

Prozessgetriebene Werkzeugintegration

Bachelorarbeit im Fach Informatik

vorgelegt von

Frank Andreas Denninger

geb. 31. Dezember 1981 in Coburg

angefertigt am

**Department Informatik
Professur für Open Source Software
Friedrich-Alexander-Universität Erlangen–Nürnberg
(Prof. Dr. Dirk Riehle, M.B.A.)**

Betreuer: *Prof. Dr. Dirk Riehle, M.B.A.*

*Beginn der Arbeit: Mai 2013
Abgabe der Arbeit: August 2013*

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

Erlangen, der 1. August 2013

Inhaltsverzeichnis

1. Einleitung	9
2. Anforderungen	10
2.1. Vision	10
2.2. Szenario	10
2.3. Stakeholder	12
2.4. Anforderungen	13
2.4.1. Nicht-funktionale Anforderungen	13
2.4.2. Funktionale Anforderungen	13
3. Kontext und Grundlagen	15
3.1. Geschäftsprozess-Management	15
3.2. Integration	16
3.2.1. Technische Integration	17
3.2.2. Semantische Integration	18
3.2.3. Entscheidung	18
4. Architektur	19
4.1. Laufzeit	20
4.2. Übersetzer	21
5. Design	22
5.1. Konfigurationsverwaltung	22
5.2. Semantische Integration	23
5.2.1. Event-Prozessor-Vokabular	23
5.2.2. Rollenvokabular	24
5.2.3. Zugriff	26
5.2.4. Issue-Tracking-Vokabular	27
5.3. Nachrichtenbus	29
5.3.1. Queued Transactions	29
5.3.2. Nachrichtenverarbeitung	31
5.3.3. Zusammenfassung	31
5.4. Event-Prozessor	32
5.4.1. EventControl	33
5.4.2. Triggers	33
5.4.3. Engine	34

5.5.	Übersetzer	36
5.5.1.	Machina-Prozessmodell	37
5.5.2.	Transformationen	39
5.5.3.	Codegenerator	39
5.5.4.	Engine	40
5.6.	Binding	40
5.6.1.	Enactment-Binding	40
5.6.2.	Workflowparameter-Binding	41
5.6.3.	Symboltabelle	43
6.	Technologien	45
6.1.	Activiti	45
6.1.1.	Business Process Modelling and Notation	45
6.1.2.	BPMN Erweiterungen	47
6.2.	HornetQ - JMS	48
6.3.	ANTLR	49
6.4.	JIRA	49
6.5.	Stages	49
6.6.	Eclipselink - JPA	49
6.7.	Spring	50
7.	Implementierung	51
7.1.	Konfiguration	51
7.2.	Trigger	51
7.3.	Jira-Modul	51
7.3.1.	Plugin	52
7.3.2.	Eventnachrichtenverarbeitung	53
7.3.3.	Aktionsnachrichtenverarbeitung	54
7.4.	Nachrichtenbus	55
7.5.	EventControl	56
7.5.1.	EventLogRepository	56
7.5.2.	Zustandsautomat	57
7.6.	Engine	58
7.6.1.	CommandExecutor	58
7.6.2.	MessageCache - MessageRepository	59
7.6.3.	JavaDelegates	60
7.7.	Übersetzer	61
7.7.1.	StagesModelReader	61
7.7.2.	Transformatoren	64
7.7.3.	Expressions-Parser	64
7.7.4.	Binder	65
7.8.	Activiti BPMN Synthese	65
7.8.1.	Activiti-Synthesemodell	65
7.8.2.	Model-To-Model Transformation	66

7.8.3. Model-To-Text Transformation	68
8. Bewertung	69
8.1. Vergleich mit Anforderungen	69
8.2. Quelltextmetrik	70
8.3. Offene Punkte	71
8.3.1. Detailverbesserungen	71
8.3.2. Offene Fragen	71
8.3.3. Zukünftige Arbeit	72
9. Fazit	74
A. Anhang	76
A.1. Beispiel Semantik	76
A.2. Analyse Change Request Fragment	77
A.3. Erweiterbare Standardsemantik für Stages	79
Literaturverzeichnis	80

Abbildungsverzeichnis

2.1. Systemvision	10
2.2. BPMN-Diagramm der Änderungsanforderung	11
3.1. Integriertes Geschäftsprozess- und Workflow-Management [Gad10, S. 2]	16
4.1. Architekturübersicht	19
4.2. Kommunikation während der Laufzeit	20
5.1. Konfigurationsverwaltung	22
5.2. Event und EventArgument	24
5.3. Event-Prozessor-Vokabular beschrieben durch Rollenschnittstellen	25
5.4. AdapterFactoryRegistry	26
5.5. Erzeugen einer BoundRole mittels AdapterFactoryRegistry	27
5.6. Issue-Tracking-Vokabular	28
5.7. „Running a Request as Three Transactions“ [BN09, S. 103]	30
5.8. Aufbau des Event-Prozessors	32
5.9. TriggerRepository und CommandExecutor Schnittstelle	34
5.10. Aufbau der Trigger	35
5.11. Engine Schnittstelle	35
5.12. „Pipe and Filter“ Struktur der Übersetzer-Pipeline	36
5.13. Machina-Prozessmodell	37
5.14. Machina-Expressions-Modell	38
5.15. ModelHandler Schnittstelle	39
5.16. MEnactment mit MEnactmentBinding	40
5.17. Modell des Prozessbindings	41
5.18. Klassen zum Repräsentieren des Workflowtyps und Enactment-Typen	42
5.19. SymbolTable Schnittstelle	43
6.1. StartEvent und EndEvent	46
6.2. Die verschiedenen Gateways	46
6.3. Task und ServiceTask	47
6.4. „Underlying store-and-forward mechanism guarantee delivery of messages“ [RMHC09, S. 24]	48
7.1. Komponentendiagramm des Jira-Moduls	52
7.2. Kommunikationsdiagramm für die Verarbeitung der Eventnachrichten	53
7.3. Kommunikationsdiagramm für die Verarbeitung der Aktionsnachrichten	54

7.4. EventLogRepository Schnittstelle	57
7.5. Zustandsautomat in der Klasse <i>EventMachine</i>	57
7.6. Zwischenspeicher für Aktionsnachrichten	60
7.7. Erzeugen der Aktionsnachricht in einem BlockingTaskDelegate	61
7.8. Gekürztes Stages-Prozessmodell	62
7.9. Graphische Repräsentation des Szenarios	62
7.10. Semantik Schnittstelle	62
7.11. Teil der Standardsemantik zum Ändern der Prozesselementerzeugung . .	63
7.12. Teil der Standardsemantik zum Ändern der Assoziationserzeugung . . .	64
7.13. Activiti-Synthesemodell	65
7.14. Generiertes BPMN Fragment einer blockierenden Aktion	67
A.1. Erweiterbare Standardsemantik für Stages-Prozesse	79

Tabellenverzeichnis

3.1. „A classification of integration standards“ [LBK07, S. 203]	17
3.2. Aufteilung der Integrationsaspekte in Machina	18
5.1. Übersicht über Aktionstypen	24
5.2. Schema für einen Verarbeitungsschritt während der Queued Transaction .	30
5.3. In Machina verwendetes Schema für die Nachrichtenverarbeitung	31
5.4. Schema für Verarbeitung im EventControl	33
7.1. Schema zum versenden der <i>IssueEvents</i> in JIRA	53

1. Einleitung

Optimierung von Geschäftsprozessen ist eine der großen Herausforderungen in Unternehmen. Man geht diese Herausforderung an, da man sich hiervon einen entscheidenden Wettbewerbsvorteil verspricht. Für Informatik ergibt sich damit das Problem wie man diese Optimierung mit Software unterstützt oder gar erst möglich macht.

Eine notwendige Bedingung für eine Umsetzung ist die Formulierung des Geschäftsprozesses in einer Notation die von Computern verarbeitet werden kann. Hier wurde mit der „Business Process Modeling and Notation 2.0“ 2011 ein neuer Standard verabschiedet. Das Besondere an dieser Sprache war, dass man keine unterschiedlichen Formate mehr für die Formulierung der Prozesse auf konzeptioneller Ebene sowie der Workflows auf operativer Ebene benötigt. Hierdurch kann man einen Prozess direkt mit den notwendigen Spezifikationen für einen Workflow anreichern ohne ihn komplett in einer anderen Sprache umzuschreiben.

In dieser Arbeit wurde der Fokus auf Prozesse gelegt, die die Softwareentwicklung unterstützen. In größeren Softwareentwicklungsprojekten sind bereits eine Vielzahl von „Product Lifecycle Management“ (PLM) Werkzeugen im Einsatz. Angefangen mit den üblichen Issue-Tracking- oder Source-Code-Management-Systemen, die schon in kleinen Projekten Einsatz finden, bis zu den großen Kollaborationsplattformen wie „Rational Team Concert“. Der übliche Ansatz eine werkzeugübergreifende Workflowengine in diese Werkzeuglandschaft zu bringen ist diese um ein weiteres Werkzeug zu erweitern. Dieses Werkzeug führt den Benutzer Anhand von Aufgabenlisten durch den Workflow. Der Benutzer wird damit angewiesen bestimmte Tätigkeiten in den anderen Werkzeugen durchzuführen. Dieses Vorgehen hat einige Nachteile, zum Einen erzeugt es Reibungsverluste durch das Wechseln zwischen verschiedenen Werkzeugen, zum Anderen sind die über mehrere Werkzeuge hinweg verteilten Daten nicht davor geschützt inkonsistent zu werden.

