRUD-Operationen, die vom *BPMN Modeler* benötigt werden und die Operation *importBpmnModel*. Ein Sonderfall stellt die Read-Operation *searchBpmnModels* dar. Dies ist eine generische Schnittstelle die von Komponenten in JARVIS implementiert werden kann, um als *External Model Provider* zu fungieren. Dies erlaubt das Abfragen aller gespeicherten BPMN Modelle über die Oberfläche von JARVIS. Anschließend können die gefundenen Modelle über die *importBpmnModel*-Operation in das System importiert und schlussendlich auch ausgeführt werden. Der Zugriff auf die BPMN Modelle aus dem *Artifact Storage* erfolgt mit Hilfe des Dateinamens als Identifizierungsmerkmal. Alle BPMN Modelle werden im XML-Format gespeichert.

Darüber hinaus wird neben der oben beschriebenen Schnittstelle, die Schnittstelle *BpmnTemplateProvider* publiziert. Die Operation *searchBpmnTemplates* wird vom *BPMN Modeler* verwendet um die Vorlagen für die Aktivitäten zu erhalten.

Pipeline Planner

Der *Pipeline Planner* ist eine bereits existierende Komponente von JARVIS die erweitert werden muss. Der Planner schließt die Lücke zwischen den externen und internen Datenmodellen, indem er unter anderem für die Umwandlung der Modelle verantwortlich ist [Dör18]. Aktuell kann der Planner nur auf YAML-basierende externe Modelle verarbeiten. Die Umwandlung wird durch sogenannte *ModelParser* durchgeführt. Abbildung 5.7 zeigt die für die Erweiterung relevanten Bestandteile der Komponente. JARVIS verwendet das

```
module BpmnCommandAPI {
    interface BpmnModelInteraction {
        void deleteBpmnModel(in DeleteBpmnModelCmd);
        void importBpmnModel(in BpmnModelImportCmd);
        void saveBpmnModel(in SaveBpmnModelCmd);
        void updateBpmnModel(in UpdateBpmnModelCmd);
        List<String> searchBpmnModels(in BpmnModelSearchCmd);
    };

    interface DeleteBpmnModelCmd {
        readonly attribute string fileName;
    };

    interface BpmnModelImportCmd {
        readonly attribute string fileName;
    };

    interface SaveBpmnModelCmd {
        readonly attribute string fileName;
        readonly attribute string xmlPipeline;
    };

    interface UpdateModelCmd {
        readonly attribute string fileName;
        readonly attribute string xmlPipeline;
    };

    interface BpmnModelSearchCmd {
    };

    interface bpmnTemplate {
        List<BpmnTemplate> searchBpmnTemplates(in
            BpmnTemplateSearchCmd);
    }

    interface BpmnTemplateSearchCmd {
    };
};
```

Quelltext 5.1: Spezifikation der BPMN Command API

CQRS Muster und somit kann auch der *Pipeline Planner* über Kommandos angesprochen werden. Trifft ein entsprechendes *PlanPipelineCmd* an der Komponente ein, wird ein passender *Command Handler* ausgeführt. Dieser wandelt zunächst das übergebene externe Modell in das interne Datenmodell um. Anschließend kann es von entsprechenden *Planner*-Komponenten weiter verarbeitet werden. Diese dienen der Anreicherung und Optimierung des Modells. Danach kann das geplante Modell ausgeführt werden.

Die erweiterbare Architektur der Komponente erlaubt das einfache Hinzufügen von neuen *Model Parsern*. Dazu muss für BPMN lediglich ein weiterer *Model Parser* implementiert werden. Über die *Parser Registry* werden alle Parser registriert und abgerufen.

Der neu zu implementierende *BpmnModelParser* erhält als Eingabe das BPMN Modell im XML-Format. Dieses dient dem Datenaustausch und daher muss die Komponente die XML-Dateien in ein verarbeitbares Datenformat umwandeln. Die von der BPMN Spezifikation bereitgestellten XSDs sollen verwendet werden um ein entsprechendes Java-Klassenkonstrukt zu generieren. Auf Basis dieser Klassen soll die Verarbeitung des BPMN Modells erfolgen.

5.2.3. Prozess-Sicht

In diesem Abschnitt werden die dynamischen Aspekte der Komponenten behandelt.

Modellierung von BPMN Modellen

Das Modellieren von neuen Modellen erfolgt über den *BPMN Modeler*. Abbildung 5.8 zeigt den typischen Ablauf der Modellierung von Delivery Modellen mit Hilfe von BPMN. In der Regel besteht das Platzieren eines Elementes aus drei verschiedenen Schritten. Als erstes wird ein Element aus der *Toolbar* ausgewählt und auf dem Arbeitsblatt platziert. Das erzeugte *placementEvent* sorgt dafür, dass das *Model* aktualisiert wird. Im Anschluss wird der *View* benachrichtigt. Dieser fragt das aktuelle interne Modell ab und weist den *DiagramRenderer* an, das Arbeitsblatt neu zu rendern.

Handelt es sich beim platzierten Element um eine Aktivität, kann der Benutzer dieses auswählen um es konfigurieren zu können. Nach dem Selektieren der Aktivität holt sich der *View* die zur Verfügung stehenden Vorlagen vom *Controller* und aktualisiert die Oberfläche inklusive dem *Properties Panel*. Durch die Auswahl einer Vorlage werden automatisch mögliche Eingaben auf dem *Properties Panel* angezeigt.

Zum Schluss muss der Benutzer alle benötigten Eingaben konfigurieren. Das Setzen einer Eigenschaft wird solange wiederholt bis alle Eingaben festgelegt wurden. Im Anschluss kann ein weiteres Element dem Diagramm hinzugefügt werden.

Öffnen von BPMN Modellen

Das Modellieren von BPMN Modellen soll auf drei unterschiedliche Arten begonnen werden können. Um von Grund auf ein neues BPMN Modell entwickeln zu können, muss es möglich sein den Modeler mit einem leeren Arbeitsblatt zu starten. Beim ersten Aufruf

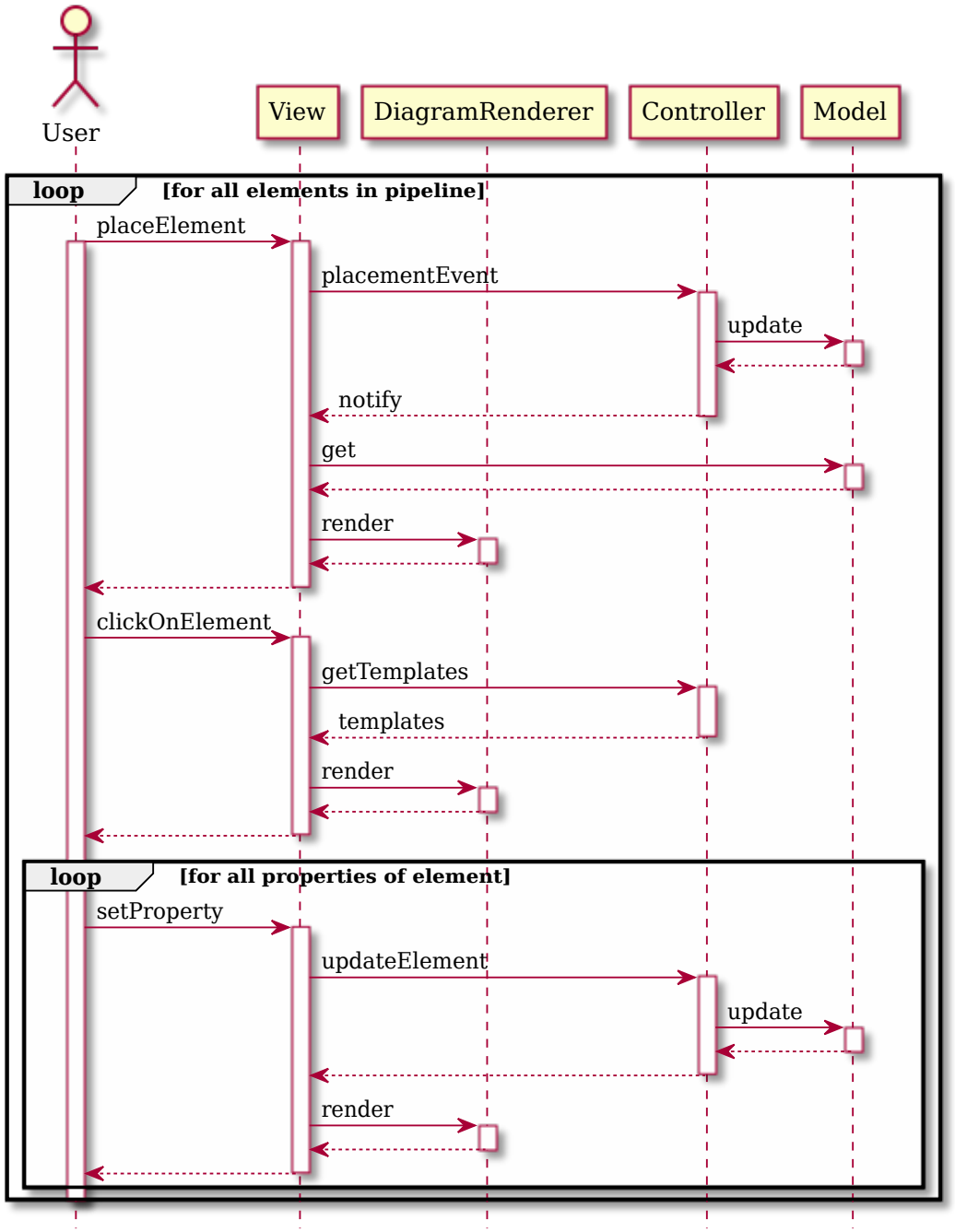

Abbildung 5.8.: Typischer Ablauf der Modellierung von BPMN Pipelines (Sequenzdiagramm)

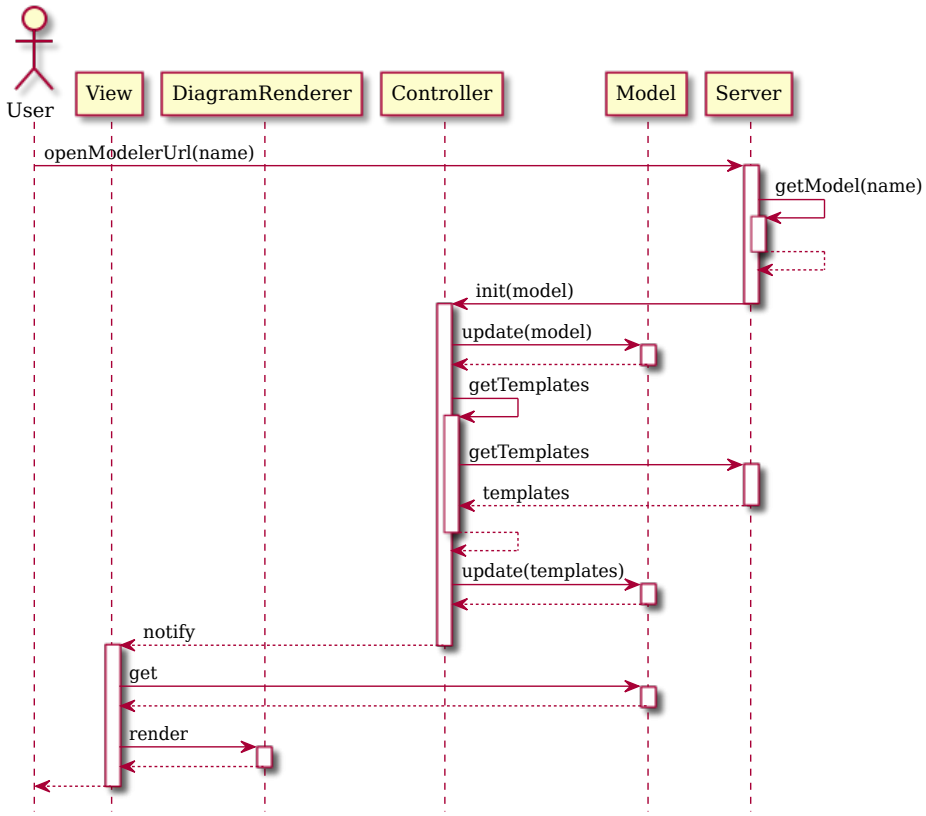

Abbildung 5.9.: Öffnen eines BPMN Modells (Sequenzdiagramm)

des Modelers sollen im Hintergrund automatisch alle möglichen Vorlagen vom Server abgefragt werden.

Neben dem Neuentwickeln von Modellen soll es möglich sein bereits existierende Modelle aufzurufen und zu bearbeiten. Zum einen sollen Modelle, die als Datei auf dem Entwicklerrechner vorliegen, per *Drag and Drop* eingefügt werden und zum anderen sollen bereits gespeicherte Modelle aus dem *Artifact Storage* geöffnet werden.

Stellvertretend für alle anderen Fälle wird in Abbildung 5.9 das Öffnen eines bereits existierenden Modells aus dem *Artifact Storage* betrachtet. Zur besseren Darstellung vereint der *Server* die Server-Seite inklusive des *Artifact Storage*. Der Benutzer ruft den Modeler über eine *URL* auf und übergibt dabei den Namen des Modells. Auf der Server-Seite wird das entsprechende Modell aus dem *Artifact Storage* geladen und der *Controller* der Client-Seite wird mit diesem befüllt. Anschließend wird die Seite geladen und der *Controller* aktualisiert das interne Datenmodell mit dem vorkonfigurierten Modell. Danach werden die aktuellen Vorlagen vom Server abgefragt und im *Model* gespeichert. Nachdem alle benötigten Daten geladen wurden, kann der *DiagrammRenderer* das importierte Diagramm darstellen.

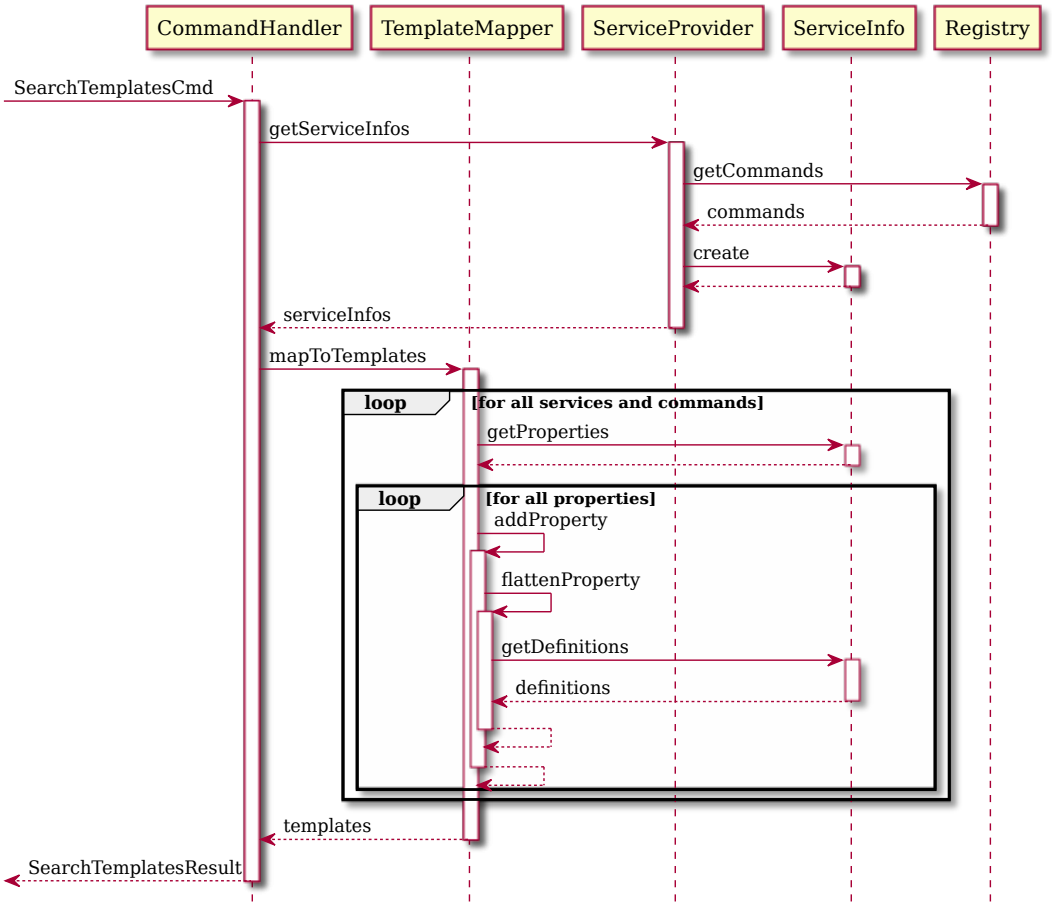

Abbildung 5.10.: Generierung von Vorlagen (Sequenzdiagramm)

Speichern von BPMN Modellen

Das Speichern von Modellen erfolgt über einen *Speichern*-Button auf der Oberfläche. Nachdem dieser angeklickt wurde, wird aus dem internen Datenmodell ein BPMN-Diagramm im XML-Format generiert und mitsamt eines Dateinamens an den Server geschickt. Die Generierung des Diagramms wird von der *XMLGenerator*-Komponente übernommen. Der Server speichert daraufhin das Modell im *Artifact Storage*.