An diesem Punkt setzt diese Arbeit an, es wurde ein Prototyp entwickelt, genannt Machina, der versucht die eigentliche Workflowengine für die Benutzer unsichtbar zu machen. Es tritt kein weiteres Werkzeug in Erscheinung. Die Aufgaben der Aufgabenliste werden auf die tatsächlichen Tätigkeiten in den Werkzeugen abgebildet. Man verspricht sich davon, dass sich der Reibungsverlust durch den Wechsel zwischen der Aufgabenliste der Workflowengine und den Werkzeugen reduziert. Zusätzlich soll die Integration Inkonsistenzen der Daten verhindern. Damit ergeben sich zwei Probleme, die es zu lösen gilt. Zum Einen musste die Prozessbeschreibung so aufbereitet werden das sie von Machina verwendet werden kann. Zum Anderen ist die Integration von unterschiedlichen Werkzeugen ein Problem, das beliebig komplexe Lösungen haben kann. Machina versucht weder alle Integrationsprobleme zu lösen noch sämtliche Prozesse ausführbar zu machen. Es sollte eine sinnvolle Balance zwischen Funktionalität und Komplexität gefunden werden.

2. Anforderungen

2.1. Vision

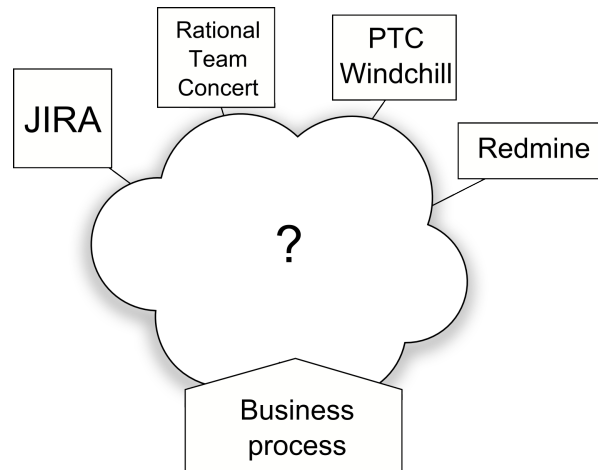


Abbildung 2.1.: Systemvision

Die Vision für das System war zunächst sehr abstrakt. Man nimmt ein Prozessmodell als Ausgangsbasis, also prozessgetrieben, und integriert damit eine Vielzahl von PLM ¹ Werkzeugen. Wie dies erreicht werden sollte war zunächst offen.

2.2. Szenario

Aufbauend auf der Vision wurde ein einfaches Szenario erstellt. Es wurde ein Beispielprozess im Änderungsmanagement weiter vereinfacht. Das Ergebnis ist der Änderungsanforderungsprozess in Abbildung 2.2. Dieses Szenario stellt keinen real existierenden Anwendungsfall dar, sondern nur das Szenario das versucht wird mit Machina umzusetzen.

Das besondere an diesem Szenario ist, dass für den Endanwender die eigentliche Workflowengine komplett in den Hintergrund tritt. Der Benutzer also durch diese Schritte geleitet wird, ohne sein gewohntes Werkzeug, in diesem Fall JIRA, zu verlassen.

¹Product Lifecycle Management

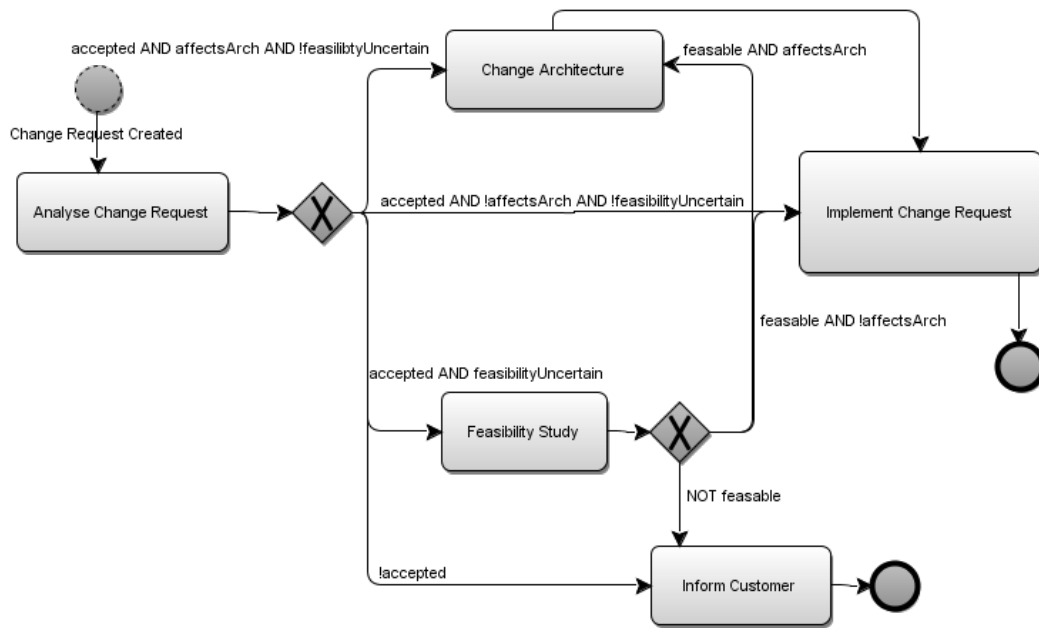


Abbildung 2.2.: BPMN-Diagramm der Änderungsanforderung

Starten ² Durch die Erzeugung eines Issues mit dem Typ „Change Request“ wird eine neue Workflowinstanz gestartet.

Analyse Change Request Es erscheint ein neuer Issue für den Benutzer, der ihn anweist die ursprüngliche Änderungsanforderung zu analysieren. Er muss dabei drei Entscheidungen treffen:

1. Wird die Änderungsanforderung akzeptiert oder abgelehnt?
2. Ist sich der Prüfer sicher, dass die Änderungsanforderung umsetzbar ist?
3. Ist sich der Prüfer sicher, dass die Änderungsanforderung keine Auswirkungen auf die Systemarchitektur hat?

Feasibility Study Der Prüfer hat die Änderungsanforderung akzeptiert, ist sich aber nicht sicher ob diese lösbar ist. Es wird ein neuer Issue erzeugt, der den Benutzer anweist eine Machbarkeitsstudie durchzuführen. Am Ende der Studie muss er darüber entscheiden, ob die Änderungsanforderung umsetzbar ist.

Change Architecture Bei der Analyse wurde festgestellt, dass diese Änderungsanforderung möglicherweise Auswirkungen auf die Systemarchitektur hat. Es wird ein neuer Issue erzeugt, der den Benutzer anweist diese Auswirkungen weiter zu analy-

²Graphisch als Kreis dargestellt

sieren und gegebenenfalls die Systemarchitektur zu ändern. Mit dem abschließen dieses Issues bestätigt der Benutzer dies getan zu haben.

Implement Change Request Alle Vorbereitungen wurden getroffen; die Änderungsanforderung kann umgesetzt werden. Es wird ein Issue erzeugt, der den Benutzer anweist diese Änderungsanforderung zu implementieren. Sobald dieser Issue abgeschlossen wird, kann diese Workflowinstanz abgeschlossen werden.

Inform Customer Die Änderungsanforderung wurde abgelehnt oder die Prüfung hat festgestellt, dass diese nicht umsetzbar ist. In diesem Task wird der Kunde darüber informiert. Anschließend wird diese Workflowinstanz beendet.

Dieses Szenario spiegelt noch nicht alle Anforderungen an Machina wieder. Es soll möglich sein mehrere unabhängige Werkzeuge³ zu benutzen.

2.3. Stakeholder

Hier sind die wichtigsten Stakeholder an Machina aufgeführt. Dabei wurde sich darauf auf diejenigen beschränkt, die direkt mit Machina arbeiten oder es verändern.

Werkzeugbenutzer Diese Stakeholder sind die Gruppe, die die integrierten Werkzeuge benutzen. Es ist zu erwarten, dass diese Gruppe ihre derzeitigen Arbeitsabläufe nur ungern ändert und sich auch nicht in neue Werkzeuge einarbeiten möchten. Da diese Gruppe Machina auch tatsächlich benutzen wird, ist die benötigte Akzeptanz hier besonders hoch. Es wurde versucht, dass sich für diese Gruppe möglichst wenig ändert. Anforderungen: REQ 2, REQ 3, REQ 7, REQ 8, REQ 16.

Prozessmodellierer, Business Analyst Für den Prozessmodellierer ist es besonders wichtig, dass er seine Prozesse weiterhin unabhängig von der operativen Ebene entwickeln kann. Er sollte dabei mit möglichst wenig technischen Details konfrontiert werden. Unterstützung durch Diagnoseausgaben für fehlerhafte Modelle ist für diese Gruppe unabdingbar. Anforderungen: REQ 9, REQ 11, REQ 12, REQ 14, REQ 15.

Administrator Der Administrator ist dafür verantwortlich die Infrastruktur am Laufen zu halten, das bedeutet vor allem fehlerhafte Zustände zu erkennen und diese korrigieren zu können. Anforderungen: REQ 4, REQ 17.

Setup Rollout Diese internen Stakeholder passen das System nach Kundenwunsch an. Für sie ist es besonders wichtig auf viele Kundenwünsche eingehen zu können. Der Aufwand, sowie die damit verbundenen Kosten für denn den Kunden, müssen dabei aber möglichst gering gehalten werden. Dies bedeutet, dass sich diese Anpassungen auf Konfigurationsmöglichkeiten beschränkten sollten und keine Änderungen am Systemkern notwendig machen. Es ist möglich diese Stakeholder besonders zu schulen, eine intuitive

³Allgemein PLM Werkzeuge, nicht beschränkt auf Issue-Tracking-Systeme

Bedienung wäre zwar angenehm, ist aber nur bedingt nötig. Die Schulungs- und Einarbeitungszeit sollte sich in einem angemessenen Rahmen halten. Anforderungen: REQ 10.

Werkzeugmodul Entwickler Diese internen Stakeholder entwickeln/erweitern die Anbindungen an Werkzeuge. Es sollte ihnen möglich sein mit geringem Aufwand Werkzeuge in das Repertoire aufzunehmen. Das lässt sich unter anderem dadurch erreichen, dass das zu erstellende Werkzeugmodul möglichst einfach aufgebaut ist und keine spezielle Logik implementiert werden muss. Anforderungen: REQ 5, REQ 6.

2.4. Anforderungen

Die Anforderungen an Machina sind zunächst als lediglich gewünscht zu verstehen. Ziel ist es zu formulieren, welche Eigenschaften an einem System angenehm wären, um dann zu bewerten welchen Aufwand bestimmte Anforderungen verursachen.

2.4.1. Nicht-funktionale Anforderungen

Diese Anforderungen beschreiben die Eigenschaften eines Systems.

REQ 1 Einsatz von Standards: Wo es möglich ist, sollen angemessene Standards eingesetzt werden.

REQ 2 Verfügbarkeit: Machina soll die Verfügbarkeit der Werkzeuge nicht negative beeinflussen, auch soll Machina nicht komplett handlungsunfähig sein nur weil ein einzelnes Werkzeug nicht verfügbar ist.

REQ 3 Akzeptable Reaktionszeiten: Der Einsatz von Machina sollte sich nicht in signifikant langsameren Reaktionszeiten der Werkzeuge bemerkbar machen. Als Obergrenze für die zusätzliche Reaktionszeit wurde eine Sekunde festgelegt.

2.4.2. Funktionale Anforderungen

Diese Anforderungen beschreiben die Funktionalität eines Systems.

Anforderungen an die Einbindung der Werkzeuge

REQ 4 Sicheres Ablegen der Konfigurationen: Die Konfigurationen der Werkzeuge müssen vor fremden Zugriff abgesichert sein.

REQ 5 Erweiterbarkeit um neue Werkzeuge: Es muss möglich sein das System um weitere Werkzeuge zu erweitern. Diese Erweiterungen sollten keine Veränderung am bestehenden System notwendig machen.

REQ 6 Wenig Veränderungen an den Werkzeugen: Die angebundenen Werkzeuge sollten nach Möglichkeit nicht verändert werden. Falls dies dennoch notwendig sein sollte, ist die Pluginschnittstelle des Werkzeugs hierfür zu verwenden.

REQ 7 Unterstützung gewohnter Arbeitsabläufe: Das System soll sich den bisherigen Arbeitsabläufen anpassen können.

REQ 8 Zugriff auf Workflowkontext: Den Benutzern der angebundenen Werkzeuge soll ein Weg zur Verfügung gestellt werden auf den Workflow-Kontext zuzugreifen, um damit ihre aktuelle Aufgabe in einen Zusammenhang setzen zu können.

Anforderungen an die Modellierung

REQ 9 Modellierung in Stages: Die Modellierung muss mit dem Produkt „Stages“ der Firma Methodpark möglich sein.

REQ 10 Projektspezifische Semantiken: Semantiken der Prozessmodelle sollen projektspezifisch festgelegt werden können, damit man sich auf die besonderen Eigenheiten einzelner Projekte einstellen kann.

REQ 11 Keine Laufzeitinformationen im Prozessmodell: Im Prozessmodell sollte sich möglichst wenig Artefakte für Machina befinden. Explizite Angaben von Laufzeitinformationen sind zu vermeiden.

REQ 12 Unterstützung bei Modellierungsfehlern: Dem Modellierer sollten aussagekräftige Diagnosemeldungen präsentiert werden, mit denen die Lokalisierung der Fehler erleichtert wird.

Anforderungen an die Laufzeit

REQ 13 Beliebige Werkzeugkombinationen sollen möglich sein.

REQ 14 Mehrere Instanzen des gleichen Werkzeugs sollen in einem Workflow möglich sein.

REQ 15 Metriken ermöglichen: Speicherung detaillierter Daten über die Arbeitsabläufe.

REQ 16 Zuverlässige Ausführung: Es muss sichergestellt werden, dass keine Benutzereingaben verloren gehen oder nicht alle gewünschten Effekte ausgelöst werden.

REQ 17 Dem Administrator müssen erkannte Fehler zuverlässig zugestellt werden. Er sollte die Möglichkeit haben korrigierend einzuwirken.

3. Kontext und Grundlagen

Zunächst werden hier einige theoretische Grundlagen vorgestellt. Diese Grundlagen begründen einige der Entscheidungen, die in der Architektur und im Design getroffen wurden. Sinn dieses Kapitels ist es nicht diese ausführlich zu erläutern, sondern einen Überblick zu verschaffen und auf weiterführende Literatur zu verweisen.

Wie das Thema schon nahelegt gibt es zwei große Blöcke die es zu beachten gilt: Geschäftsprozess-Management und Integration.

3.1. Geschäftsprozess-Management

Das zentrale Element hierbei ist der Geschäftsprozess, häufig abgekürzt zu Prozess. Er wird definiert als „eine Reihe von festgelegten Tätigkeiten (Aktivitäten, Aufgaben), die von Menschen oder Maschinen ausgeführt werden, um ein oder mehrere Ziele zu erreichen“ [Sti13, S. 3]. Als Teilbereich der Betriebswirtschaftslehre beschäftigt sich das Geschäftsprozess-Management unter anderem mit diesem.

Prozess-Management ist ein zentraler Bestandteil eines integrierten Konzeptes für das Geschäftsprozess- und Workflow-Management. Es dient dem Abgleich mit der Unternehmensstrategie, der organisatorischen Gestaltung von Prozessen sowie deren technischen Umsetzung mit geeigneten Kommunikations- und Informationssystemen. [Gad10, S. 1]

Das Konzept des *Geschäftsprozess-Managements* besteht aus drei Ebenen (Abbildung 3.1). „Auf der *strategischen Ebene* werden die Geschäftsfelder eines Unternehmens einschließlich der hier wirksamen kritischen Erfolgsfaktoren betrachtet“ [Gad10, S. 1]. Diese Ebene beschäftigt sich mit der Weiterentwicklung der Unternehmensstrategie. Auf der *fachlich-konzeptionellen Ebene* wird die Unternehmensstrategie vom *Prozess-Management* auf einzelne Prozesse heruntergebrochen. Diese Prozesse werden wiederum an die *operative Ebene* weitergeben. Auf der operativen Ebene wird im *Workflow-Management* während der *Workflowmodellierung* der Geschäftsprozess in einen Workflow umgewandelt.

Für diese Arbeit ist hauptsächlich das Workflow-Management auf der operativen Ebene interessant. Es besteht aus den drei Teilen *Workflowmodellierung*, *Workflowausführung* und *Workflowmonitoring*.

Workflowmodellierung „Hierbei wird der modellierte Geschäftsprozess um Spezifikationen erweitert, die für eine automatisierte Prozessausführung unter der Kontrolle eines Workflow-Management-Systems notwendig sind“ [Gad10, S. 2]. Das Ergebnis der Workflowmodellierung ist ein Workflowmodell.