Dynamische Generierung von Vorlagen

Das Generieren der Vorlagen erfolgt auf Basis der von JARVIS zur Verfügung gestellten Spezifikationen aller Services, die Aktivitäten bereitstellen. Diese werden zur Laufzeit ermittelt, wodurch die Flexibilität der Architektur erhalten bleibt und dynamisch hinzugefügte Aktivitäten bei der Generierung der Vorlagen mit einbezogen werden. Angestoßen wird die Generierung durch ein *SearchTemplatesCmd*. Der Ablauf ist in Abbildung 5.10 dargestellt. Zunächst bestimmt der *Command Handler* über den *Service Provider* alle

aktuell zur Verfügung stehenden Service-Spezifikationen (genannt ServiceInfos) über die *Registry*. Eine solche Spezifikation beinhaltet den Namen des Services und eine Liste aller existierenden Kommandos, die der Service entgegen nehmen kann. Ein Kommando kann vom Typ *Transformation, Assessment* oder *Quality-Gate* sein und repräsentiert somit eine Aktivität im Domänenmodell. Das bedeutet auch, dass pro Kommando eine Vorlagen angelegt werden muss. Da die Spezifikationen zunächst als JSON-Objekt vorliegen, müssen sie noch in *ServiceInfo*-Objekte umgewandelt werden.

Nachdem der *CommandHandler* alle Service-Spezifikationen vorliegen hat, kann mit dem Erstellen der Vorlagen weitergemacht werden. Dafür wird über alle Services und deren Kommandos iteriert und jeweils eine Vorlage für ein Kommando erstellt.

Die Spezifikation der Kommandos definiert alle Eingabe- und Ausgabe-Parameter der repräsentierten Aktivität. Die Parameter sind hierarchisch strukturierte Key-Value-Paare, die beliebig oft verschachtelt werden können. Daher ist es zunächst nötig jeden Parameter zu *ebnen*, da die Vorlagen nur eine *Ebene* beinhalten sollen. Dieser Prozess wird im Folgenden vorgestellt.

Ein Parameter der Spezifikation ist ein 3-Tupel und die Elemente können die folgenden Typen annehmen:

- (String, ParameterValue, Boolean)
- (String, List<Parameter>, Boolean)
- (String, Reference, Boolean)

Ein Parameter kann als Key-Value-Paar angesehen werden. Zusätzlich kann angegeben werden, ob ein Parameter zur Ausführung einer Aktivität gesetzt sein muss oder nicht (letztes Element im Tupel). Dadurch kann der Modeler dem Benutzer mitteilen, wenn erforderliche Parameter nicht konfiguriert wurden.

Der einfachste Typ ist ein Key-Value-Paar, wobei der Wert als String und der Schlüssel als ParameterValue dargestellt wird. Diese können direkt, ohne weitere Berechnungen in die Vorlage übernommen werden. Aufwändiger ist die Bestimmung von Parametern, bei denen der Wert eine Liste von Parametern ist. Angenommen wir haben das 3-Tupel („project“, {(„groupId“,<ParameterValue>), („artifactId“, <ParameterValue>)}, <Constraints>). Die Operation *flattenProperty* bestimmt daraus die Tupel:

- (project.artifactId, <ParameterValue>, <Constraints>)
- (project.groupId, <ParameterValue>, <Constraints>)

Das zweite Element des Tupels zeigt hier an, das der Wert ein String sein muss. Da eine Liste alle Typen von Parametern beinhalten kann, können theoretisch auch Referenzen oder wieder Listen verwendet werden. Falls es sich um eine Liste handelt, wird das *ebnen* erneut durchgeführt, bis der Schlüssel-Typ ein String ist. Handelt es sich um eine Referenz muss diese zunächst aufgelöst werden. Referenzen beziehen sich auf Listen oder Strings, die an einer anderen Stelle in der Spezifikation, dem *Definitionsbereich*, festgelegt werden.

Angenommen wir verändern das oben beschriebene Tupel wie folgt: („project“, {„#/definitions/projectProps“}, <Constraints>) und verwenden den *Definitionsbereich* („projectProps“, {(„artifactId“, <ParameterValue>), („groupId“, <ParameterValue>)}). Dann „folgt“ die Operation der Referenz und das produzierte Ergebnis unterscheidet sich nicht vom Obigen. Es ist möglich, dass Referenzen wieder Referenzen enthalten. Die Operation folgt diesen bis ein Parameter vom Typ (String, ParameterValue, Boolean) vorliegt und fügt ihn erst dann der Vorlage hinzu.

Schlüssel eines Parameters sind vom Typ *ParameterValue*. Bei diesem handelt sich es um einen generischen Typ, der die konkreten Typen: *String, Boolean, Choice* zusammenfasst. Die zugrunde liegende Idee hinter diesen drei Typen ist, dass wir nicht nur einfache Zeichenketten konfigurieren, sondern dem Benutzer die Konfiguration erleichtern wollen, indem wir unterschiedliche Darstellungskomponenten verwenden, abhängig von den in der Spezifikation angegebenen Informationen. Das bedeutet, dass wenn die Spezifikation bereits eine mögliche Wertemenge vorgibt, dann soll der Benutzer über ein Dropdown-Menü entsprechende Werte direkt auswählen können (*Choice*). Das erleichtert die Konfiguration der Aktivität und dient der *Benutzerunterstützung (E.8)*. Im Falle des Typen *Boolean* soll der Benutzer auch nur *true* oder *false* auswählen können. Dies soll entweder über ein Dropdown-Menü oder aber über eine *Checkbox* realisiert werden.

Importieren und Ausführen von Modellen

Der Prozess, der das Importieren von Modellen ermöglicht existiert bereits für das Importieren von Modellen aus der Versionsverwaltungssoftware GIT. Aufgrund der Tatsache, dass der Modellierungsprozess in JARVIS durch den *BPMN Modeler* integriert wird, bietet es sich an, beide Prozesse miteinander zu koppeln. Dadurch wird in JARVIS ein durchgängiges Arbeiten von der Modellierung bis hin zur Ausführung der Modelle ermöglicht.

Der vollständige Prozess ist in Abbildung 5.11 zu sehen. Zunächst erstellt der Benutzer sein Delivery Modell im *BpmnModeler* (vgl. Abbildung 5.8). Anschließend speichert er das Modell im *ArtifactStorage*. Über die Oberfläche kann der Benutzer den Importvorgang starten. Dazu wählt der Benutzer in der Eingabemaske zunächst den *External Model Provider* aus. Damit der neue *BpmnModelService* als Provider ausgewählt werden kann, muss dieser das spezielle *searchModelsCmd* implementieren. In diesem Fall fragt das Kommando alle im *Artifact Storage* gespeicherten BPMN Modelle ab. Nachdem der Benutzer eines dieser Modelle auf der Oberfläche ausgewählt hat, kann der Import angestoßen werden. Bei diesem Import wird nicht nicht nur das Modell gespeichert, sondern es wird ein neues *PipelineModelAggregate* angelegt, welches eine Referenz auf das Modell enthält. Dieses Prinzip wurde auch bei der bestehenden Variante mit GIT verwendet. Der Unterschied zwischen beiden Varianten liegt nur in der Persistenz-Implementierung. Beim *BpmnModelService* werden Modelle im *ArtifactStorage* abgelegt und bei GIT liegen sie direkt im Repository. Da durch diese Flexibilität keine direkte Referenz auf den *ArtifactStorage* und das darin gespeicherte Modell möglich ist, wird stattdessen ein parametrisiertes *importPipelineModel*-Kommando eines Services referenziert. Dieses kann über die angegebenen Parameter ein bestimmtes Modell ermitteln. Im Fall von BPMN wird der *BpmnModelService* referenziert.

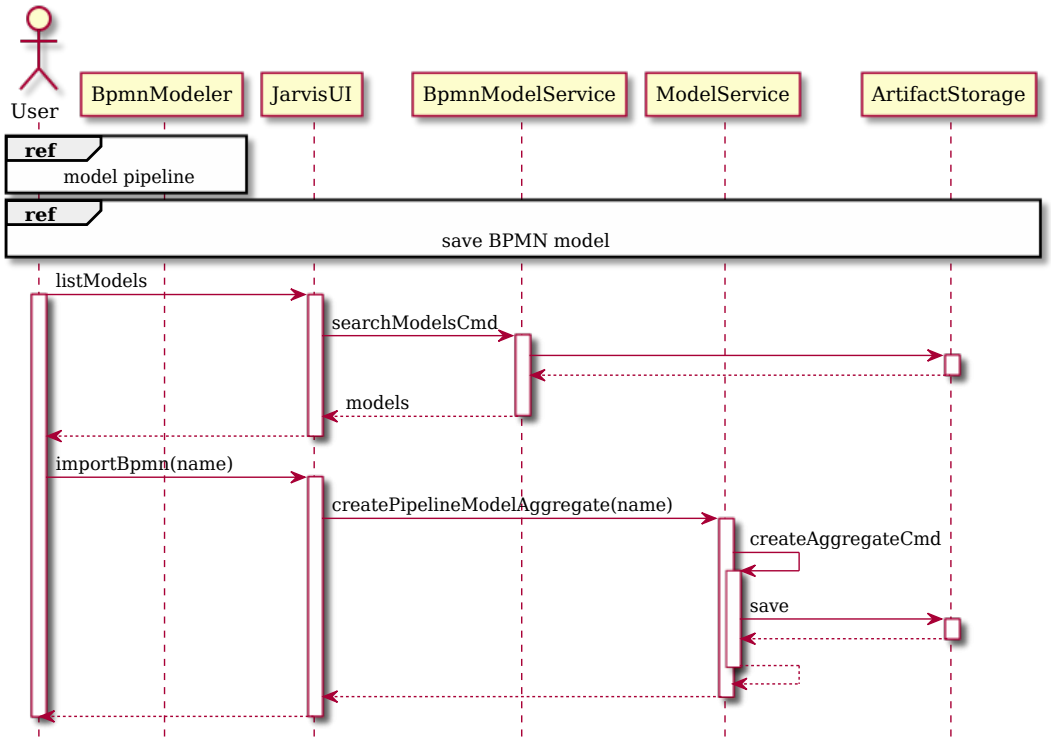

Abbildung 5.11.: Importieren von Modellen (Sequenzdiagramm)

Vorbereitungen zur Ausführung

Nach dem Anlegen des *PipelineModelAggregates* kann die Ausführung erfolgen. Angestoßen über die Oberfläche wird vor der Ausführung des Modells das referenzierte *importPipelineModel*-Kommando vom *ModelService* erstellt und an den *BpmnModelService* weitergereicht. Abbildung 5.12 stellt diesen Prozess dar.

Das Kommando ermittelt das korrekte Modell und im Anschluss wird das Planen der Ausführung über den *PipelinePlanner* angestoßen. Der Planungsschritt ist nötig, da JARVIS mit dem Ziel der *Selbstorganisation* entworfen wurde. Das bedeutet, dass es das externe Modell analysiert und sowohl fehlende Parameter bei der Konfiguration von Aktivitäten ergänzt, als auch die Ausführungsreihenfolge ermittelt und neu berechnet. Damit die Analyse möglichst unabhängig von der konkreten Modellierungssprache erfolgen kann, wird das externe BPMN Modell zunächst in das interne Modell überführt. Dieser Schritt muss für jede neue externe Modellierungssprache umgesetzt werden. Das interne Datenmodell verwendet die aus der Domäne bekannten Elemente (vgl. Abschnitt 4.2). Aus diesem Grund implementiert der *BpmnModelParser* die in Abschnitt 5.1.1 eingeführte Abbildung der BPMN Elemente auf die Delivery Prozess Domäne.

Abbildung 5.13 zeigt den internen Ablauf des *BpmnModelParsers*. Zur besseren Übersicht ist auf nicht relevante Bestandteile des Prozesses verzichtet worden. Wie bereits in

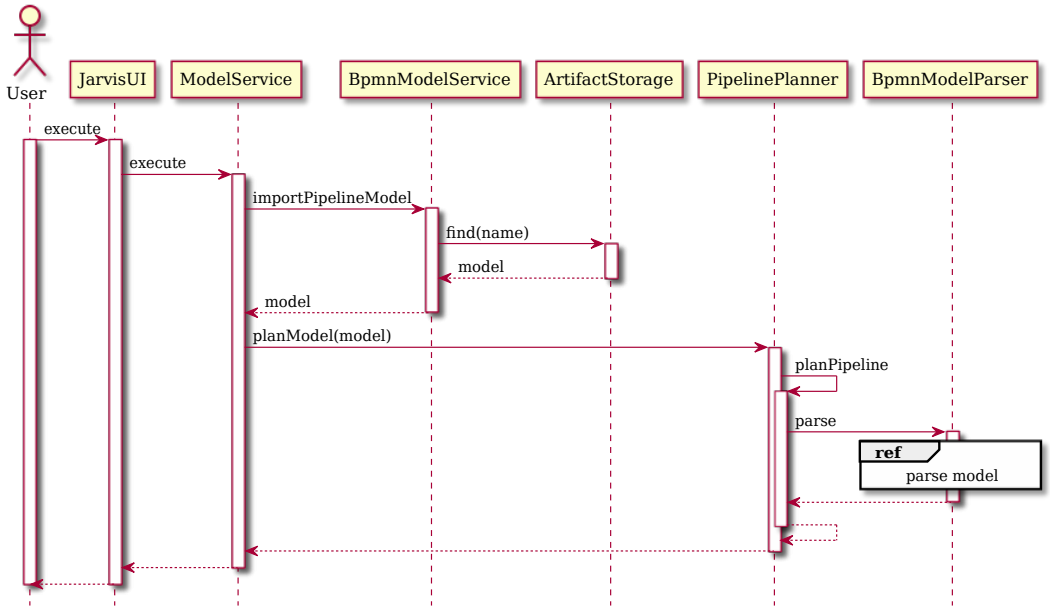

Abbildung 5.12.: Vorbereitungen zur Ausführung (Sequenzdiagramm)

Abschnitt 5.2.2 beschrieben, wandelt der *BpmnModelParser* zunächst das im XML-Format vorliegende Delivery Modell in ein Objekt um. Die Klassenstruktur des Objekts wurde mit Hilfe der BPMN XSDs generiert. Anschließend werden Basis-Validierungen durchgeführt, welche sicherstellen, dass nur die erwarteten BPMN Elemente zur Modellierung der Pipeline verwendet wurden.

Das interne Datenmodell von JARVIS zur Beschreibung von Delivery Modellen lässt sich in vier Gruppen unterteilen. Die erste Gruppe beschreibt welche Phasen in der Pipeline existieren und welche *Transformations* in welcher Phase ausgeführt werden sollen. Die zweite bzw. dritte Gruppe beschreibt die Konfiguration aller *Transformations* bzw. *Assessments* in der Pipeline. Die letzte Gruppe beinhaltet alle *Quality-Gates* und deren *Policies*.

Angelehnt an die Struktur des internen Datenmodells werden die einzelnen Gruppen nacheinander bestimmt. Zur Bestimmung der Phasen werden alle *Transformations* ermittelt und den Phasen zugeordnet. Im Anschluss wird die Konfiguration der *Transformations* und *Assessments* ermittelt. Dabei können Key-Value-Parameter, ohne Referenzen oder Objekte als Schlüssel, direkt aus dem BPMN Modell in das interne Modell übernommen werden (*mapSimpleProps*). Handelt es sich um Parameter mit Objekten muss für die Umwandlung zunächst das *Ebnen* rückgängig gemacht werden. Anschließend kann der Parameter in das interne Modell übernommen werden.