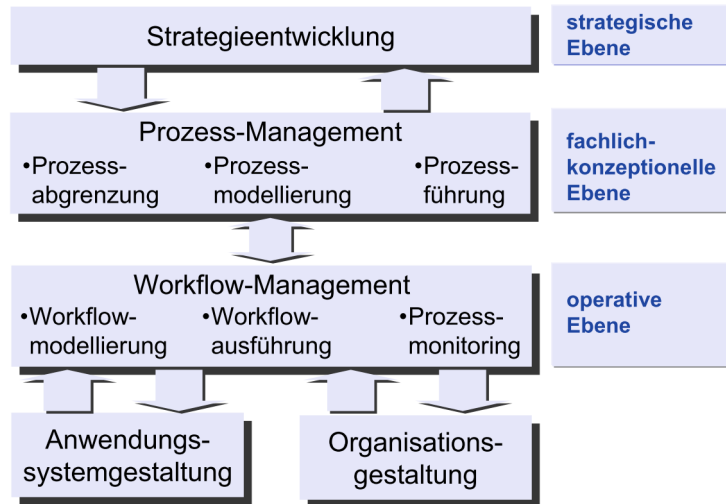


Abbildung 3.1.: Integriertes Geschäftsprozess- und Workflow-Management [Gad10, S. 2]

Workflowausführung Hierbei wird der zum Workflow erweiterte Prozess unter der Kontrolle eines *Workflow-Management-Systems* ausgeführt.

Workflowmonitoring „Dient der laufenden Überwachung des Prozessverhaltens“ [Gad10, S. 2], indem es während der Ausführung kontinuierlich die Zustände der Workflowinstanzen sammelt und diese für Korrekturen oder Optimierungen bereit hält.

Eine genauere Analyse der Unterschiede zwischen den Begriffen Prozess und Workflow ist in [Gad10, S. 40 f.] zu finden. In dieser Arbeit wird das Wort „Prozess“ nur für die fachlich-konzeptionelle Ebene gewählt. Ein Workflow ist ein Geschäftsprozess, der um die notwendigen Spezifikation für eine Ausführung angereichert wurde. Da ein Workflow auch mehrfach ausgeführt werden kann, wird das Wort „Workflowinstanz“ für eine konkrete Ausführung eines Workflows benutzt.

3.2. Integration

Als zweiter großer Grundlagenbereich ergibt sich die Integration. „Enterprise integration is the task of making disparate applications work together to produce a unified set of functionality.“ [HW04, S. 39] Man versucht also Anwendungen, die zunächst unabhängig funktionieren, zu einer Zusammenarbeit zu bewegen, um eine gemeinsame Funktionalität bereitzustellen. Diese Definition ist noch so unspezifisch, dass sie kaum dabei hilft dieses Ziel zu erreichen. Für diese Arbeit sehr hilfreich ist die in [LBK07] vorgestellte Klassifikation für Integrationsaspekte. Diese Klassifikation lässt sich durch Matrix 3.1 zusammenfassen.

Integration besteht demnach aus zwei orthogonalen Dimensionen: Zum Einen die Reichweite der Integration: Belässt man es bei einer *technischen Integration* oder geht man

	Technical integration	Semantic integration
Data integration	Syntactic frameworks	Ontology and vocabulary
Functional integration	Middleware	Application frameworks

Tabelle 3.1.: „A classification of integration standards“ [LBK07, S. 203]

weiter zur *semantischen Integration*. Als zweite Dimension gibt es die Mittel. Bei *Datenintegration* erzeugt man eindeutige semantische Referenzen für gemeinsame benutzte Daten, während die *funktionale Integration* eine sinnvolle Zusammenarbeit der verschiedenen Komponenten ermöglicht.

3.2.1. Technische Integration

Die technische Integration ist dafür zuständig, die Zusammenarbeit der einzelnen Systeme auf technischer Ebene zu ermöglichen. In [HW04, S. 41] werden vier Stile der funktionalen Seite der Technischen Integration identifiziert.

File Transfer „Have each application produce files of shared data for others to consume and consume files that others have produced.“ [HW04, S. 41]

Shared Database „Have the applications store the data they wish to share in a common database“ [HW04, S. 41]

Remote Procedure Invocation „Have each application expose some of its procedures so that they can be invoked remotely, and have applications invoke those to initiate behavior and exchange data“ [HW04, S. 41]

Messaging „Have each application connect to a common messaging system, and exchange data and invoke behavior using messages.“ [HW04, S. 41]

Jeder dieser Stile hat bestimmte Vorteile aber auch Nachteile; keiner kann allen Anwendungsfällen gerecht werden. Neben den funktionalen Aspekten der technischen Integration ist es notwendig, sich auf ein gemeinsames Datenaustauschformat zu einigen [HW04, S.40]. Diese lassen sich in zwei Hauptkategorien einteilen.

Binär Die Daten werden binär codiert. Hier sind eine Vielzahl von Problemen zu lösen, z.B. Endianess. Ein großer Vorteil dieser Formate ist der geringere Platzbedarf und Verarbeitungsaufwand.

Textbasiert Alle Daten werden als Text repräsentiert. Dies benötigt zusätzlichen Speicherplatz, vermeidet jedoch viele Probleme der binären Formate. Bekannte Vertreter dieser Kategorie sind XML und JSON.

3.2.2. Semantische Integration

Bei der semantischen Integration stellt man sicher, dass sich die einzelnen Anwendungen tatsächlich verstehen. Die Vorgehensweisen um dies zu erreichen sind spezifisch für die jeweilige Domäne. In [HW04] werden einige Muster vorgestellt, die als Inspiration für das Abschnitt 5.2 vorgestellte Design genutzt wurden. Generell lässt sich sagen, dass die Datenintegration durch die Definition von geeigneten Schnittstellen und Datenklassen durchgeführt wurde, während die funktionale Integration durch Implementierung dieser Schnittstellen ausgedrückt wird.

3.2.3. Entscheidung

Es wurde sich bei Machina für die Technologien in Matrix 3.2 entschieden. Die semantische Integration wird durch das Rollenkonzept realisiert, das im Abschnitt 5.2 genauer beschrieben wird.

Technische funktionale Integration Da eine wichtige Anforderung (REQ 2) die Verfügbarkeit ist, ist die Wahl auf „Messaging“ gefallen. Durch asynchrone Nachrichtenübertragung werden die einzelnen Systeme weitgehend entkoppelt, damit ist sichergestellt, dass Machina keinen negativen Einfluss auf die Verfügbarkeit der angeschlossenen Werkzeuge hat. Sollte ein Werkzeug dieses nicht direkt benutzen können, wird mittels Plugin oder einem Proxy (z.B. RESTful Webservice auf Nachrichtenbus) nachgeholfen.

Technische Datenintegration Bei den zu erwartenden kleinen Datenmengen von wenigen Kilobyte, fällt der zusätzliche Platzbedarf durch den Einsatz von textbasierten Formaten kaum ins Gewicht. Es wurde also aufgrund ihrer Einfachheit für technische Datenintegration XML bzw. JSON eingesetzt. Sollte sich herausstellen, dass größere Datenmengen nötig werden, muss diese Entscheidung jedoch überdacht werden.

	Technische Integration	Semantische Integration
Datenintegration	Textbasiert XML und JSON	Rollen und Adapter Schnittstellen
Funktionale Integration	Messaging Java Message Service	Rollen und Adapter Implementierungen

Tabelle 3.2.: Aufteilung der Integrationsaspekte in Machina

4. Architektur

Die Abbildung 4.1 beschreibt die Hauptkomponenten von Machina. Diese Komponenten kann man weiter zerlegen in die drei Laufzeit Komponenten: *Execution*, *Integration* und *Tools* sowie in die *Compiler*-Komponente.

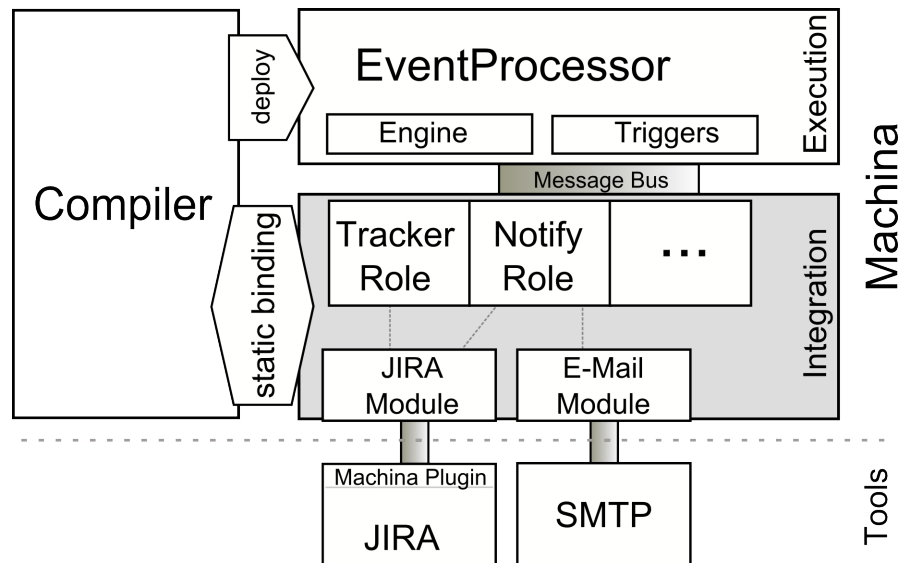


Abbildung 4.1.: Architekturübersicht

Integration Diese Komponente verarbeitet und erzeugt werkzeugspezifische Nachrichten. Dabei erzeugt sie entweder generische Events für die *Execution*-Komponente oder reagiert auf deren Aktionen indem sie Effekte in den angebunden Werkzeugen auslöst.