Die Verarbeitung von Parametern mit Referenzen als Schlüssel wird vom *DependenciesIdentifier* übernommen. Dieser ermittelt funktionale Abhängigkeiten und bereitet die Referenzen auf. Funktionale Abhängigkeiten werden anhand der Parameterwerte erkannt, welche die Form „@<taskId>/<depFlatten>“ besitzt. Die *FlowElementsFacade* bildet

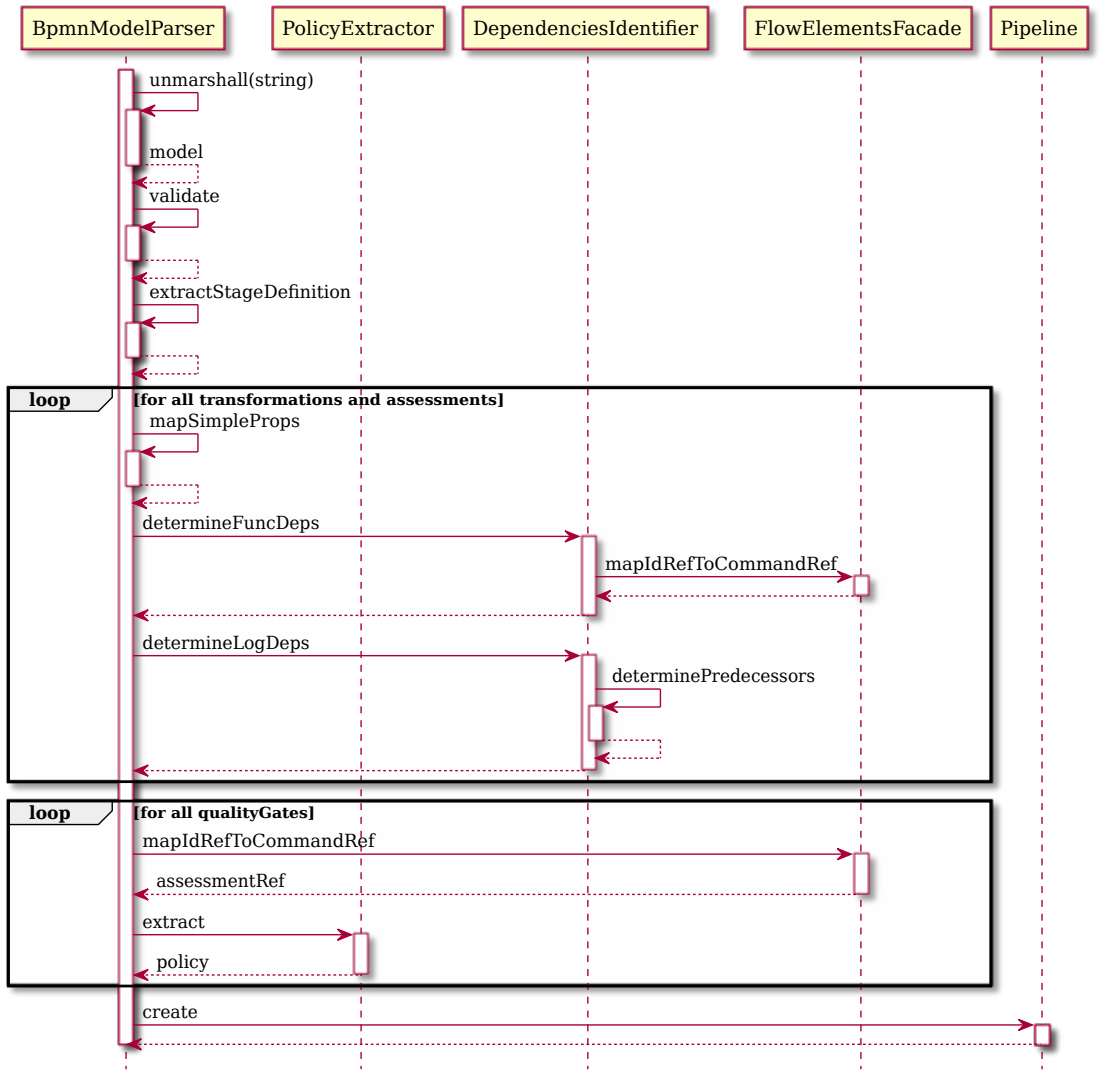

Abbildung 5.13.: Umwandlung des BPMN Modells in das interne Datenmodell (Sequenzdiagramm)

die angegebene *TaskId* auf den Namen des im Task referenzierten Kommandos ab. Dieser Name wird verwendet um die referenzierte Aktivität innerhalb der aktuellen Aktivität zu identifizieren. Im letzten Schritt wird die TaskId durch diese Referenz ersetzt. Das *Ebnen* muss in diesem Fall nicht rückgängig gemacht werden, sondern nur an das interne Modell angepasst werden, da dieses keine Punkte, sondern Schrägstriche zur Trennung verwendet.

Nachdem die funktionalen Abhängigkeiten der Aktivität ermittelt wurden, müssen noch die logischen Abhängigkeiten bestimmt werden. Dabei werden die in Abschnitt 5.1.1 beschriebenen Fälle beachtet.

Zum Schluss müssen noch die *Quality-Gates* ermittelt werden. Das interne Datenmodell unterscheidet dabei zwischen *Quality-Gates* und *Policies*. Prinzipiell ist es möglich, dass ein *Quality-Gate* mehrere *Policies* definiert. Im Gegensatz dazu wurde bereits bei der Abbildung der Delivery Prozess Domäne keine getrennte Modellierung beider Elemente vorgenommen, da *Policies* kein Teil der Domäne sind (vgl. Abschnitt 4.1.1 und 5.1.1). Aus diesem Grund wird festgelegt, dass *Quality-Gates* und *Policies* nur über eine „1:1“ Beziehung modelliert werden können. Daher werden beide Elemente im Folgenden unter dem Begriff *Quality-Gate* zusammengefasst.

Ein *Quality-Gate* referenziert immer ein *Assessment*, da es die Ausgabe der Aktivität bewertet. Ähnlich wie bei den Referenzen der *Transformations* und *Assessments* werden diese in das interne Datenmodell umgewandelt. Anschließend kann die Konfiguration des *Quality-Gates* extrahiert werden. Da es sich hierbei um einfache Key-Value-Paare handelt, können diese direkt übernommen werden.

Als letzten Schritt fasst der *BpmnModelParser* alle ermittelten Objekte in einem Pipeline-Objekt zusammen. Damit ist die Aufgabe des *BpmnModelParser* beendet und der *PipelinePlanner* kann mit dem „planen“ des Modells beginnen. Da dieser Teil jedoch unabhängig von BPMN erfolgt sei hierbei für weitere Details auf [Dör18] verwiesen.

5.3. BPM Konzepte zur Analyse und Optimierung von Delivery Prozessen

Mit BPM werden verschiedene Konzepte zur kontinuierlichen Verbesserung von Prozessen definiert. In Kapitel 2 wurde bereits der BPM-Regelkreis vorgestellt. Dieser fokussiert die kontinuierliche Optimierung und Anpassung von Prozessen. Dafür werden Metriken in der Plan-Phase definiert, in der Do-Phase erfasst und anschließend in der Check-Phase ausgewertet (vgl. Abschnitt 2.1.1). Die Erfassung und Auswertung von KPIs wird in der Regel von einem BPMS unterstützt (vgl. Abschnitt 2.1.4). Dieses ermöglicht die aggregierte Darstellung von Messdaten und unterstützt damit die Analyse der erhobenen Daten.

Die mit BPM definierten Methoden *Prozess-Reporting* und *Business Activity Monitoring* ermöglichen innerhalb des Regelkreises die kontinuierliche Verbesserung eines Prozesses. Auch CD liegt dieses Prinzip zu Grunde (vgl. Abschnitt 2.6.1). Daher ist die Anwendung der Konzepte auch bei CD sinnvoll.

Für die Umsetzung beider Methoden muss als erstes festgelegt werden, welche KPIs im Prozess ausgewertet werden sollen. Da es eine Vielzahl von möglichen Kennziffern gibt, werden hier nur die folgenden betrachtet:

- Erfolgsrate
- Durchschnittliche Ausführungszeit
- Durchschnittliche Zeit der externen Kommunikation

Alle KPIs können sowohl auf Prozess-Ebene, als auch auf Aktivitäts-Ebene erfasst werden. Die Erfolgsrate wird in Prozent angegeben und gibt an, ob der Prozess oder die Aktivität zuverlässig durchgeführt wird. Dadurch lassen sich sowohl instabile Prozesse, als auch kritische Aktivitäten identifizieren.

Neben der Erfolgsrate ist auch die durchschnittliche Ausführungszeit bei der Auswertung wichtig. Durch eine zeitliche Betrachtung der Durchschnittswerte können Trends oder Abweichungen erkannt werden. Des Weiteren ermöglicht die separate Betrachtung der Durchschnittswerte von Aktivitäten die Identifikation einzelner Problemstellen im Prozess. Neben diesen beiden allgemeingültigen KPIs ist die durchschnittliche Zeit der externen Kommunikation speziell für JARVIS wichtig. Da JARVIS auf einer Microservice-Architektur basiert, wird bei der Ausführung von Aktivitäten meist nicht nur ein Service ausgeführt, sondern vielmehr sind viele verschiedene Services bei der Durchführung einer Aktivität beteiligt.

Die den KPIs zu Grunde liegenden Messwerte werden bereits von der aktuellen Implementierung erfasst, aber noch nicht aggregiert und aufbereitet dargestellt. Die Darstellung soll in den *Pipeline-View* integriert werden und neben der Darstellung der einzelnen Kennziffern auch erste Auswertungsmöglichkeiten bieten. Um Trends in der Ausführungszeit zu erkennen, sollen die letzten beiden Ausführungszeiten jeweils miteinander verglichen werden. Das Ergebnis soll in der Oberfläche geeigneten dargestellt werden, zum Beispiel mit Hilfe eines Pfeils der nach oben zeigt, falls die Laufzeit sich verlängert hat. Alternativ ist eine farbliche Markierung möglich. Dabei würde Grün eine Verbesserung und Rot eine Verschlechterung der Laufzeiten andeuten.

5.4. Zusammenfassung

In diesem Kapitel wurde ein konkretes Konzept zur Verwendung von BPMN als Modellierungssprache für CD vorgestellt. Dazu wurde als erstes eine Abbildung der BPMN Elemente auf die Elemente der Delivery Prozess Domäne erstellt. Dabei wurde festgestellt, das funktionale Abhängigkeiten und die Konfiguration der Aktivitäten auf zwei unterschiedliche Arten modelliert werden können. Beide Möglichkeiten wurden diskutiert und es wurde sich für die Umsetzung mit *ExtensionElements* entschieden. Der wichtigste Vorteil dieser Variante war die einfachere Modellierung der Abhängigkeiten.

Als nächstes wurde die Integration von BPMN in das SDS JARVIS konzipiert. Das Konzept beinhaltet sowohl neue Komponenten, als auch notwendige Anpassungen an bestehenden Software-Teilen. Hierbei wurden die notwendigen Erweiterungen aus Komponenten-

und Prozess-Sicht beschrieben.

Auf Architektur-Ebene werden für die Integration die zwei neuen Komponenten *BPMN Modeler* und *BPMN Model Service* benötigt. Der *BPMN Modeler* ermöglicht dem Benutzer neue Modelle zu entwickeln. Dabei werden verschiedene Mechanismen integriert, um die Eigenschaft *Benutzerunterstützung (E.8)* zu erfüllen (vgl. Abschnitt 4.1.2). Eine wichtige Komponente stellt dabei der *BpmnTemplateMapper* dar. Diese erstellt Aktivitäts-Vorlagen, die der Benutzer zur einfacheren Konfiguration der Aktivitäten verwenden kann. Die Komponente bedient sich dafür an den von JARVIS zur Verfügung gestellten Aktivitäts-Spezifikationen.

Die Komponente *BPMN Model Service* ist ein interner Service von JARVIS, der sowohl Schnittstellen für den *BPMN Modeler*, als auch für JARVIS selber implementiert. Der Service implementiert die *External Model Provider* Schnittstelle. Dies ermöglicht das Importieren von BPMN Modellen über die Oberfläche von JARVIS.

Eine weitere wichtige Erweiterung stellt der *BpmnModelParser* dar. Dieser ist Teil der *Pipeline Planner* Komponente und ist für die Umwandlung des externen BPMN Modells in ein äquivalentes internes Modell zuständig.

Nachdem das Integrations-Konzept von BPMN in JARVIS vorgestellt wurde, konnte ein Konzept entwickelt werden, das eine Symbiose zwischen BPM und CD herstellt, indem die BPM-Methoden *Prozess-Reporting* und BPM in JARVIS integriert wurden. Dabei wurden drei KPIs ausgewählt, deren zugrunde liegenden Metriken bereits erfasst werden. Das Konzept umfasst neben einer Erweiterung des *Pipeline View Services* auch eine Integration in die Oberfläche von JARVIS.

6. Integration in JARVIS

Inhalt

6.1. Umfang und Technologien 87
6.2. Realisierung des BPMN Modelers 88
6.2.1. Oberfläche 89
6.3. Generieren von Vorlagen 91
6.4. Datenmodellierung 93
6.5. Umgang mit BPMN Modellen 94
6.5.1. Verwaltung 94
6.5.2. Import 95
6.5.3. Ausführung 96
6.6. Prozess-Reporting und BAM 97

Dieses Kapitel befasst sich mit der Implementierung des in Kapitel 5 erstellten Konzepts. Anschließend wird die Implementierung evaluiert und diskutiert.

6.1. Umfang und Technologien

Aufgrund des begrenzten Umfangs der Arbeit kann das Konzept aus Kapitel 5 nicht mit allen Details implementiert werden, da die vollständige Implementierung weitreichende Erweiterungen von JARVIS nach sich ziehen würden. Aus diesem Grund implementiert die Komponente *BpmnModelParser* nicht die vollständige Abbildung aus Abschnitt 5.1.1, sondern verzichtet auf *Execution Preconditions* und manuelle Quality-Gates. Beide Konzepte fehlen in der aktuellen Implementierung von JARVIS. Da aber die grundsätzliche Modellierung beider Konzepte mit BPMN möglich ist, kann dies ohne großen Aufwand nachgeholt werden.

Des Weiteren beschränkt das Konzept von JARVIS den Geltungsbereich von Referenzen nicht auf die aktuelle Pipeline. Theoretisch sind somit beliebige Referenzen möglich. Da aber bisher kein konkretes Konzept vorliegt, wurde festgelegt, dass in BPMN Modellen nur Referenzen auf Elemente im aktuellen Prozess verwendet werden können.

Abgesehen von den oben genannten Einschränkungen konnte das vorgestellte Konzept umgesetzt werden.

Aufgrund der Tatsache, dass die im Konzept vorgestellten Komponenten in ein bestehendes System integriert werden, wurden die aus dem System bekannten Technologien verwendet. Das bedeutet, dass die beiden neuen Komponenten *BPMN Modeler* und *BPMN Model Service* als Microservices mit Spring Boot [Pivb] realisiert wurden.

Beide Services werden mittels Docker-Container [Doc] paketiert und können daher einfach ausgeliefert werden. Als Build-Management-Tool wird *Apache Maven (MVN)* [The] eingesetzt. Im Folgenden werden weitere Details bezüglich der Technologieauswahl gegeben.

BPMN Modeler

Der Modeler verwendet zur Darstellung und Modellierung der BPMN Diagramme das Framework *bpmn-js*. Dieses kann vorpaketiert über einen *Content Delivery Network (CDN)* Service oder über npm [Npm] bezogen werden. Da die Basiskonfiguration von *bpmn-js* für diesen Einsatzzweck nicht ausreicht, werden die benötigten Dateien über npm bezogen, weil dies eine einfache Konfiguration des Frameworks ermöglicht. Zusätzlich können die über npm bereitgestellten Dateien nicht einfach ausgeliefert werden, sondern müssen zuvor mit Hilfe des Präprozessors *browserify* [HB] in ausführbare Dateien umgewandelt werden. Dies führt jedoch zu der Problematik, dass der Build-Prozess des Moduls um npm spezifische Schritte erweitert werden muss. Diese wurden vollautomatisiert in den Build-Prozess von MVN integriert.

Aufgrund der Tatsache, dass der Modeler eine Spring Boot Applikation darstellt, wurde zusätzlich Spring MVC verwendet. Der View stellt dabei den Rahmen für die vom *bpmn-js* zur Verfügung gestellten Komponenten dar. Ein entsprechender Controller übernimmt die Kommunikation mit dem *BPMN Model Service*.

BPMN Model Service und Pipeline Planner

Der *BPMN Model Service* stellt zum einen REST-Schnittstellen für den *BPMN Modeler* zur Verfügung und zum anderen implementiert er verschiedene Kommandos. Diese werden per Reflection automatisch registriert und können von außen an den Service weitergereicht werden.

Der *Pipeline Planner* muss zusätzlich zu den bisher verwendeten Technologien JAXB verwenden, um die Umwandlung des im XML-Format vorliegenden BPMN Modells in ein Java-Objekt zu ermöglichen.

6.2. Realisierung des BPMN Modelers

Der BPMN Modeler basiert auf dem Open Source Javascript-Framework *bpmn-js* [cama] und kann daher gut in JARVIS integriert werden. In der Basiskonfiguration bietet das Framework:

- Eine Werkzeugleiste mit verschiedenen Elementen
- Ein Arbeitsblatt zum Modellieren

In dieser Konfiguration ist das Modellieren von Daten nicht möglich, da sowohl eine Oberflächenkomponente zur Eingabe der Daten, als auch eine Schnittstelle zum Einbinden von XSD-Dateien fehlt.

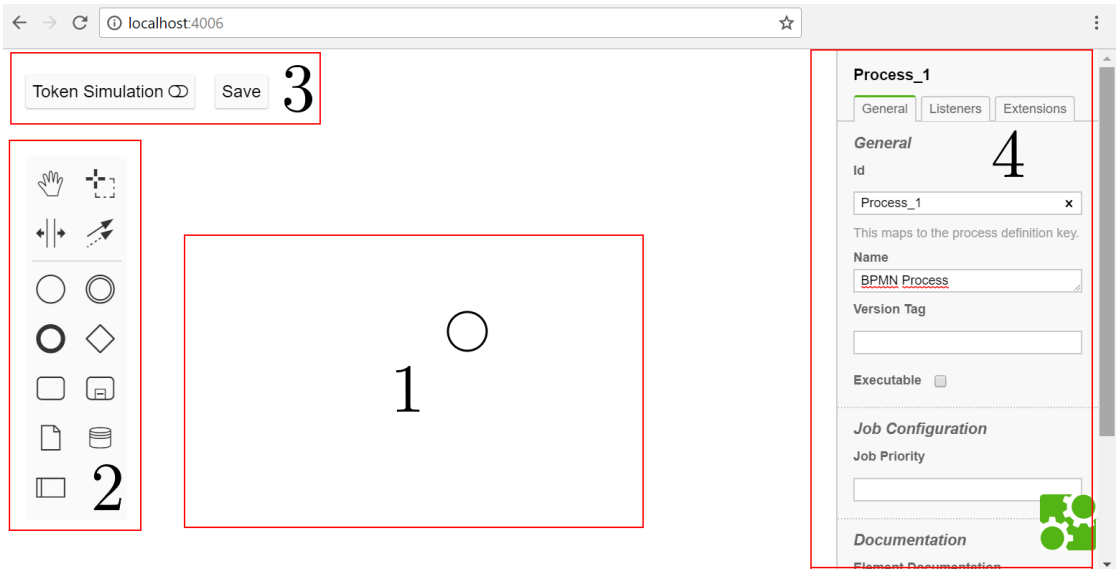

Abbildung 6.1.: Oberfläche des BPMN Modelers

Um den Funktionsumfang des Modelers erweitern zu können, müssen eigene oder fremde Plugins hinzugefügt werden. Mittels des Plugins *bpmn-js-properties-panel* wird der Oberfläche ein neuer Bereich zur Konfiguration von Aktivitäten, das *Properties Panel*, hinzugefügt. Es basiert auf der Camunda BPM Erweiterung mittels des BPMN-Elements *ExtensionElements* und erlaubt die Modellierung von funktionalen Abhängigkeiten.

Damit das resultierende BPMN valide ist, wird zusätzlich das Plugin *camunda-bpmn-moddle* benötigt. Dieses stellt sicher, dass das resultierende XML-Dokument mit dem Camunda spezifischen Namespace versehen wird. Dieser stellt alle benötigten Elemente für die Datenmodellierung bereit.

6.2.1. Oberfläche

Die Oberfläche wird weitestgehend von *bpmn-js* bereitgestellt (vgl. Abbildung 6.1). Sie wird im Browser dargestellt und kann in vier Bereiche unterteilt werden. Das Arbeitsblatt (1) befindet sich im Zentrum und links davon ist die Werkzeugleiste (2). Neben der Darstellung des Modells unterstützt das Arbeitsblatt den Benutzer bei der Auswahl von Elementen, indem es ein übersichtliches Menü mit Folgeelementen darstellt. Dabei werden nur Elemente angezeigt, die nach dem aktuell ausgewählten Element folgen dürfen (vgl. Abbildung 6.2).

Die Werkzeugleiste beinhaltet alle BPMN Elemente, die vom Modeler unterstützt werden. Der Modeler erlaubt das Platzieren von Elementen nur wenn dies erlaubt ist. Zum Beispiel darf auf ein Endereignis keine Aktivität folgen. In diesem Fall verhindert der Modeler das Kombinieren beider Elemente.

Der Menü-Bereich (3) ermöglicht das Aufrufen eines weiteren Plugins, der sogenannte *Token Simulator*. Dieses bereits existierende Plugin wurde in den Modeler integriert

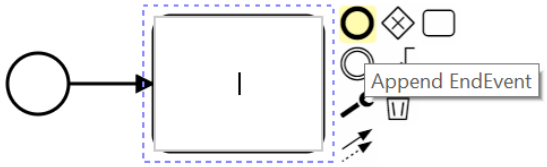

Abbildung 6.2.: Benutzerunterstützung beim Modellieren

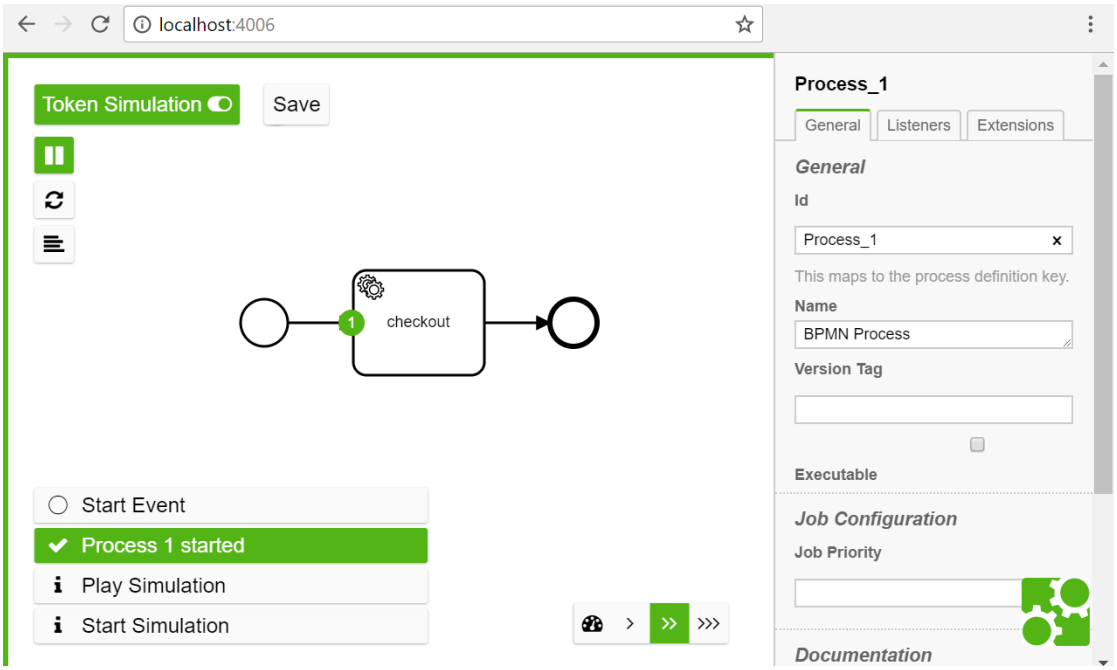

Abbildung 6.3.: Oberfläche des Token Simulators

um das Potenzial der Plugin-Architektur von *bpmn-js* zu zeigen. Der Simulator erlaubt die Visualisierung des in Abschnitt 2.2.1 eingeführten Token-Konzepts. Abbildung 6.3 zeigt die Oberfläche des Simulators. Über einen Start-Knopf wird eine Simulation der Ausführung des Modells gestartet und der Benutzer kann den Ablauf beobachten. Dadurch können zum Beispiel Fehler, wie die Verwechslung von einem *datenbasierten exklusiven Gateway* mit einem *parallelem Gateway*, erkannt und anschließend behoben werden.

Neben dem Button für den *Token Simulator* befindet sich der *Speichern-Button*. Durch Anklicken kann ein Name des Modells festgelegt werden und anschließend wird das Modell im XML-Format im *Artifact Storage* gespeichert.

Das *Properties Panel* (4) dient zur Konfiguration von BPMN-Elementen. Im Standardfall können zum Beispiel Namen oder Variablen vergeben werden. Wurde eine Vorlage für das Element konfiguriert, kann zusätzlich die zu verwendende Vorlage ausgewählt werden. Nachdem eine Vorlage ausgewählt wurde, werden im *Properties Panel* entsprechende Parameter zur Konfiguration angezeigt. Wird in der Vorlage ein Parameter als *required*

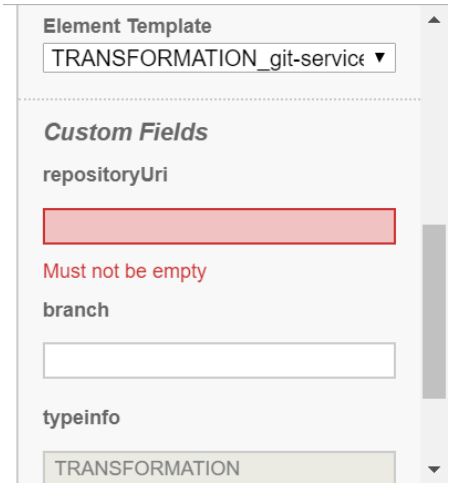

Abbildung 6.4.: Properties Panel mit ausgewählter Vorlage

konfiguriert, wird der Benutzer entsprechend über die Oberfläche darüber informiert (vgl. Abbildung 6.4). Neben Vorlagen für Aktivitäten, können auch andere Vorlagen, zum Beispiel für Phasen, erstellt werden.

6.3. Generieren von Vorlagen

Das Generieren der Vorlagen erfolgt auf Basis der, von JARVIS zur Verfügung gestellten, Spezifikationen der Aktivitäten. Diese können über die *Registry* im JSON-Format abgerufen werden. Abbildung 6.5 zeigt beispielhaft die Umwandlung einer Spezifikation in eine Vorlage. Die vollständige Spezifikation und Vorlage sind in Anhang A.2 und A.3 zu finden. Eine Spezifikation definiert alle von der Aktivität benötigten Parameter innerhalb eines *properties*-Bereichs. Die Vorlage zu einem Parameter beinhaltet die folgenden Informationen:

- *label*: Name des Parameters
- *type*: Boolean, String oder Dropdown
- *constraints*: Entweder ein regulärer Ausdruck oder *notEmpty*
- *binding*: Definiert welches Element im BPMN Modell verwendet werden soll
- *value*: Vordefinierter Wert des Parameters
- *choices*: Gibt die Auswahlmöglichkeiten eines Dropdown-Menüs an

String-Parameter, wie die *repositoryUri*, können ohne großen Aufwand in eine Vorlage übernommen werden. Wird in der Spezifikation das Schlüsselwort *oneOf* verwendet, müssen für jeden der angegebenen Typen entsprechende Parameter extrahiert werden.

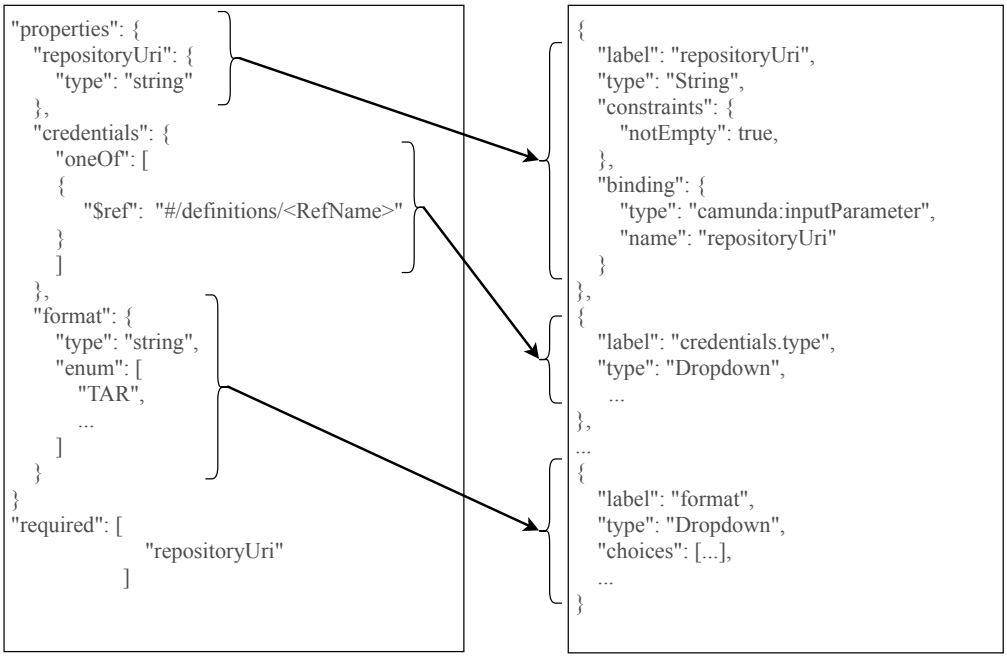

Abbildung 6.5.: Umwandlung einer Spezifikationen in eine Vorlage

In der Regel werden diese Typen im Definitionsbereich der Spezifikation angegeben. Die Implementierung sucht anhand der angegebenen Referenz den korrekten Typ aus dem Definitionsbereich. Diese Typen besitzen wiederum einen eigenen *properties*-Bereich. In Abbildung 6.5 werden zum Beispiel die *credentials* im Definitionsbereich definiert. Dieser beinhaltet die vollständige Typen-Definition, die unter anderem den Parameter *type* definiert.

Eine Spezifikation kann neben beliebigen Strings oder Referenzen auch eine vordefinierte Auswahl an Werten für einen Parameter festlegen. Dazu wird in der Spezifikation mit Hilfe des Schlüsselwortes *enum* eine entsprechende Liste von möglichen Werten angegeben. In der Vorlage wird dies als Dropdown-Menü dargestellt.

Jede Spezifikation kann mittels des Schlüsselwortes *required* zwingend benötigte Parameter markieren. In der Vorlage wird dies mit Hilfe eines notEmpty-constraints des Parameters widergespiegelt.

Neben den Parametern beinhaltet eine Vorlage zusätzlich die Informationen *typeinfo, command* und *service*. Die *typeinfo* gibt den Typ der Aktivität (Transformation, Assessment oder Quality-Gate) an. Die Schlüsselwörter *command* und *service* identifizieren eindeutig die jeweilige Aktivität. Alle drei Informationen werden im *Pipeline Planner* für die Umwandlung in das interne Datenmodell benötigt.

```
<bpmn2:serviceTask id="Task_10nopz6" name="checkout"
    camunda:modelerTemplate="git-service_checkout"
    typeinfo="TRANSFORMATION"
        service="git-service" command="checkout">
    <bpmn2:extensionElements>
      <camunda:inputOutput>
        <camunda:inputParameter name="prefix" />
        <camunda:inputParameter name="format" />
        <camunda:inputParameter
            name="credentials.username">bpmnPipeline</camunda:inputParameter>
        <camunda:inputParameter
            name="branch">master</camunda:inputParameter>
        <camunda:inputParameter
            name="credentials.password">SECRET</camunda:inputParameter>
        <camunda:inputParameter
            name="credentials.type">plain</camunda:inputParameter>
        <camunda:inputParameter name="repositoryUri">
            https://github.com/bpmnPipeline/cobertura-example.git
        </camunda:inputParameter>
      </camunda:inputOutput>
    </bpmn2:extensionElements>
    <bpmn2:incoming>SequenceFlow_149gqir</bpmn2:incoming>
    <bpmn2:outgoing>SequenceFlow_1c3yjuu</bpmn2:outgoing>
</bpmn2:serviceTask>
```

Quelltext 6.1: Beispiel einer Aktivität im XML-Format

6.4. Datenmodellierung

Der mit dem Modeler erstelle Prozess wird als XML-Datei von JARVIS weiterverarbeitet. Dabei handelt es sich um eine nach der BPMN Spezifikation gültige Repräsentation des Modells. Neben den regulären Namensräumen für BPMN wird zusätzlich der Namensraum `http://camunda.org/schema/1.0/bpmn` eingebunden. Dieser legt unter anderem das Element inputParameter fest [camb].

Quelltext 6.1 zeigt beispielhaft eine Aktivität im XML-Format. Das Beispiel veranschaulicht eine Aktivität, die ein Projekt mittels GIT herunterlädt. Zur späteren Weiterverarbeitung werden durch die Vorlage automatisch die Attribute *typeinfo, service* und *command* an der Aufgabe gesetzt. Im Beispiel ist gut zu sehen, an welchen Stellen der Namensraum von BPMN (*bpmn2*) und der von Camunda (*camunda*) verwendet wird. Die BPMN Spezifikation definiert das Element *extensionElements* und schränkt die Typen der Kind-Elemente nicht ein. Dadurch können die Elemente des Camunda-Namensraums verwendet werden, ohne dass das BPMN Modell ungültig wird.