Execution Diese Komponente verarbeitet die generische Events und erzeugt dabei neue Aktionen. Zusammen mit der *Integration*-Komponente bildet sie das Laufzeitsystem.

Compiler Der Übersetzer bereitet ein Prozessmodell für Machina auf. Er ist orthogonal zu den beiden anderen Komponenten angeordnet, da er nur aktiv ist, sobald ein neuer Prozess eingebracht werden soll. Während des Deploymentvorgangs kommuniziert er zum einen mit der *Integration*-Komponente um Informationen über

die angebundenen Werkzeuge zu erhalten, und zum anderen mit der *Execution*-Komponente, um das neue Programm einzuspielen.

Tools Die angebundenen externen Systeme oder Werkzeuge. Diese sollen mittels Machina integriert werden und zusammenarbeiten können. Falls es keine Möglichkeit gibt diese Systeme so zu konfigurieren, dass sie Events an Machina weiterleiten, ist es notwendig diese Systeme durch eine Plugin zu erweitern. Polling kann in Erwägung gezogen werden, man muss aber die damit entstehenden Latenzen¹ akzeptieren.

4.1. Laufzeit

Die Laufzeit ist eine aus drei Schichten bestehende Schichtarchitektur [BMR⁺96, S. 31 ff.]. Die drei Schichten *Tools*, *Integration* und *Execution* werden im Design detailliert beschrieben. Für die Kommunikation zwischen den einzelnen Schichten gibt es mehrere Möglichkeiten. Eine Operation, die eine permanente Änderung des Zustands zur Folge haben könnte, muss über den Nachrichtenbus durchgeführt werden. Für alle anderen Operationen genügt es diese mittels Standard Methodenaufrufe zu realisieren. Die Kommunikation mit den Werkzeugen findet durch geeignet Fernaufrufe² statt. Das Kommunikationsdiagramm in Abbildung 4.2 beschreibt die drei Phasen während der Verarbeitung zur Laufzeit.

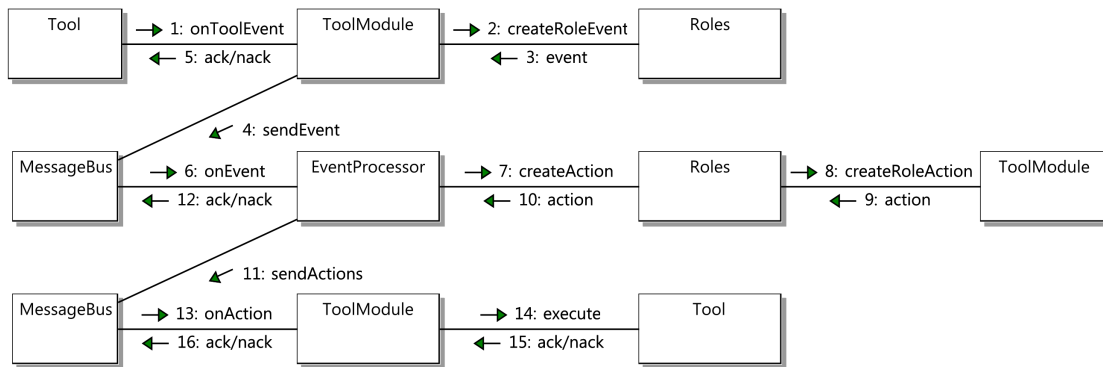


Abbildung 4.2.: Kommunikation während der Laufzeit

1-5 Vorverarbeitung Das Werkzeugplugin erkennt einen Event im Werkzeug. Dieser werkzeugspezifische Event wird an das Werkzeugmodul weitergegeben. Das Werkzeugmodul bereitet zusammen mit seinen Rollen den Event auf. Der entstehende generische Event wird mit dem Nachrichtenbus versendet.

¹Die mittlere Latenz ist gegeben durch die Polling-Frequenz

²Für die Anbindung von JIRA wird auch hierfür der Nachrichtenbus verwendet.

6-12 Ausführung Der Event-Prozessor nimmt generische Events entgegen und verarbeitet diese. Während der Verarbeitung werden von den Rollen und den Werkzeugmodellen eine oder mehrere Aktionsnachrichten erzeugt. Diese werden gebündelt, sobald der komplette Event verarbeitet wurde, an den Nachrichtenbus übergeben.

13-16 Auslösung Einzelne Werkzeugmodule nehmen diese Aktionsnachrichten entgegen und sorgen dafür, dass diese Aktion einen bestimmten Effekt im Werkzeug genau einmal auslöst.

4.2. Übersetzer

Der Übersetzer erzeugt aus einem Prozessmodell einen ausführbaren Workflow. Er folgt dem Aufbau einer „Pipe and Filters“ [BMR⁺96, S. 53 ff.] Architektur. Er besteht aus mehreren unabhängigen Filtern, die durch Pipes miteinander verbunden sind. Jeder dieser Filter übernimmt einen kleinen Teil der Transformation von Prozess nach Workflow. Hierdurch bleiben die einzelnen Filter überschaubar und folgen dem Prinzip des „Separation of Concerns“. Mit der *Integration*-Komponente kommuniziert der Übersetzer, um an die notwendigen Laufzeitinformationen zum Durchführen des Bindevorgangs zu erhalten. Während des Deploymentvorgangs wird der erzeugte Workflow an die Engine in der *Execution*-Komponente übergeben.

5. Design

Zur Beschreibung des Designs wurde ein Bottom-up Ansatz gewählt. Es werden zunächst die einzelnen Komponenten beschrieben, die anschließend zum vollständigen System zusammengesetzt werden.

5.1. Konfigurationsverwaltung

Diese Komponente verwaltet die Konfigurationen der angebundenen Werkzeuge, sie stellt einen Teil der Integrationskomponente dar. Sie soll vom tatsächlicher Speicherort der Konfigurationsdaten abstrahieren und einen einheitlichen Weg zur Verfügung stellen, mit dem die einzelnen Werkzeugmodule ihre Daten einlesen können. In Abbildung 5.1 sind die beteiligten Klassen und Schnittstellen aufgeführt.

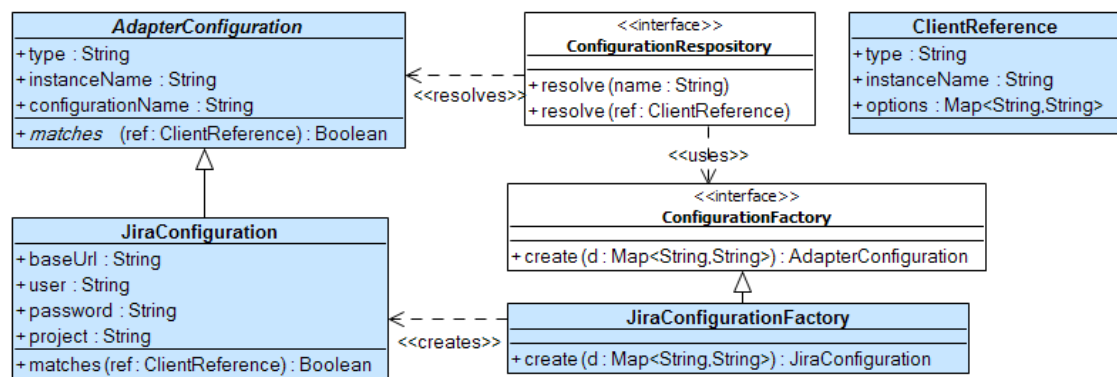


Abbildung 5.1.: Konfigurationsverwaltung

AdapterConfiguration Eine Konfiguration besteht aus einem allgemeinen Teil, der *AdapterConfiguration* mit Attributen, die für alle Konfigurationen relevant sind. Sie wird eindeutig durch das Attribut *configurationName* identifiziert. Durch Spezialisierung ist es den Werkzeugen möglich eigene Attribute zu hinterlegen. Instanzen werden gemäß dem Pattern AbstractFactory [GHJV94, S. 107 ff.] von Implementierungen der *ConfigurationFactory* Schnittstelle erzeugt.

ConfigurationRepository Das *ConfigurationRepository* ist dafür zuständig die einzelnen Konfigurationen auffindbar zu machen. Hierzu gibt es zwei Varianten der *resolve* Operation. Die Operation *resolve(String)* ermittelt eine Konfiguration basierend

auf deren Namen. Alternativ dazu können Konfigurationen auch über eine *ClientReference* gefunden werden. Diese *ClientReference* wird aus den Attributen der werkzeugspezifischen Events generiert. Eine Konfiguration kann damit basierend auf mehreren Attributen¹ eines Events aufgelöst werden. Die Logik der zweiten Variante wird durch die Implementierung der *matches(ClientReference)* Operation in der konkreten Konfiguration ermöglicht.