Die Konfiguration einer Aktivität wird in einem *inputOutput*-Element definiert. Dieses fasst sowohl *inputParameter*, als auch *outputParamter* zusammen. Da die Eingaben der Aktivitäten konfiguriert werden soll, wird das Element *inputParameter* verwendet.

Das Attribut *name* identifiziert den Parameter, dessen Wert innerhalb des Elements als einfache Zeichenkette dargestellt wird. Auch funktionale Abhängigkeiten werden innerhalb des Elements mit Hilfe von Referenzen, wie bereits in Abschnitt 5.2.3 beschrieben, ausgedrückt.

6.5. Umgang mit BPMN Modellen

Der Umgang mit BPMN Modellen kann in JARVIS in vier Schritte unterteilt werden. Als erstes wird ein BPMN Modell über den *BPMN Modeler* erstellt. Die Realisierung dieses Schrittes wurde bereits in Abschnitt 6.2 eingeführt. Nachdem das Modell gespeichert wurde, kann es über das *Delivery System Management Tool*, die Oberfläche von JARVIS, abgerufen werden (vgl. Abschnitt 5.2.1). Anschließend kann ein gespeichertes Modell importiert und ausgeführt werden. Im Folgenden werden die dazu notwendigen Erweiterungen vorgestellt.

6.5.1. Verwaltung

Das *Delivery System Management Tool* wurde um eine neue Darstellung erweitert. Über den Menü-Punkt *External Models* (links) können nun alle gespeicherten BPMN Modelle eingesehen werden. Neben der Darstellung der Modelle kann über diese Oberfläche der *BPMN Modeler* aufgerufen werden (vgl. Abbildung 6.6). Wird ein existierendes Modell ausgewählt, wird der Modeler entsprechend initialisiert. Die Oberfläche ist so

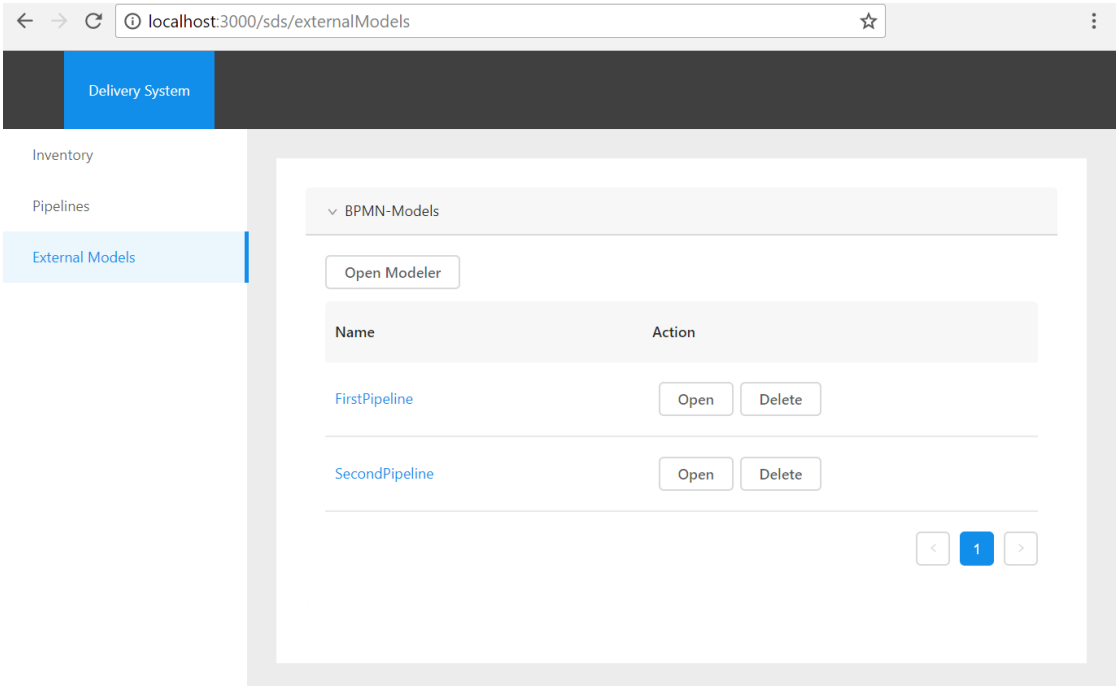

Abbildung 6.6.: Ansicht der externen Modelle

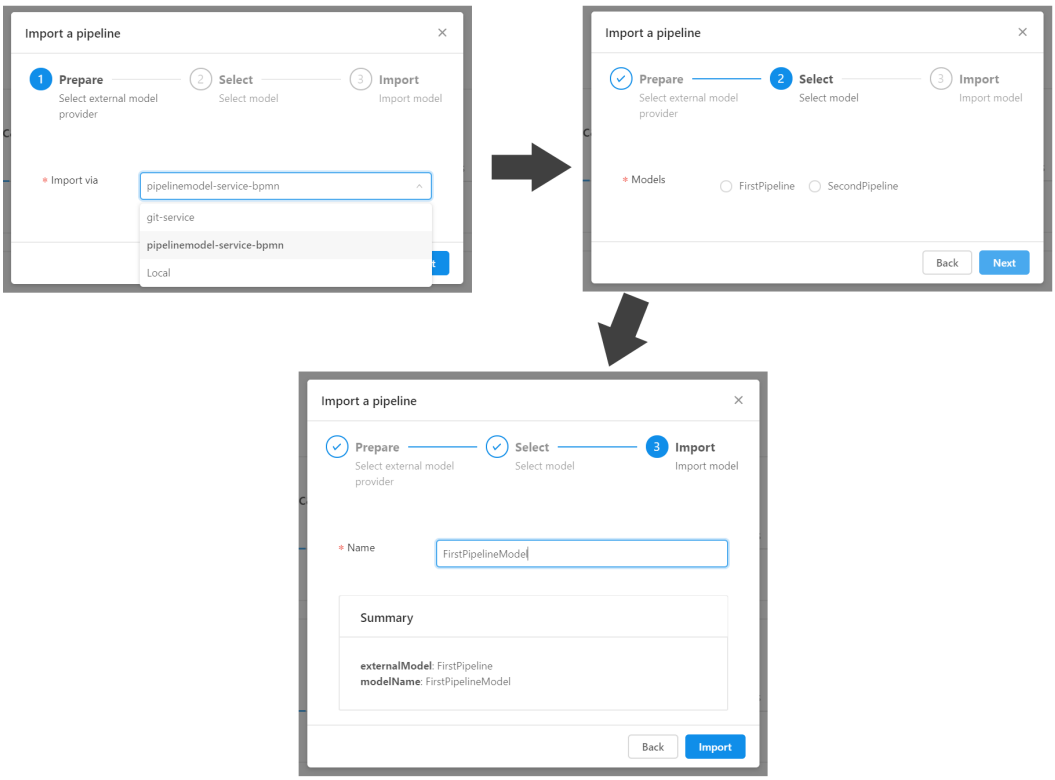

Abbildung 6.7.: Importieren von BPMN Modellen

gestaltet, dass der BPMN-spezifische Teil eingeklappt werden kann. Dadurch können theoretisch weitere Modellierungssprachen integriert werden, indem die einklappbaren Elemente untereinander angeordnet werden. Neben dem Öffnen von Modellen können diese zusätzlich wieder gelöscht werden. Wird der *BPMN Modeler* über die Oberfläche aufgerufen, wird dieser über das Gateway angesprochen. Das hat den Vorteil, dass sich bei der Adresse des Services nur der konkrete Pfad ändert. Die IP-Adresse und der Port bleiben unverändert. Dadurch kann die Microservice-Architektur vor dem Benutzer versteckt werden und es ist nicht ersichtlich, dass der *BPMN Modeler* einen eigenständigen Service darstellt.

6.5.2. Import

Die bereits existierende Oberfläche zum Importieren von Modellen erkennt automatisch, dass der *BPMN Model Service* ein *External Model Provider* ist und erlaubt daher ohne weitere Implementierungen das Importieren von BPMN Modellen (vgl. Abbildung 6.7). Das Importieren von Modellen erfolgt über einen in drei Schritte unterteilten Dialog.

Im ersten Schritt muss der *BPMN Model Service* als *Provider* ausgewählt werden. Dies sorgt dafür, dass das search-Kommando an den Service geschickt wird.

Dieser fragt alle im *Artifact Storage* gespeicherten Artefakt-Metadaten mit dem Content-Type *application/bpmn* ab. Anhand dieser Metadaten werden die Dateinamen der Modelle an die Oberfläche zurückgeliefert. Diese stellt die gefundenen Modelle dar und der Benutzer kann im zweiten Schritt eines der Modelle auswählen. Im letzten Schritt kann ein Name für die Pipeline festgelegt werden.

Nachdem der Dialog erfolgreich durchgeführt wurde, legt JARVIS ein *PipelineModel-Aggregate* an. Dabei ist es wichtig, dass das Aggregat nicht das konkrete BPMN Modell enthält, sondern nur eine Referenz auf das *importPipeline*-Kommando des *BPMN Model Service*. Dieses sucht zur Ausführungszeit der Pipeline das BPMN Modell über den Dateinamen im *Artifact Storage*. Dadurch wird ein nachträgliches Ändern des BPMN Modells ermöglicht.

6.5.3. Ausführung

Damit ein importiertes BPMN Modell ausgeführt werden kann, muss es zunächst in das interne Datenmodell umgewandelt werden. Das Konzept sieht dazu die neue Komponente *BpmnModelParser* vor (vgl. Abschnitt 5.2.2 und 5.2.3). Diese wurde implementiert und in die Komponente *Pipeline Planner* integriert.

Die aktuelle Implementierung der Komponente beachtet keine *Execution Preconditions* und *manuelle Quality-Gates*, da entsprechende Konzepte bisher im internen Datenmodell fehlen. Alle anderen Elemente des internen Datenmodells können aus dem BPMN Modell erstellt werden.

Das Umwandeln des als XML gespeicherten Modells in ein Java-Objekt wird mittels *JAXB* realisiert. Über ein MVN-Plugin werden aus den inkludierten BPMN XSDs entsprechende Java-Klassen generiert.

Aufgrund der Tatsache, dass die Modellierungssprache BPMN keine großen Einschränkungen bezüglich der Orchestrierung der Elemente macht, führt der *BpmnModelParser* zunächst Validierungen durch, die die Struktur des Modells sicherstellen. Das bedeutet, dass ein Prozess nur die Aktivität Subprozess beinhalten darf und keine anderen Aktivitäten. Die Subprozesse dürfen wiederum nur *Service-Aufgaben* beinhalten und nicht wieder Subprozesse. Wird das interne Datenmodell entsprechend erweitert, könnten solche Konstellationen jedoch mit BPMN konstruiert werden.

Im Anschluss kann aus dem BPMN Modell ein *StaticPipelineModel* erstellt werden. Dieses wird dann vom *Pipeline Planner* verwendet, um einen Ausführungsplan zu erstellen.

6.6. Prozess-Reporting und BAM

Aufgrund der Tatsache, dass die benötigten KPIs bereits erfasst wurden, konnte das Prozess-Reporting direkt in die Oberfläche integriert werden. Die Abfrage der KPIs erfolgt über den *View Service* (vgl. Abbildung 5.3). Abbildung 6.8 zeigt den neuen *KPIs-Tab*. Dieser kann für jedes importierte Modell angezeigt werden und aggregiert die KPIs aller bisherigen Ausführungen eines Prozesses als Durchschnittswerte. Die Darstellung ist als Tabelle organisiert und zeigt für jeden KPI:

- Name
- Durchschnittswert
- Trend-Indikator

Bezieht sich ein KPI auf eine bestimmte Aufgabe im Prozess wird dies in der Spalte *Name* angegeben. Der Trend-Indikator gibt an, wie sich der aktuelle Wert des KPIs vom Vorherigen unterscheidet. Die Farbe *Rot* gibt eine Verschlechterung, *Gelb* eine exakte Stagnierung und *Grün* eine Verbesserung des Durchschnittswertes an.

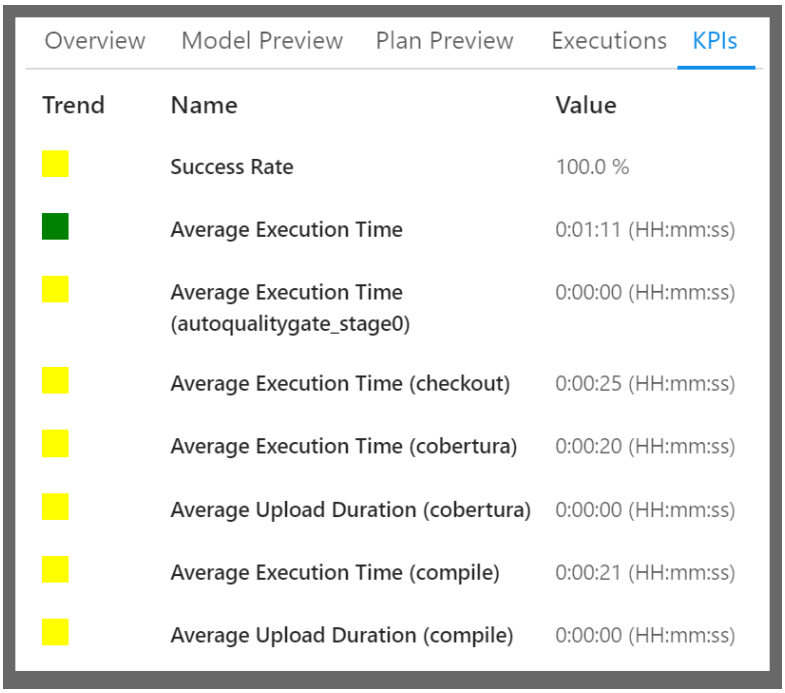

Abbildung 6.8.: KPIs-Tab eines Modells

7. Diskussion

Inhalt

7.1. Implementierung von CD mittels BPMN 99
7.1.1. Vergleich BPMN Modell mit Ausführungsplan 99
7.1.2. BPMN im Vergleich mit YAML als externe Modellierungssprache 101
7.2. BPM Konzepte zur Analyse und Optimierung von Delivery Prozessen . . 103
7.3. Selbstorganisierung 103
7.4. Integriertes Modellierungswerkzeug 104

In diesem Kapitel wird die Implementierung ausgewertet. Dazu werden die zu Beginn der Arbeit definierten Forschungsfragen (RQ-2 & RQ-3) bezüglich BPMN und BPM an konkreten Beispielen diskutiert.

7.1. Implementierung von CD mittels BPMN

Zur Beantwortung der Frage, ob sich BPMN zur Modellierung von CD eignet (RQ-2), wird ein konkreter Delivery Prozess mit BPMN modelliert. Dieses Modell wird mit dem von JARVIS generierten Ausführungsplan verglichen. Zusätzlich wird das Modell mit Hilfe eines äquivalenten Modells in der auf YAML basierenden DSL von JARVIS verglichen. Dabei wird geprüft, ob das BPMN Modell Vorteile bzw. Nachteile im Vergleich zu den anderen beiden Repräsentationen besitzt.

Für die Diskussion wird das Modell aus der Evaluierung von JARVIS [Dör18, S.126] verwendet. Das Modell ist in der YAML-Modellierungssprache in Anhang A.4 zu finden. Dieses beinhaltet alle wichtigen Elemente und ist komplex genug, um die Diskussion führen zu können.

7.1.1. Vergleich BPMN Modell mit Ausführungsplan

Der Ausführungsplan wird auf Basis des internen Datenmodells erstellt. Das bedeutet, dass das BPMN Modell theoretisch nahezu identisch mit dem Ausführungsplan sein sollte, da im BPMN Modell bereits alle benötigten Informationen enthalten sind.

Die Abbildungen 7.1 und 7.2 zeigen das BPMN Modell und den Ausführungsplan. Auffällig ist, dass das BPMN Modell Gateways zur expliziten Modellierung von Verzweigungen verwendet. Ein äquivalentes Element im Ausführungsplan fehlt. Dabei handelt es sich an dieser Stelle jedoch „nur“ um eine alternative Darstellung. Die Ausführung wird dadurch nicht beeinflusst.

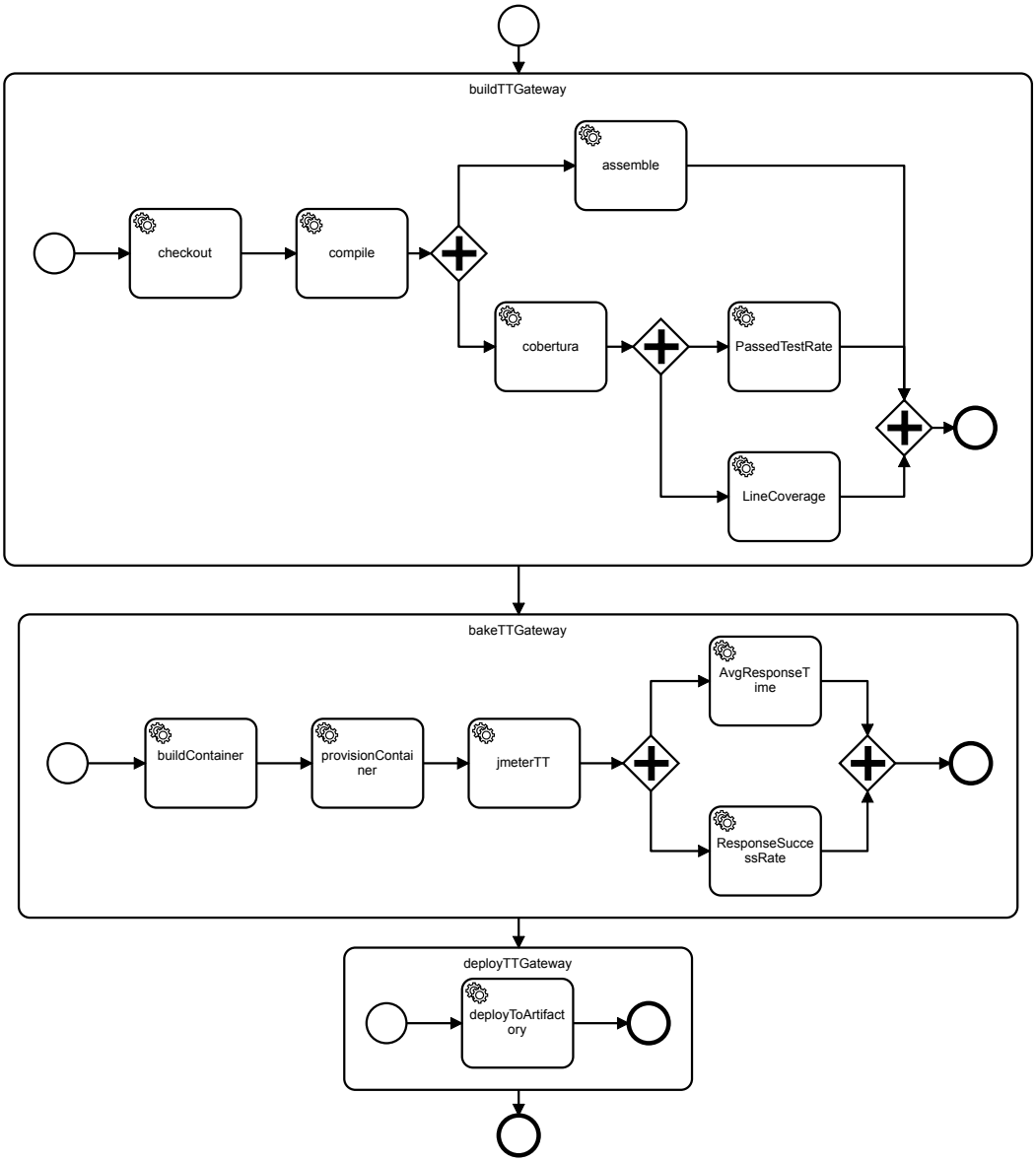

Abbildung 7.1.: Beispiel BPMN Modell

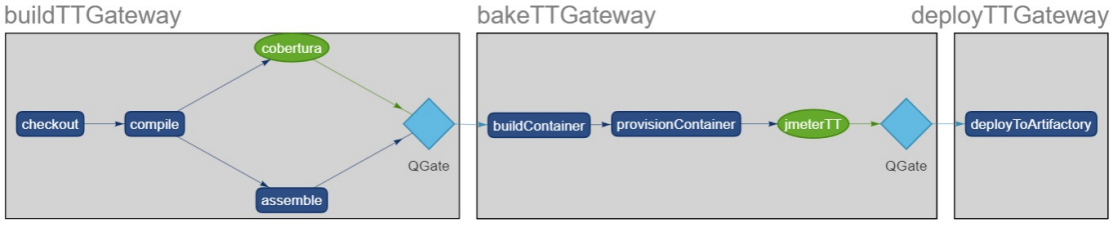

Abbildung 7.2.: Ausführungsplan des Modells

```
// BPMN
<camunda:inputParameter
    name="workspace">@Task_16tyqwt/workspace</camunda:inputParameter>

// YAML-DSL
parameters:
    workspace: "@repo"
dependsOn:
    - alias: repo
      ref: p://this/transformations/checkout/workspace
```

Quelltext 7.1: Vergleich der Modellierung von funktionalen Abhängigkeiten

Des Weiteren besitzt das BPMN Modell Elemente zur Darstellung des Anfangs und des Endes eines Prozesses oder einer Phase mittels Ereignissen. Das Ausführungsmodell besitzt keine speziellen Elemente für diesen Zweck. Dies ist auf die vereinfachte Darstellung des Ausführungsplan zurückzuführen, da dieser möglichst übersichtlich den Prozess darstellen soll.

Es kann festgehalten werden, dass das Umwandeln des externen BPMN Modells in das interne Datenmodell funktioniert. Alle Aktivitäten werden korrekt aus dem BPMN Modell extrahiert und auch die Reihenfolge der Aktivitäten ist bei beiden Modellen gleich. Die Aktivitäten sind in beiden Modellen den gleichen Phasen zugeordnet. Die Parallelität der Aktivitäten *cobertura* und *assemble* aus der ersten Phase wurde korrekt erkannt.

7.1.2. BPMN im Vergleich mit YAML als externe Modellierungssprache

Das BPMN Modell unterscheidet sich vom YAML Modell im Wesentlichen durch die spezifizierte grafische Darstellung der Elemente (vgl. Quelltext A.4). Das YAML Modell besitzt keine grafische Repräsentation. Beide Modelle beinhalten grundsätzlich die gleichen Informationen, wobei sich die YAML-DSL stark am internen Datenmodell orientiert. Im Gegensatz zur YAML-DSL verfügt BPMN bereits über Konzepte zur Abbildung von *Execution Preconditions*, die aktuell sowohl in der DSL als auch im internen Datenmodell fehlen.

Ein Vorteil des BPMN Modells ist, dass es den Ablauf des Delivery Prozesses veranschaulicht und somit direkt logische Abhängigkeiten erkennbar sind. Dadurch können bereits bei der Modellierung Optimierungen, wie das Parallelisieren von Aktivitäten, vorgenommen werden.

Ein wichtiger Unterschied zwischen beiden Modellen ist die Modellierung von funktionalen Abhängigkeiten (vgl. Quelltext 7.1). Benötigt eine Aktivität ein Artefakt von einer anderen Aktivität, wird diese im BPMN Modell direkt über eine vom Modeler generierte *Id* in der Eingabe-Konfiguration referenziert. Im Gegensatz dazu muss bei der YAML-DSL immer zuerst eine Abhängigkeit konfiguriert werden, bevor diese über einen Alias-Namen referenziert werden kann.

Neben funktionalen Abhängigkeiten werden auch *Quality-Gates* unterschiedlich dargestellt. In der YAML-DSL werden diese mittels sogenannten *Policies* konfiguriert. Wie im internen Datenmodell, kann in der DSL ein *Quality-Gate* beliebig viele *Policies* definieren. Bei BPMN wurde eine *eins-zu-eins*-Beziehung gewählt. Dadurch ist im grafischen Modell direkt über den Namen der Aktivität ersichtlich, welche Überprüfung das Gate vornimmt. Die Konfiguration beliebig vieler *Policies* ist zwar theoretisch möglich, würde aber das Modellieren deutlich komplexer gestalten, da die verwendete Erweiterung von BPMN keine Listen von komplexen Datentypen unterstützt. Eine mögliche Lösung wäre eine komplexe Modellierung über die bisherige Erweiterung von *Camunda BPM* mit Hilfe einfacher Key-Value-Paare. Dies ist jedoch nicht sehr benutzerfreundlich, da der Benutzer eigenständig spezielle Schlüssel verwenden müsste, damit das System jede *Policy* eindeutig identifizieren kann. Eine Alternative dazu wäre eine eigene Erweiterung des BPMN Modells um Listen von Parametern. Dies würde allerdings auch eine Erweiterung des Frameworks *bpmn-js* bedingen.

Abhängigkeiten zwischen Phasen

Während der Umsetzung des Konzepts wurde festgestellt, dass in einem BPMN Modell logische Abhängigkeiten zwischen Phasen modelliert werden. Dies geschieht durch die Verbindung von Subprozessen miteinander, mit Hilfe von Sequenzflüssen (vgl. Abbildung 7.1).

Diese Art von Abhängigkeiten kann sowohl im internen Datenmodell, als auch in YAML Modellen nicht modelliert werden. Im konkreten Beispiel führt das dazu, dass die Aktivität *deployToArtifactory* nach der Aktivität *jmeterTT* Phase ausgeführt werden soll, obwohl keine Abhängigkeiten zwischen den Aktivitäten der *bakeTTGateway* Phase und der *deployTTGateway* Phase modelliert wurden. Dadurch, dass JARVIS versucht einen optimalen Ausführungsplan zu erstellen, würde er die Aktivitäten der beiden Phasen *bakeTTGateway* und *deployTTGateway* parallel ausführen. Dies ist aber nicht sinnvoll, da die letzte Phase nur ausgeführt werden soll, wenn die vorherige Phase durchgeführt wurde.

Eine Modellierung als logische Abhängigkeit zwischen beiden Aktivitäten der Phasen ist an dieser Stelle nicht zielführend, da diese nicht anhand der Spezifikationen der Aktivitäten berechnet werden kann. Dies ist aber zwingend für den Erhalt der Eigenschaft *Selbstorganisierung* notwendig.

In der aktuellen Implementierung von JARVIS können daher Delivery Modelle sogenannte *Hinweise* beinhalten, die vordefinierte Abfolgen von Aktivitäten vorgeben (vgl. Zeile 1 aus Quelltext A.4). Diese *Hinweise* werden von JARVIS bei der Erstellung des Ausführungsplans beachtet.

Aufgrund der gleichen Intention von logischen Abhängigkeiten und Hinweisen ist an dieser Stelle die *Selbstorganisierung* nur bedingt gegeben. Ein besserer Lösungsweg wäre die Verwendung von *Execution Preconditions*. Diese sind bisher allerdings nicht implementiert worden.

Im Gegensatz zu YAML Modellen findet in BPMN Modellen eine explizite Modellierung von Abhängigkeiten statt. Daher beinhaltet ein BPMN Modell immer die notwendigen

Informationen zur Bestimmung von logischen Abhängigkeiten zwischen Aktivitäten, wodurch JARVIS diese nicht eigenständig berechnen muss. Phasen-Abhängigkeiten werden zu logischen Abhängigkeiten zwischen den am nächsten liegenden *Transformations* oder *Assessments* beider Phasen aufgelöst. Das bedeutet, dass die Aktivität *deployToArtifactory* eine logische Abhängigkeit zu *jmeterTT* besitzt und daher nach dieser ausgeführt werden muss. Dadurch muss an dieser Stelle kein *Hinweis* zur Modellierung der Abhängigkeit verwendet werden.

7.2. BPM Konzepte zur Analyse und Optimierung von Delivery Prozessen

Es wurde gezeigt, dass der BPM Regelkreis auf Delivery Prozesse angewendet werden kann (RQ-3). Dieser stellt ein Vorgehensmodell zur kontinuierlichen Verbesserung von Prozessen dar (vgl. Abschnitt 2.1.1). Innerhalb des Regelkreises werden die Methoden Prozess-Reporting und BAM zur stetigen Analyse und Optimierung der Prozesse verwendet.

Um die kontinuierliche Verbesserung von Delivery Prozessen zu ermöglichen, wurde die Oberfläche von JARVIS um einen *KPIs-Tab* erweitert. Dieser stellt die von JARVIS gesammelten Metriken übersichtlich dar. Basierend auf diesen Daten können Analysen durchgeführt und Trends erkannt werden. Die aktuelle Implementierung ermöglicht zum Beispiel die Identifikation der folgenden Probleme:

- Ein Prozess ist instabil (Erfolgsrate ist niedrig)
- Der Prozess wird immer langsamer (Durchlaufzeit steigt)
- Einzelne Aktivitäten sind langsam (Durchlaufzeit einer Aktivität hoch)

Durch das Erkennen von Trends kann diesen Problemen frühzeitig entgegengewirkt werden.

7.3. Selbstorganisierung

Die Möglichkeiten der *Selbstorganisierung* wurde bisher nicht vollständig betrachtet, da die Eigenschaft ein Alleinstellungsmerkmal von JARVIS ist. Des Weiteren müssen CD Sprachen Abhängigkeiten modellieren können, da sie nicht immer zuverlässig von JARVIS bestimmt werden können. Daher sollte auch BPMN die Möglichkeit besitzen logische und vor allem funktionale Abhängigkeiten zu modellieren.

Dennoch ermöglicht die *Selbstorganisierung* einen interessanten Anknüpfungspunkt an BPM. Wie bereits beschrieben, ermöglicht die automatische Erkennung von Abhängigkeiten das Anreichern von Aktivitäten um Informationen, die nur für die Ausführung relevant sind. Dies deckt sich mit dem Verständnis von *fachlichen* und *technischen Modellen*, die bereits mit BPM in Abschnitt 2.1.2 eingeführt wurden. Das *fachliche Modell* ist in diesem Fall die grafische Repräsentation des Modells ohne die Konfiguration der Aktivitäten. Durch die manuelle Konfiguration der Aktivitäten und der automatischen

Anreicherung dieser um funktionale Abhängigkeiten durch die Planner-Komponenten wird das *fachliche Modell* in ein *technisches Modell* semi-automatisch überführt. Dieser Transformationsschritt wirkt sich nur auf die Konfiguration der Aktivitäten aus und erfordern nicht die Erweiterung des Modells um neue Elemente oder die Umstrukturierung einzelner Elemente. Daher ist die grafische Repräsentation des fachlichen und technischen Modells eines Delivery Prozesses fast identisch. Beide Modelle unterscheiden sich allein in der Konfiguration der Aktivitäten.

Im Falle des Delivery Prozesses kann mit Hilfe von JARVIS somit ein funktionierendes *round-tripping* erreicht werden (vgl. Abschnitt 2.1.2). Dies ist aber nur durch die *Selbstorganisierung* von JARVIS möglich.

7.4. Integriertes Modellierungswerkzeug

Die bisherige Implementierung von JARVIS besitzt kein integriertes Modellierungswerzeug für eine externe Modellierungssprache. Mit BPMN wurde erstmals ein solches Werkzeug integriert. Neben der *Selbstorganisierung* ist dies ein weiteres Unterscheidungsmerkmal von JARVIS zu anderen SDSs. Das Ziel der Integration war es, den Benutzern schon während der Modellierungszeit beim Erstellen der Modelle zu unterstützen, da dies bei den aktuellen Systemen fehlt. Der *BPMN Modeler* erleichtert sowohl das Modellieren des Prozessablaufs, als auch die Konfiguration der Aktivitäten. Durch die automatische Generierung von Vorlagen muss der Benutzer nur eine passende Vorlage auswählen und ausfüllen. Die Vorlage beinhaltet bereits alle möglichen Konfigurationsparameter, die bei Bedarf ausgefüllt werden können.

Die grafische Modellierung wird dahingehend unterstützt, dass dem Benutzer das Platzieren und Kombinieren nur von nach der BPMN Spezifikation zugelassenen Elementen erlaubt wird. Dadurch können viele Fehler bei der Modellierung verhindert werden. Problematisch an dieser Stelle ist, dass BPMN insgesamt sehr viele unterschiedliche Elemente anbietet, die im Delivery Prozess nicht verwendet werden dürfen.

Dennoch konnte mit der Integration eines BPMN spezifischen Modellierungswerkzeugs gezeigt werden, dass ein solches Werkzeug das Modellieren von Modellen stark vereinfacht und daher auch die Modellierungszeit verkürzt. Damit erfüllt JARVIS die Eigenschaft *Benutzerunterstützung (E.8)*.

8. Zusammenfassung und Ausblick

Inhalt

8.1. Zusammenfassung 105
8.2. Ausblick 106

8.1. Zusammenfassung

In dieser Arbeit wurde gezeigt, dass sich BPMN als Modellierungssprache für Delivery Prozesse eignet. Dazu wurde als erstes das notwendige Wissen vermittelt. Dabei wurde BPMN als Teil des BPM vorgestellt und wichtige Konzepte eingeführt. Zur Einführung von CD wurde zunächst der Software Delivery Prozess und weitere Begriffe eingeführt.