5.2. Semantische Integration

Für die semantische Integration gibt es nach Abschnitt 3.2 zwei wesentliche Mittel zu beachten: Datenintegration und funktionale Integration. Für die Datenintegration definiert man ein Vokabular und implementiert es in der funktionalen Integration Möglichkeiten dies zu verwenden. In Machina findet dies in zwei Schritten statt. Zunächst wird das Vokabular des Event-Prozessors definiert. Dieses Vokabular wird von den Rollen implementiert, die wiederum eigene rollenspezifische Vokabulare definieren. Dieses rollenspezifische Vokabular wird dann in den einzelnen Werkzeugmodulen implementiert, die diese bestimmte Rolle unterstützen wollen.

5.2.1. Event-Prozessor-Vokabular

Für den Event-Prozessor gibt es zwei Arten von Wörtern die jeweils in Nachrichten über den Nachrichtenbus verschickt werden. Es gibt zum Einen die eingehenden Ereignisse (*Events*) und zum Anderen die hierdurch entstehenden Aktionen (*Actions*).

Aktion

Im Event-Prozessor-Vokabular sind drei Kategorien von Aktionen definiert. Diese können von der Rolle benutzt werden, um ihre speziellen Rollenaktionen abzuleiten. Tabelle 5.1 fasst diese zusammen.

Nicht-blockierende Aktion Bei der nicht-blockierenden Aktion wird ein Effekt ausgelöst und der Kontrollfluss anschließend direkt fortgesetzt. Es wird nicht auf eine Reaktion des Empfängers gewartet. Diesem Aktionstyp können Argumente mittels einer Zeichenkette übergeben werden. Ein typisches Beispiel für ein nicht-blockierende Aktion ist das Versenden einer E-Mail.

Blockierende Aktion Bei der blockierenden Aktion wird ein Effekt ausgelöst und der Kontrollfluss wird solange angehalten, bis eine Reaktion durch ein Ereignis des Empfängers vorliegt. Durch diese Reaktion ergibt sich die Möglichkeit zusätzlich Daten zu übermitteln, die in den Workflowkontext eingetragen werden. Diese Daten bilden den Typ dieser Aktion. Auch diesem Aktionstyp können Argumente mittels einer Zeichenkette übergeben werden. Ein Beispiel für eine blockierende Aktion ist das Erzeugen eines Issues in einem Issue-Tracking-System mit anschließendem Warten bis dieser Issue abgeschlossen wurde.

¹insbesondere Attribute die nicht in *AbstractConfiguration* definiert sind

Prozessstartende Aktion Die prozessstartende Aktion wird durch ein Ereignis ausgelöst, das eine neue Workflowinstanz erzeugen soll. Wie bei der blockierenden Aktion können auch hier Daten im Ereignis übermittelt werden, die in den Workflowkontext eingetragen werden. Einer prozessstartenden Aktion können keine Argumente übergeben werden. Ein Beispiel für eine prozessstartende Aktion ist das Warten bis ein Issue angelegt wurde.

Aktion	Parameter	Rückgabewert
Nicht-blockierend	ja	nein
Blockierend	ja	ja
Prozessstartend	nein	ja

Tabelle 5.1.: Übersicht über Aktionstypen

Ereignis

Abbildung 5.2 beschreibt den generischen Event den der Event-Prozessor verarbeiten kann. Er besteht aus mehreren *RoleEvents*, die wiederum einen oder mehrere *TriggerKeys* sowie *EventArguments* besitzen. Mit Hilfe *TriggerKey* wird ein *Trigger*² aufgelöst.

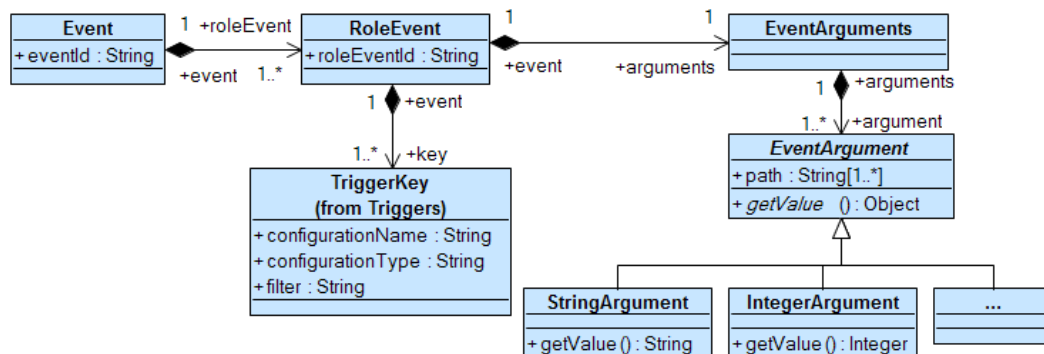


Abbildung 5.2.: Event und EventArgument

Mittels der *EventArguments* werden zusätzliche Argumente übergeben. Sie bilden den Rückgabewert der zugeordneten Aktion.

5.2.2. Rollenvokabular

In den Rollen Schnittstellen wird das Event-Prozessor-Vokabular formuliert (siehe Abbildung 5.3). Eine konkrete Rolle delegiert dann auf bestimmte Operationen, in den von

²genauer in Abschnitt 5.4.2

ihnen definierten Adapter-Schnittstellen. Diese Schnittstellen definieren somit ein eigenes Vokabular für die jeweilige Rolle. Zusätzlich zur Unterscheidung im Vokabular wurde nach prozessabhängig oder prozessunabhängig unterschieden. Die prozessabhängigen *BoundRole* und *BoundAdapter* verfügen über in der Prozessbeschreibung und Prozess-Binding festgelegte Optionen, die die Erzeugung von Aktionen ermöglichen. Die einfachen Varianten können nur auf Ereignisse in den Werkzeugen reagieren.

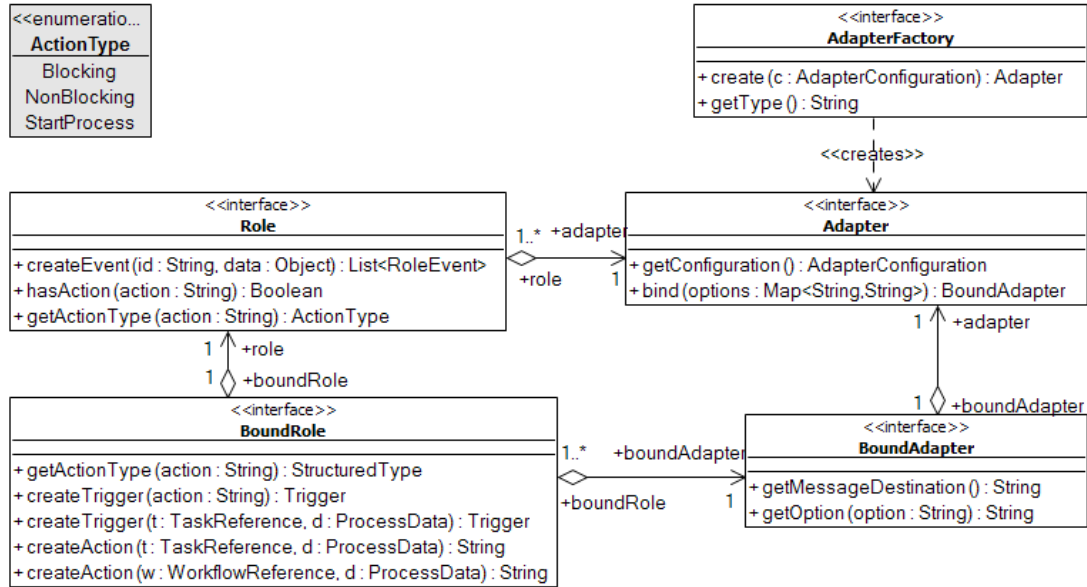


Abbildung 5.3.: Event-Prozessor-Vokabular beschrieben durch Rollenschnittstellen

Role und BoundRole

Diese Schnittstellen definieren das Vokabular, das der Event-Prozessor verwendet.

Role Realisierungen von *Role* übernehmen die Verantwortung für die Abbildung der prozessunabhängigen rollenspezifischen Ereignisse. Sie bekommen einen werkzeugspezifischen Event mittels der *createEvent(Object)* Operation übergeben. Diesen werkzeugspezifischen Event bilden sie mithilfe ihres zugeordneten *Adapters* auf den generischen Event für den Event-Prozessor ab. Falls dieser werkzeugspezifische Event für diese Rolle irrelevant ist, ist es auch möglich, dass daraus kein generischer Event entsteht.

BoundRole Realisierungen von *BoundRole* übernehmen die Verantwortung für die Erzeugung von prozessabhängigen Aktionen. Diese Nachrichten führen, sobald sie ausgeführt werden dazu, dass in den angebundenen Werkzeuge Effekte ausgelöst werden.

Adapter und BoundAdapter

Spezialisierungen dieser Schnittstellen formulieren das Vokabular, das die entsprechende Rolle verwenden kann. Für diese Rolle stellen sie die Datenintegration dar.

Adapter Spezialisierungen von *Adapter* formulieren für die entsprechende Rolle das prozessunabhängige Vokabular. Die Implementierungen in den Werkzeugmodulen bilden dieses rollenspezifische Vokabular auf entsprechende Ereignisse dieses Werkzeugs ab. Zusätzlich sorgt die *AdapterFactory* dafür, dass es nur eine Instanz je Konfiguration der Implementierung gibt³, damit kann es als Synchronisationspunkt genutzt werden.

BoundAdapter Spezialisierungen von „BoundAdapter“ formulieren für die entsprechende Rolle das prozessabhängige Vokabular. Die Implementierungen in den Werkzeugmodulen bilden dieses rollenspezifische Vokabular auf Effekte in diesem Werkzeug ab.

5.2.3. Zugriff

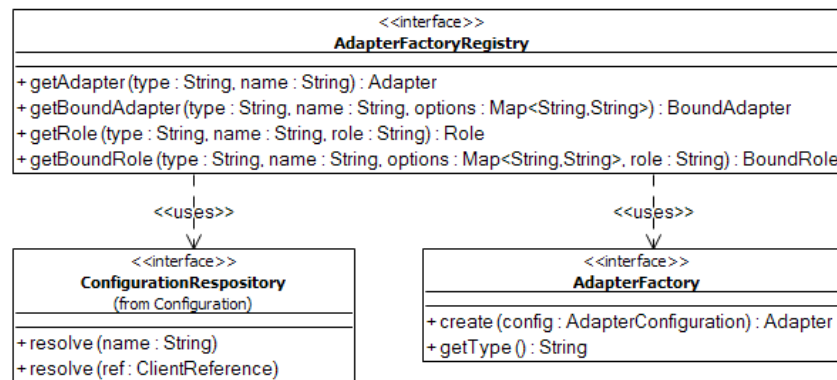


Abbildung 5.4.: AdapterFactoryRegistry

Den Zugriff auf diese Klassen regelt eine Implementierung der Schnittstelle *AdapterFactoryRegistry* in Abbildung 5.4. Bei dieser Klasse registrieren sich die Implementierungen der *AdapterFactories* aus den Werkzeugmodulen. Zusammen mit dem *ConfigurationRepository* kann sie Anfragen nach Instanzen von Implementierungen von *Adapter*, *BoundAdapter*, *Role* und *BoundRole* beantworten.

Erzeugen einer BoundRole

Das Sequenzdiagramm in Abbildung 5.5 beschreibt, wie die *AdapterFactoryRegistry* eine *BoundRole* erzeugt. Beim Aufruf der Operation *AdapterFactoryRegistry.getBoundRole()*

³ Ähnlich einem Multiton Pattern

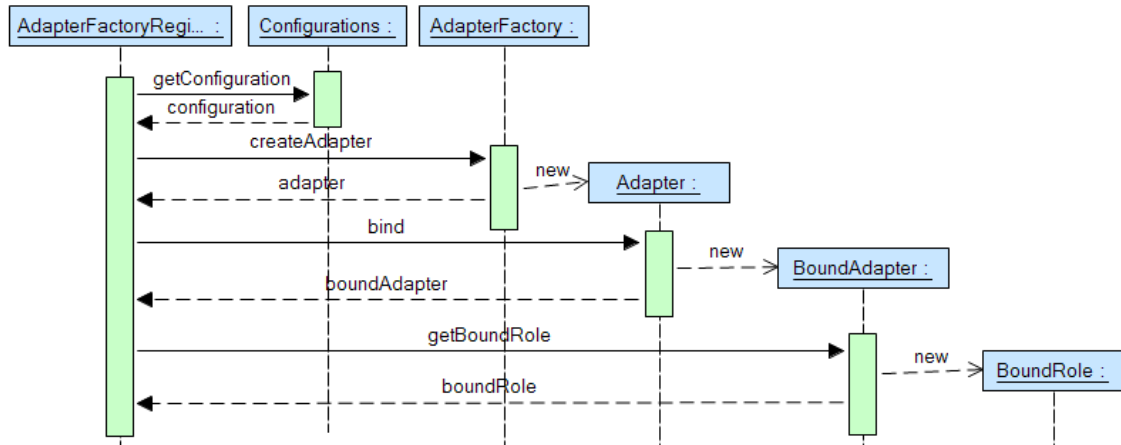


Abbildung 5.5.: Erzeugen einer BoundRole mittels AdapterFactoryRegistry

String type, String name, Map<String,String> options, String role) werden folgende Schritte ausgeführt.

1. Finden der *AdapterConfiguration* mittels *ConfigurationRepository* mit dem Namen *name*.
2. Erzeugen eines *Adapter* mittels der entsprechenden *AdapterFactory*. Falls für diese Konfiguration schon ein *Adapter* erzeugt wurde wird diese zwischengespeicherte Instanz benutzt.
3. Erzeugen eines *BoundAdapter* in der Operation *bind(adapterOptions:Map<String,String>)* mittels der zur Verfügung gestellten *adapterOptions*.
4. Erzeugen einer *BoundRole* in der Operation *getRole(role:String)* mit dem Namen *role*.

Jeder dieser Schritte kann durch ungültige Werte einen Fehler verursachen. Instanzen von *Adapter* bzw. *BoundAdapter* werden durch Abbruch dieser Sequenz an der entsprechenden Stelle erzeugt. Eine *Role* bekommt man durch den Aufruf *getRole(name:String)* am *Adapter*.

5.2.4. Issue-Tracking-Vokabular

Als Beispiel für ein rollenspezifisches Vokabular wurde das Vokabular für Issue-Tracking-Systeme gewählt. Das Vokabular wird durch die Schnittstellen *TrackerAdapter* *BoundTrackerAdapter*, sowie durch die darin verwendeten Klassen *IssueCreateAction* und *IssueCloseAction* definiert (siehe Abbildung 5.6). Jedes Werkzeugmodul, das die Funktionalität der Issue-Tracking Rolle anbieten will, muss diese Schnittstellen implementieren. Die Klassen *TrackerRole* und *BoundTrackerRole* implementieren die Schnittstellen *Role*

und *BoundRole*, indem sie ihr Vokabular auf das vom Event-Prozessor abbilden. Aufbauend auf die generelle Struktur der Rollen wird die Issue-Tracking-Abstraktion wie in Abbildung 5.6 definiert.

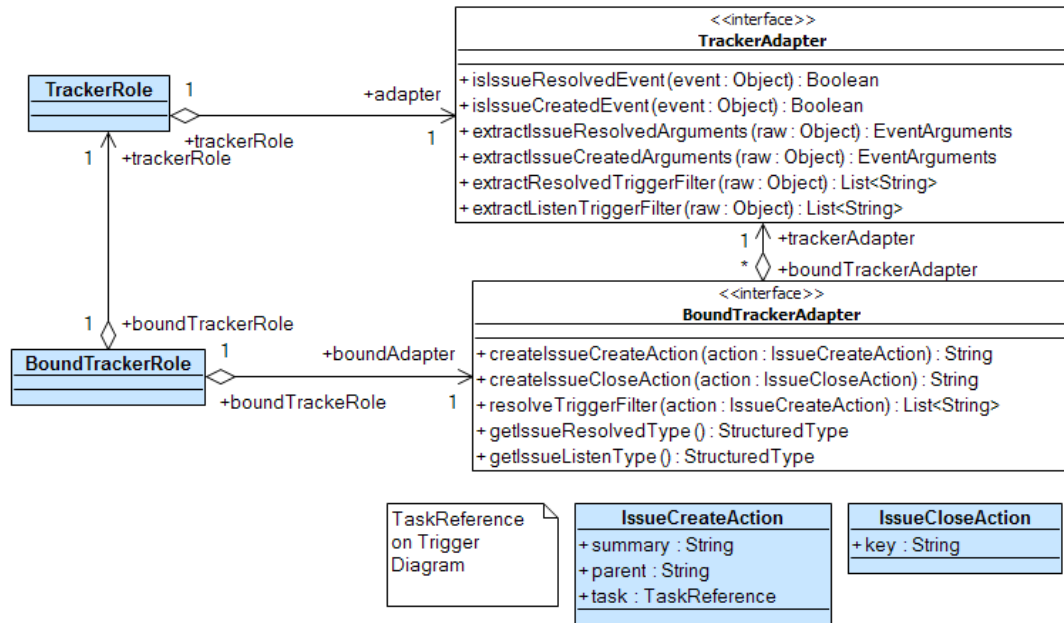


Abbildung 5.6.: Issue-Tracking-Vokabular

TrackerRole, BoundTrackerRole Die Implementierungen *Role* und *BoundRole* bilden das generische Vokabular des Event-Prozessors auf das von den Schnittstellen *BoundTrackerAdapter* und *TrackerAdapter* definierte Vokabular. Dabei werden folgende Rollenaktionen definiert.