Als nächstes wurde anhand aktueller Probleme der Bedarf an neuen und besseren Lösungen motiviert. Unter der Beachtung von *Value Streams* konnte eine Verwendung von BPMN als Modellierungssprache für Delivery Prozesse gerechtfertigt werden.

Anschließend wurden wichtige Eigenschaften von CD Sprachen und Systemen identifiziert. Diese Eigenschaften wurden im Anschluss für bereits existierende Sprachen und Systeme geprüft und es konnte festgestellt werden, dass aktuelle Implementierungen viele dieser Eigenschaften nur teilweise erfüllen. Am schlechtesten haben die Sprachen bei der Abbildung der Delivery Prozess Domäne und bei der Modellierung von Flüssen und Entscheidungen abgeschnitten. Bei den Systemen wurde eine schlechte Benutzerunterstützung als größtes Problem aktueller Implementierungen identifiziert. Aufgrund der hohen Flexibilität von BPMN konnte gezeigt werden, dass die Sprache alle untersuchten Eigenschaften erfüllt (RQ-1).

Damit BPMN als Modellierungssprache für Delivery Prozesse eingesetzt werden kann, wurde eine konkrete Abbildung von BPMN Elementen auf die Delivery Prozess Domäne vorgestellt. Dadurch konnten die für die Implementierung notwendigen Elemente der BPMN identifiziert werden.

Nachdem die theoretische Eignung von BPMN gezeigt wurde, konnte ein Konzept für die Integration der Sprache in das SDS JARVIS erstellt werden. Dieses beschreibt sowohl die neu benötigten Komponenten, als auch die neuen Prozesse innerhalb von JARVIS. Da vor allem die Benutzerunterstützung negativ bei den existierenden System aufgefallen ist, wurde beim Konzept besonders auf diesen Aspekt Wert gelegt. Dies führte dazu, dass ein eigenständiges Modellierungswerkzeug für BPMN Modelle in JARVIS integriert wurde, welches eine grafische Modellierung von Delivery Prozessen ermöglicht und den Benutzer dabei unter anderem mit Hilfe von Vorlagen unterstützt. Damit BPMN von JARVIS als

externe Modellierungssprache genutzt werden kann, wurde ein Konzept zur Umwandlung von BPMN Modellen in das interne Datenmodell von JARVIS entwickelt. Anschließend wurde gezeigt, wie der BPM Regelkreis und das Prozess-Reporting und das BAM zur Analyse und Optimierung von Delivery Modellen eingesetzt werden kann.

Die eingeführten Konzepte konnten anschließend für JARVIS implementiert werden. Die Implementierung zeigt, dass sich BPMN zur Implementierung von CD eignet (RQ-2) und dass sich BPM Methoden auch für CD verwenden lassen (RQ-3).

Als letztes wurden die gewonnen Erkenntnisse aus dem Konzept und der Implementierung ausgewertet. Dazu wurde ein bereits existierender Delivery Prozess mit BPMN modelliert und mit dem passenden Modell in der YAML-DSL und dem von JARVIS generierten Ausführungsplan verglichen. Dabei konnte festgestellt werden, dass die drei Prozess-Repräsentationen zwar Unterschiede aufweisen, aber sich nur in Details unterscheiden, die nicht die Ausführung beeinflussen.

Während der Integration von BPMN in JARVIS ist aufgefallen, dass das von JARVIS verwendete Konzept der *Selbstorganisierung* zum Erreichen eines *round-trips* führt, da das BPMN theoretisch nur ein *fachliches Modell* sein muss und von JARVIS automatisch in ein *technisches Modell* überführt werden kann.

8.2. Ausblick

Die aktuelle Implementierung ermöglicht viele interessante Erweiterungen. Durch die erweiterbare Architektur von JARVIS könnten neue Modellierungssprachen eingeführt werden. Spezielle Modellierungswerkzeuge könnten, ähnlich dem *BPMN Modeler*, als zusätzliche Komponenten in das System integriert werden. Aus diesem Grund wurde bereits die Oberfläche so gestaltet, dass weitere externe Modelle angezeigt und verwaltet werden können.

Die Plugin-Architektur des Modelers erlaubt eine einfache Erweiterung seiner Oberfläche um neue Funktionalitäten. Da die BPMN Spezifikation theoretisch auch eigene Elemente erlaubt, könnten anstelle der Vorlagen für Service-Aktivitäten auch direkt eigene Aktivitäten generiert werden. Diese könnten mit einer eigenen Darstellung die Modelle ausdrucksstärker machen. Neben dem Hinzufügen neuer Elemente wäre das Entfernen von nicht erlaubten Elementen aus der Werkzeugleiste denkbar. Dadurch würden zum Beispiel Ereignisse oder spezielle Aktivitäten erst gar nicht dem Modell hinzugefügt werden können.

Eine große Stärke von BPMN ist die Modellierung von Entscheidungen mittels *Gateways*. Das Potenzial von BPMN wird an dieser Stelle noch nicht vollständig ausgenutzt, da diese zum Beispiel nicht zwischen Phasen verwendet werden dürfen.

Des Weiteren werden *Ereignisse* in der aktuellen Implementierung nicht genutzt, da das interne Datenmodell keine entsprechenden Konzepte bietet. Die Verwendung von zum Beispiel angehefteten Fehler-Ereignissen, könnte den Ablauf eines Prozesses expliziter machen. Aktuell werden Fehler-Fälle nicht modelliert, sondern automatisch vom System behandelt.

JARVIS bietet zur Zeit keine Möglichkeit zur Abbildung von manuellen Aufgaben

innerhalb eines Delivery Prozesses. Würde das System entsprechend erweitert werden, könnte BPMN diese mittels einer Benutzer- oder Manuellen-Aufgabe abbilden.

Die Implementierung des KPIs-Tabs könnte um weitere Funktionen erweitert werden. Eine zeitliche Darstellung der KPIs mit Hilfe von Diagrammen würde den Benutzern zum Beispiel bei der Analyse der Laufzeiten eines Prozesses helfen. Eine Benachrichtigungsfunktionalität bei Unterschreitung bestimmter Grenzen der KPIs würde zusätzlich für schnelles Feedback sorgen.

A. Anhang

A.1. Beispiel einer Skript-Aufgabe in BPMN

```
<scriptTask id="task" name="Groovy Script" scriptFormat="groovy">
      <script>
        <![CDATA[
        sum = 0

        for ( i in inputArray ) {
          sum += i
        }

        println "Sum: " + sum
        ]]>
      </script>
</scriptTask>
```

Quelltext A.1: scriptTaskExample.groovy

A.2. Spezifikation der GIT-Checkout Aktivität

```
{
    "name": "checkout",
    "paramSchema": {
        "$schema": "http://json-schema.org/draft-04/schema#",
        "title": "Git Checkout Properties",
        "type": "object",
        "additionalProperties": false,
        "properties": {
            "repositoryUri": {
                "type": "string"
            },
            "credentials": {
                "oneOf": [
                    {
                        "$ref":
                           "#/definitions/UsernamePasswordCredentials"
                    }
                ]
            },
            "format": {
```

```
                "type": "string",
                "enum": [
                    "TAR",
                    "TAR_GZ",
                    "TAR_BZ_2",
                    "ZIP",
                    "TAR_XZ"
                ]
            }
        },
        "required": [
            "repositoryUri"
        ],
        "definitions": {
            "UsernamePasswordCredentials": {
                "type": "object",
                "additionalProperties": false,
                "properties": {
                    "type": {
                        "type": "string",
                        "enum": [
                            "plain"
                        ],
                        "default": "plain"
                    },
                    "username": {
                        "type": "string"
                    },
                    "password": {
                        "type": "string"
                    }
                },
                "title": "plain",
                "required": [
                    "type",
                    "username",
                    "password"
                ]
            }
        }
    },
    "returnSchema": {...},
    "type": "TRANSFORMATION",
    "expansionJsonPath": null,
    "teardownCallback": false
}
```

Quelltext A.2: Spezifikation der GIT-Checkout Aktivität

A.3. Vorlage der GIT-Checkout Aktivität

```
{
    "name": "TRANSFORMATION_git-service_checkout",
    "id": "git-service_checkout",
    "appliesTo": [
        "bpmn:ServiceTask"
    ],
    "properties": [
        {
            "label": "repositoryUri",
            "type": "String",
            "editable": true,
            "constraints": {
                "notEmpty": true
            },
            "binding": {
                "type": "camunda:inputParameter",
                "name": "repositoryUri"
            }
        },
        {
            "label": "branch",
            "type": "String",
            "editable": true,
            "constraints": {
                "notEmpty": false
            },
            "binding": {
                "type": "camunda:inputParameter",
                "name": "branch"
            }
        },
        {
            "label": "credentials.type",
            "type": "Dropdown",
            "choices": [
                {
                    "name": "plain",
                    "value": "plain"
                }
            ],
            "editable": true,
            "constraints": {
                "notEmpty": true
            },
            "binding": {
                "type": "camunda:inputParameter",
                "name": "credentials.type"
```

```
48              }
49          },
50          {
51              "label": "credentials.username",
52              "type": "String",
53              "editable": true,
54              "constraints": {
55                  "notEmpty": true
56              },
57              "binding": {
58                  "type": "camunda:inputParameter",
59                  "name": "credentials.username"
60              }
61          },
62          {
63              "label": "credentials.password",
64              "type": "String",
65              "value": null,
66              "choices": null,
67              "editable": true,
68              "constraints": {
69                  "notEmpty": true
70              },
71              "binding": {
72                  "type": "camunda:inputParameter",
73                  "name": "credentials.password"
74              }
75          },
76          {
77              "label": "prefix",
78              "type": "String",
79              "editable": true,
80              "constraints": {
81                  "notEmpty": false
82              },
83              "binding": {
84                  "type": "camunda:inputParameter",
85                  "name": "prefix"
86              }
87          },
88          {
89              "label": "format",
90              "type": "Dropdown",
91              "choices": [
92                  {
93                      "name": "TAR",
94                      "value": "TAR"
95                  },
96                  {
```

```
97                      "name": "TAR_GZ",
98                      "value": "TAR_GZ"
99                  },
100                 {
101                     "name": "TAR_BZ_2",
102                     "value": "TAR_BZ_2"
103                 },
104                 {
105                     "name": "ZIP",
106                     "value": "ZIP"
107                 },
108                 {
109                     "name": "TAR_XZ",
110                     "value": "TAR_XZ"
111                 }
112             ],
113             "editable": true,
114             "constraints": {
115                 "notEmpty": false
116             },
117             "binding": {
118                 "type": "camunda:inputParameter",
119                 "name": "format"
120             }
121         },
122         {
123             "label": "typeinfo",
124             "type": "String",
125             "value": "TRANSFORMATION",
126             "editable": false,
127             "constraints": {
128                 "notEmpty": false
129             },
130             "binding": {
131                 "type": "property",
132                 "name": "typeinfo"
133             }
134         },
135         {
136             "label": "service",
137             "type": "String",
138             "value": "git-service",
139             "editable": false,
140             "constraints": {
141                 "notEmpty": false
142             },
143             "binding": {
144                 "type": "property",
145                 "name": "service"
```

```
146                 }
147             },
148             {
149                 "label": "command",
150                 "type": "String",
151                 "value": "checkout",
152                 "editable": false,
153                 "constraints": {
154                     "notEmpty": false
155                 },
156                 "binding": {
157                     "type": "property",
158                     "name": "command"
159                 }
160             },
161             {
162                 "label": "name",
163                 "type": "String",
164                 "value": "checkout",
165                 "editable": false,
166                 "constraints": {
167                     "notEmpty": false
168                 },
169                 "binding": {
170                     "type": "property",
171                     "name": "name"
172                 }
173             }
174         ]
175 }
```

Quelltext A.3: Vorlage der GIT-Checkout Aktivität

A.4. Vollständiges Modell

```
1  planner:
2    expand: true
3    hints:
4      - type: schedule
5        target: deployToArtifactory
6        after: jmeter
7
8  stages:
9    - name: buildTTGateway
10     transformations:
11       - checkout
12       - compile
13       - assemble
```

```
14    - name: bakeTTGateway
15      transformations:
16        - buildContainer
17        - provisionContainer
18    - name: deployTTGateway
19      transformations:
20        - deployToArtifactory
21
22  transformations:
23    - name: checkout
24      service: git-service
25      command: checkout
26      parameters:
27        repositoryUri:
             https://gitlab.example.com/ttgateway/ttgateway.git
28        branch: pipeline
29
30    - name: compile
31      service: maven-service
32      command: compile
33      dependsOn:
34        - alias: repo
35          ref: p://this/transformations/checkout/workspace
36      parameters:
37        workspace: "@repo"
38
39    - name: assemble
40      service: maven-service
41      command: assemble
42      parameters:
43        workspace: "@repo"
44        classes: "@compile"
45      dependsOn:
46        - alias: repo
47          ref: p://this/transformations/checkout/workspace
48        - alias: compile
49          ref: p://this/transformations/compile/classes
50
51    - name: buildContainer
52      service: docker-service
53      command: buildTTGateway
54      parameters:
55        workspace: "@repo"
56        javaPackage: "@assemble"
57      dependsOn:
58        - alias: repo
59          ref: p://this/transformations/checkout/workspace
60        - alias: assemble
61          ref: p://this/transformations/assemble/assembly
```

```
62
63    - name: provisionContainer
64      service: docker-service
65      command: provisionTTGateway
66      parameters:
67        wildflyImageName: "@buildContainer/wildflyImage"
68        databaseImageName: "@buildContainer/databaseImage"
69        keycloakImageName: "@buildContainer/keycloakImage"
70      dependsOn:
71        - alias: buildContainer
72          ref: p://this/transformations/buildContainer
73
74    - name: deployToArtifactory
75      service: maven-service
76      command: deploy
77      parameters:
78        workspace: "@repo"
79        assemblies: "@assemble"
80        classes: "@compile"
81      dependsOn:
82        - alias: repo
83          ref: p://this/transformations/checkout/workspace
84        - alias: assemble
85          ref: p://this/transformations/assemble/assembly
86        - ref: p://this/transformations/compile/classes
87          alias: compile
88        - ref: p://this/assessments/jmeter
89
90  assessments:
91    - name: cobertura
92      service: maven-service
93      command: cobertura
94      parameters:
95        workspace: "@repo"
96        classes: "@compile"
97      dependsOn:
98        - alias: repo
99          ref: p://this/transformations/checkout/workspace
100       - alias: compile
101         ref: p://this/transformations/compile/classes
102
103   - name: jmeter
104     service: jmeter-service
105     command: jmeterTT
106     parameters:
107       jmeterProperties:
108         wildfly_host: "@provision/wildflyIp"
109         wildfly_port: "@provision/wildflyPort"
110         keycloak_host: "@provision/keycloakIp"
```

```
111         keycloak_port: "@provision/keycloakPort"
112       testPattern:
113         - "**ttgateway-ftest-2.0.jmx"
114       jmeterTestsRef: "@repo"
115     dependsOn:
116       - alias: repo
117         ref: p://this/transformations/checkout
118       - alias: provision
119         ref: p://this/transformations/provisionContainer
120
121 qualityGates:
122   - strategy: auto
123     policies:
124       - name: PassedTestRate
125         interpretation: threshold-
126         actualValue: passedRate
127         setPoint: 1
128         assessmentRef: p://this/assessments/cobertura
129
130       - name: LineCoverage
131         interpretation: threshold-
132         actualValue: lineCoverage
133         setPoint: 0
134         assessmentRef: p://this/assessments/cobertura
135
136       - name: AvgResponseTime
137         interpretation: threshold+
138         actualValue: avgResponseTimeMs
139         setPoint: 400
140         assessmentRef: p://this/assessments/jmeter
141
142       - name: ResponseSuccessRate
143         interpretation: threshold-
144         actualValue: successRate
145         setPoint: 0.26
146         assessmentRef: p://this/assessments/jmeter
```

Quelltext A.4: Beispiel einer YAML-Pipeline

Literatur

[Abe14] A. Abel. *ThoughtWorks Open Sources Go, a CD Tool*. 2014. URL: https://www.infoq.com/news/2014/03/thoughtworks-go-snap (besucht am 23. 04. 2018) (zitiert auf Seite 30).

[All09a] T. Allweyer. *BPMN 2.0 – Business Process Model and Notation: Einführung in den Standard für die Geschäftsprozessmodellierung*. 2. Norderstedt: Books on Demand, 2009. ISBN: 978-3-8391-2134-4 (zitiert auf Seite 18).

[All09b] T. Allweyer. „Kollaborationen, Choreographien und Konversationen in BPMN 2.0“. In: *kurze-prozesse.de* (2009), S. 25. URL: http://kurze-prozesse.de/blog/wp-content/uploads/2009/06/kollaborationen-choreographien-und-konversationen-in-bpmn-20.pdf (zitiert auf den Seiten 17, 18).

[Arm16] B. Armijo. *Declarative vs Imperative (or, do we really need a model for deployment)*. 2016. URL: https://opsani.com/devops/declarative-vs-imperative-really-need-model-deployment/ (besucht am 30. 04. 2018) (zitiert auf Seite 27).

[Bar+10] J. Bartkowiak u. a. *Geschäftsinnovationen durch BPM-Technologien und SOA*. Techn. Ber. BITKOM, 2010, S. 52. URL: https://www.bitkom.org/Publikationen/2010/Leitfaden/Geschaeftsinnovationen-durch-BPM-Technologien-und-SOA-Leitfaden-und-Nachschlagewerk/bitkom-leitfaden-bpm.pdf (zitiert auf Seite 5).

[Ben16] Benzvan. *Feature Proposal: Webhooks and notification*. 2016. URL: https://github.com/gocd/gocd/issues/2287 (besucht am 11. 02. 2018) (zitiert auf Seite 48).

[Blo16] J. Bloemberg. *HTTP API Resource*. 2016. URL: https://github.com/aequitas/concourse-http-api-resource (besucht am 12. 02. 2018) (zitiert auf Seite 52).

[BWZ15] L. Bass, I. Weber und L. Zhu. *DevOps: A Software Architect's Perspective*. New York: Addison-Wesley, 2015. ISBN: 978-0-13-404984-7 (zitiert auf Seite 41).

[Che17] L. Chen. „Continuous Delivery: Overcoming adoption challenges“. In: *Journal of Systems and Software* 128 (2017), S. 72–86. ISSN: 01641212. DOI: 10.1016/j.jss.2017.02.013. URL: http://dx.doi.org/10.1016/j.jss.2017.02.013 (zitiert auf Seite 33).

[Clo] CloudBees. *Codeship Developer Resources*. URL: https://documentation.codeship.com/ (besucht am 09. 02. 2018) (zitiert auf den Seiten 38, 48).

[CSA15] G. G. Claps, R. B. Svensson und A. Aurum. „On the journey to continuous deployment: Technical and social challenges along the way“. In: *Information and Software Technology* 57 (2015) (zitiert auf Seite 33).

[Dör18] J. S. Döring. „An Architecture for Self-Organizing Continuous Delivery Pipelines“. Master Thesis. RWTH Aachen, 2018. URL: https://www.swc.rwth-aachen.de/thesis/architecture-self-organizing-continuous-delivery-pipelines/ (zitiert auf den Seiten 21, 31, 36, 73, 84, 99).

[EAB14] EABPM. *BPM CBOK® - Business Process Management BPM Common Body of Knowledge, Version 3.0, Leitfaden für das Prozessmanagement*. 2. Aufl. Gießen: Dr. Götz Schmidt, 2014, S. 542 (zitiert auf den Seiten 3–5, 7, 8).

[Fow06] M. Fowler. *Continuous Integration*. 2006. URL: https://www.martinfowler.com/articles/continuousIntegration.html (besucht am 24. 01. 2018) (zitiert auf Seite 25).

[Fow12] M. Fowler. *SnowflakeServer*. 2012. URL: https://martinfowler.com/bliki/SnowflakeServer.html (besucht am 24. 01. 2018) (zitiert auf Seite 26).

[Fow14] M. Fowler. *Open-Sourcing ThoughtWorks Go*. 2014. URL: https://martinfowler.com/articles/go-interview.html (besucht am 23. 04. 2018) (zitiert auf Seite 30).

[FR14] J. Freund und B. Rücker. *Bpmn 2.0*. 2014, S. 296. ISBN: 978-3-446-44292-4. DOI: doi:10.3139/9783446429871 (zitiert auf den Seiten 3, 4, 7, 10, 16).

[Ger+91] A. Geraci u. a. *IEEE Standard Computer Dictionary. A Compilation of IEEE Standard Computer Glossaries*. 1991. DOI: 10.1109/IEEESTD.1991.106963 (zitiert auf Seite 20).

[HB] J. Halliday und Beitragende. *browserify*. URL: http://browserify.org/ (besucht am 15. 04. 2018) (zitiert auf Seite 88).

[HF10] J. Humble und D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. 2010, S. 497. ISBN: 978-0-321-60191-9 (zitiert auf den Seiten 1, 21–26, 34, 39).

[Jd] Jenkinsci-docs@googlegroups.com. *Jenkins User Handbook*. Techn. Ber. URL: https://jenkins.io/user-handbook.pdf (zitiert auf Seite 28).

[Kim+16] G. Kim u. a. *DevOps Handbook*. 2016, S. 480 (zitiert auf den Seiten 20, 25, 33, 34).

[Kom11] A. Komus. *BPM Best Practice*. Springer-Verlag Berlin Heidelberg, 2011, S. 236. ISBN: 978-3-642-16724-9. DOI: 10.1007/978-3-642-16725-6. URL: http://link.springer.com/10.1007/978-3-642-16725-6 (zitiert auf den Seiten 3, 4, 6).

[Kum16] R. Kumar. *CONCOURSE-CI*. 2016. URL: http://techforgeek.com/concourse-ci/ (besucht am 23. 04. 2018) (zitiert auf Seite 30).

[Lab18] S. Labourey. *Codeship and CloudBees Join Forces!* 2018. URL: https://www.cloudbees.com/blog/codeship-and-cloudbees-join-forces (besucht am 23. 04. 2018) (zitiert auf Seite 30).

[LAS15] A. Lübbe, T. Allweyer und S. Schnägelberger. *BPM Toolmarktmonitor 2015: Marktübersicht zu BPM-Systeme für Prozessautomatisierung*. Techn. Ber. 2015, S. 31 (zitiert auf Seite 18).

[LIL15] B. Len, W. Ingo und Z. Liming. *DevOps: A Software Architect's Perspective*. 1. Aufl. Addison-Wesley Professional, 2015. ISBN: 0134049845, 9780134049847 (zitiert auf Seite 21).

[LIL17] E. Laukkanen, J. Itkonen und C. Lassenius. „Problems, causes and solutions when adopting continuous delivery—A systematic literature review". In: *Information and Software Technology* 82 (2017), S. 55–79. ISSN: 09505849. DOI: 10.1016/j.infsof.2016.10.001 (zitiert auf den Seiten 33, 41).

[LL13] J. Ludewig und H. Lichter. *Software Engineering : Grundlagen, Menschen, Prozesse, Techniken*. 3. Aufl. dpunkt.verlag, 2013, S. 665. ISBN: 9783864900921 (zitiert auf Seite 20).

[McI+11] S. McIntosh u. a. „An empirical study of build maintenance effort". In: *Proceeding of the 33rd international conference on Software engineering - ICSE '11* (2011), S. 141. ISSN: 0270-5257. DOI: 10.1145/1985793.1985813. URL: http://portal.acm.org/citation.cfm?doid=1985793.1985813 (zitiert auf Seite 33).

[McI68] D. McIlroy. „Mass-Produced Software Components". In: *Proceedings of NATO Software Engineering Conference*. 1968, S. 138–155 (zitiert auf Seite 38).

[MFS14] M. Marschall, V. Firmen und D. Softwareentwicklung. „Wie Value Stream Mapping helfen kann, in Richtung Continuous Delivery zu optimieren". In: *OBJEKTspektrum - Ausgabe DevOps/2014* (2014) (zitiert auf Seite 34).

[Mug15] K. Mugrage. *5 KEY DEPLOYMENT PIPELINE PATTERNS*. 2015. URL: https://www.gocd.org/2015/08/28/pipeline-patterns/ (besucht am 26. 01. 2018) (zitiert auf den Seiten 26, 43).

[Nat11] C. Natschläger. „Towards a BPMN 2.0 ontology". In: *Business Process Model and Notation*. Hrsg. von R. Dijkman, J. Hofstetter und J. Koehler. Springer Berlin Heidelberg, 2011, S. 1–15. ISBN: 978-3-642-25159-7. DOI: 10.1007/978-3-642-25160-3. URL: http://link.springer.com/10.1007/978-3-642-25160-3 (zitiert auf Seite 33).

[NCS17] NCSC. *Secure the build and deployment pipeline*. 2017. URL: https://www.ncsc.gov.uk/guidance/secure-build-and-deployment-pipeline (besucht am 04. 05. 2018) (zitiert auf Seite 41).

[Ora] Oracle. *Wercker Developer Documentation*. URL: http://devcenter.wercker.com/docs/ (besucht am 09. 02. 2018) (zitiert auf den Seiten 30, 38, 53).

[PP08] M. Pfeiffer und J. Pichler. „A comparison of tool support for textual domain-specific languages“. In: *Proceedings of the 8th OOPSLA Workshop on Domain-Specific Modeling (DSM'08)* (2008), S. 1–7 (zitiert auf Seite 40).

[Rad16] C. Radoi. *Pretty unit/integration test reports*. 2016. URL: https://github.com/codeship/documentation/issues/336 (besucht am 11. 02. 2018) (zitiert auf Seite 49).

[Raf16] Rafael. *Feature, Run tests at a regular interval*. 2016. URL: https://community.codeship.com/t/feature-run-tests-at-a-regular-interval/123 (besucht am 11. 02. 2018) (zitiert auf Seite 50).

[Rob16] RoboPhred. *concourse-svn-resource*. 2016. URL: https://github.com/RoboPhred/concourse-svn-resource (besucht am 12. 02. 2018) (zitiert auf Seite 52).

[Rod+12] P. Rodríguez u. a. „Survey on agile and lean usage in finnish software industry“. In: *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement - ESEM '12* (2012), S. 139. ISSN: 1938-6451. DOI: 10.1145/2372251.2372275. URL: http://dl.acm.org/citation.cfm?doid=2372251.2372275 (zitiert auf Seite 1).

[Rod+17] P. Rodríguez u. a. „Continuous deployment of software intensive products and services: A systematic mapping study“. In: *Journal of Systems and Software* 123 (2017), S. 263–291. ISSN: 01641212. DOI: 10.1016/j.jss.2015.12.015 (zitiert auf Seite 33).

[RS09] M. Rother und J. Shook. *Learning to See: Value Stream Mapping to Add Value and Eliminate Muda*. 1.4. Cambridge: Lean Enterprise Institute, 2009, S. 132. ISBN: 0966784308. URL: http://www.leanenterprises.com/Library/Learning_to_See_Foreword.pdf (zitiert auf Seite 20).

[Sam97] J. Sametinger. *Software Engineering with Reusable Components*. Springer-Verlag, 1997. ISBN: 3-540-62695-6 (zitiert auf Seite 38).

[Shi17] P. Shipley. *Easy Blue-Green Deployments on Amazon EC2 Container Service*. 2017. URL: https://blog.codeship.com/easy-blue-green-deployments-on-amazon-ec2-container-service/ (besucht am 11. 02. 2018) (zitiert auf Seite 50).

[Sti16] S. Stipl. *API documentation*. 2016. URL: https://github.com/concourse/concourse/issues/771 (besucht am 12. 02. 2018) (zitiert auf Seite 52).

[Val+13] U. Valentini u. a. *Requirements Engineering und Projektmanagement*. Hrsg. von A. Herrmann, E. Knauss und R. Weißbach. 2013. ISBN: 9783642294327. DOI: 10.1007/978-3-642-29432-7_3 (zitiert auf Seite 1).

[Wea16] R. Weaver. *The Shift Left Principle and DevOps*. 2016. URL: https://dzone.com/articles/the-shift-left-principle-and-devops-1 (besucht am 06. 05. 2018) (zitiert auf Seite 37).

[Zor10] S. Zor. „Using BPMN for Modeling Manufacturing Processes“. In: *Conference on Manufacturing* (2010), S. 1–8. URL: http://www.iaas.uni-stuttgart.de/RUS-data/INPROC-2010-48-UsingBPMNforModelingManufacturingProcesses.pdf (zitiert auf den Seiten 33, 34).

[Ama] Amazon Web Services, Inc. *Amazon EC2*. URL: https://aws.amazon.com/de/ec2/ (besucht am 06. 05. 2018) (zitiert auf Seite 50).

[Bet13] M. Betts, Dominic and Dominguez, Julian and Melnik, Grigori and Simonazzi, Fernando and Subramanian. *Exploring CQRS and Event Sourcing: A Journey into High Scalability, Availability, and Maintainability with Windows Azure*. Microsoft patterns & practices, 2013. ISBN: 9781621140160 (zitiert auf Seite 72).

[Doc] Docker, Inc. *Docker - Build, Ship, and Run Any App, Anywhere*. URL: https://www.docker.com/ (besucht am 15. 04. 2018) (zitiert auf den Seiten 37, 88).

[Her] Heroku, Inc. *HEROKU*. URL: https://www.heroku.com/ (besucht am 05. 05. 2018) (zitiert auf Seite 50).

[Mic] Microsoft Corporation. *Azure Container Services*. URL: https://azure.microsoft.com/de-de/services/container-service/ (besucht am 06. 05. 2018) (zitiert auf Seite 50).

[Mon] MongoDB, Inc. *MongoDB*. URL: https://www.mongodb.com/ (besucht am 24. 04. 2018) (zitiert auf Seite 69).

[Npm] Npm, inc. *npm*. URL: www.npmjs.com (besucht am 05. 05. 2018) (zitiert auf Seite 88).

[Obj] Object Management Group (OMG). *CORBA IDL*. URL: http://www.corba.org/omg_idl.htm (besucht am 08. 04. 2018) (zitiert auf Seite 73).

[Obj11] Object Management Group (OMG). *Business Process Model and Notation (BPMN) Version 2.0*. 2011 (zitiert auf den Seiten 9, 11, 13, 17, 59, 67).

[Per15] Perforce Software, Inc. *Continuous Delivery: The New Normal for Software Development*. Techn. Ber. 2015. URL: http://www.perforce.com/continuous-delivery-report (zitiert auf Seite 1).

[Piva] Pivotal Software Inc. *Concourse Documentation*. URL: https://concourse.ci/single-page.html (besucht am 09. 02. 2018) (zitiert auf den Seiten 30, 38, 51).

[Pivb] Pivotal Software Inc. *Spring Boot*. URL: https://projects.spring.io/spring-boot/ (besucht am 15. 04. 2018) (zitiert auf Seite 87).

[Sof] Software in the Public Interest. *Jenkins User Documentation*. URL: https://jenkins.io/doc/ (besucht am 09. 02. 2018) (zitiert auf den Seiten 28, 38, 44).

[Sta17] Stark & Wayne. *Pipeline Templates*. 2017. URL: https://github.com/starkandwayne/pipeline-templates (besucht am 12. 02. 2018) (zitiert auf Seite 51).

[The] The Apache Software Foundation. *Apache Maven Project*. URL: https://maven.apache.org/ (besucht am 15. 04. 2018) (zitiert auf Seite 88).

[Tho] ThoughtWorks Inc. *GoCD User Documentation v18.1.0*. URL: https://docs.gocd.org/18.1.0/ (besucht am 09. 02. 2018) (zitiert auf den Seiten 38, 46).

[cama] camunda Services GmbH. *User Guide*. URL: https://docs.camunda.org/stable/guides/user-guide/ (besucht am 08. 01. 2018) (zitiert auf Seite 88).

[camb] camunda Services GmbH. *User Guide - Extension Elements*. URL: https://docs.camunda.org/manual/7.8/reference/bpmn20/custom-extensions/extension-elements/ (zitiert auf den Seiten 19, 20, 93).

Akronyme

API Application Programming Interface

B2B Business-to-Business

BAM Business Activity Monitoring

BPEL WS-Business Process Execution Language

BPM Business Process Management

BPMN Business Process Modell and Notation

BPMS Business Process Management System

CD Continuous Delivery

CDN Content Delivery Network

CI Continuous Integration

CLI Command Line Interface

CMMN Case Management Model and Notation

CQRS Command Query Responsibility Segregation

DMN Decision Model and Notation

DSL Domain Specific Language

EABPM European Association of Business Process Management

EL Unified Expression Language

EPK Ereignisgesteuerten Prozesskette

HG Mercurial

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

IaC Infrastructure as Code

JSON JavaScript Object Notation

KPI Key Performance Indicator

MVC Model-View-Controller

MVN Apache Maven

OMG Object Management Group

PaC Pipeline-as-Code

PDCA Plan-Do-Check-Act

REST Representational State Transfer

SaaS Software as a Service

SDS Software Delivery System

SVN Subversion

UML Unified Modeling Language

VSM Value Stream Mapping

XML Extensible Markup Language

XPDL XML Process Definition Language

XSD XML Schema Definition

YAML YAML Ain't Markup Language