1. Blockierende Aktion „createIssue“, es wird ein Issue erzeugt und anschließend darauf gewartet, dass dieses Issue abgeschlossen wurde.
2. Nicht-blockierende Aktion „closeIssue“, es wird ein Issue abgeschlossen.
3. Prozessstartende Aktion „listen“, es wird darauf gewartet, dass ein Issue mit einem bestimmten Typ angelegt wird.

TrackerAdapter Spezialisierung der *Adapter* Schnittstelle, die Operationen enthält, um das Verarbeiten der eingehenden Events zu erlauben. Hiermit wird das Vokabular definiert, das die *TrackerRole* verwenden kann. Mit den Prädikaten *is** kann die *TrackerRole* testen, ob der entsprechende werkzeugspezifische Event als dieser Event interpretiert werden kann. Über die *extract** Operationen kann er anschließend die Nutzdaten aus den werkzeugspezifischen Events extrahieren. Hier wurde bewusst darauf verzichtet vom *TrackerAdapter* direkt ein explizites Zwischenformat *IssueEvent* erzeugen zu lassen. Durch die Interaktivität ist sichergestellt, dass s

bei neuen Eventtypen Übersetzungsfehler durch nicht implementierte Operationen entstehen. Werkzeugmodulentwickler müssen sich also bewusst entscheiden Eventtypen nicht zu unterstützen. Es folgt dem Open-Close Prinzip des objektorientierten Designs.

BoundTrackerAdapter Spezialisierung der *BoundAdapter* Schnittstelle die Operationen enthält, die das Erzeugen von Aktionsnachrichten für Issue-Tracking erlauben. Hiermit wird das Vokabular definiert, das die *BoundTrackerRole* verwenden kann.

IssueCreateAction und IssueCloseAction Kapseln die Daten, die benötigt werden, um die entsprechende Aktionsnachricht vom Werkzeugmodul erzeugen zu lassen. Sie sind nur prototypisch angelegt, weitere Arbeit ist notwendig, damit diese mehr Funktionalität anbieten können.

5.3. Nachrichtenbus

Diese Komponente stellt die zuverlässige Kommunikation zwischen der *Integration*-Komponente und der *Execution*-Komponente her. Sie kann aber auch für die Kommunikation der angebundenen Werkzeuge mit ihren Werkzeugmodulen benutzt werden. Bei ihm können sich die einzelnen Komponenten für bestimmte Themen registrieren (subscribe). Dadurch wird eine bestimmte Operation aufgerufen sobald eine Nachricht für diese Komponente eingetroffen ist. Zum Versenden veröffentlicht (publish) man eine Nachricht unter einem bestimmten Thema. Ein besonderes Problem, das es zu lösen gilt, ist das mehrfache Empfangen von Nachrichten, welches zu mehrfachen Ausführen führen könnte. Es könnten Workflowinstanzen mehrfach entsteht, oder auch mehrfach Effekte in den Werkzeugen ausgelöst werden. Aber auch das Verlieren von Nachrichten führt zu unerwünschten Situationen. Diese Probleme umgeht man, wenn man darauf achtet, dass eine Nachricht genau einmal ausgeführt wird (exactly once).

5.3.1. Queued Transactions

Eine einfache Möglichkeit dieses *exactly once* zu erreichen, ist die komplette Kommunikation in einer Transaktion auszuführen. Das sorgt dafür, dass nur im Erfolgsfall Ergebnisse sichtbar werden. In einem verteilten System führt das zu einer verteilten Transaktion (distributed transaction) [GR93, vgl. S. 202]. Die Einfachheit dieses Ansatzes muss man sich leider mit einigen Nachteilen erkaufen. Der Größte davon ist die Verfügbarkeit⁴; das System ist nur handlungsfähig, wenn sämtliche an der Transaktion beteiligten Knoten verfügbar sind. Das deutlichste Ausschlusskriterium ist jedoch was im Falle eines Rollbacks passieren müsste. Es müssten sämtlichen Artefakte der Transaktion entfernt werden und damit auch die von Nutzern erzeugten Artefakte, z.B. neue Einträge im Issue-Tracking-System.

Als Alternative bietet sich das Konzept Queued Transactions an [BN09, S. 99 ff.]. In diesem werden Transaktionsnachrichten zwischen verschiedenen Knoten mittels Warte-

⁴Würde die Anforderung REQ 2 verletzen

schlangen ausgetauscht. Hierbei muss aber darauf geachtet werden eine genaue Abarbeitungsreihenfolge einzuhalten, Abbildung 5.7 beschreibt diesen Vorgang. Eine Transaktion wird damit über mehrere Knoten hinweg verteilt, ohne dass eine direkte Abhängigkeit besteht. Um die Knoten zusätzlich von der Verfügbarkeit der Warteschlange zu entkoppeln ist es notwendig lokale Warteschlange einzuführen, die verwendet werden müssen, falls die entfernte Warteschlange nicht verfügbar ist [BN09, vgl. S. 111].

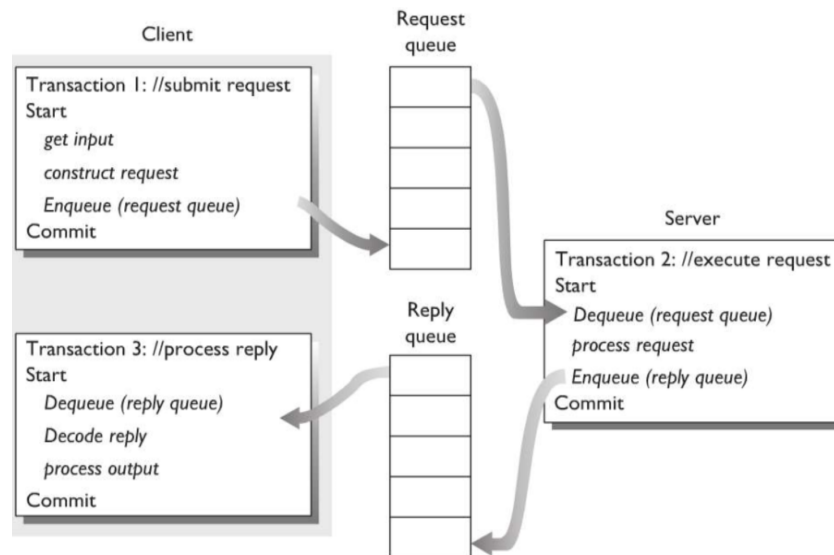


Abbildung 5.7.: „Running a Request as Three Transactions“ [BN09, S. 103]

Schema

Für die Verarbeitung einer Transaktionsnachricht ergibt sich das Schema in Tabelle 5.2. Es besteht aus fünf Schritten, die unbedingt in dieser Reihenfolge durchgeführt werden müssen. Insbesondere Schritt Drei, das Ausführen der Nachricht, muss vollständig abgeschlossen sein, bevor durch Schritt Vier die Ergebnisse veröffentlicht werden. Des weiteren muss bedacht werden, das alles in einer Transaktion abläuft. Es finden alle Schritte nach den üblichen ACID-Transaktionen statt.

1	Starten der Transaktion
2	Dequeue der Nachricht
3	Ausführen der Nachricht
4	Enqueue der Ergebnisse
5	Commit der Transaktion

Tabelle 5.2.: Schema für einen Verarbeitungsschritt während der Queued Transaction

Strategien für das Ausführen der Nachricht

Für das Ausführen der Nachrichten in diesem Schema gibt es bestimmte Voraussetzung. Falls die Ausführung rückgängig machbar ist, dann gibt es einen Rollback. Damit kann dieser Vorgang in der gleichen Transaktion wie das „dequeue/enqueue“ ausgeführt werden. Falls dies nicht möglich ist, kann man nach [BN09, vgl. S. 107] auf zwei weitere Arten von Operationen zurückgreifen. Diese erzeugen zumindest ein beobachtbares Verhalten wie ein *exactly once*, sobald die Nachricht einmal vollständig verarbeitet wurde. Dies wird erreicht, da sie duplizierte Nachrichten tolerieren können und man Nachrichten so oft wiederholt, bis sie garantiert ausgeführt wurden.

1. Verwendung von rein *idempotenten* Operationen in der Ausführung. Bei idempotenten Operation hat das mehrfache Ausführen keine weiteren Auswirkungen.
2. *Testbare* Verarbeitung, man kann feststellen ob eine Nachricht schon verarbeitet wurde. Man führt sie nur aus, wenn dieser Test negativ ist. Wichtig bei testbarer Verarbeitung ist, dass man die Ergebnisse speichern muss, da man diese nicht durch erneute Ausführung erneut erzeugen kann.

5.3.2. Nachrichtenverarbeitung

Basierend auf den *Queued Transaction* aus dem letzten Abschnitt ergibt sich dann das Schema in Tabelle 5.3 für die Nachrichtenverarbeitung das in Machina Verwendung findet.

1	Empfang einer Nachricht N
2	Ausführen der Nachricht N mit den Ergebnissen E .
3	Senden der Ergebnisse E .
4	Bestätigen des Empfangs der Nachricht N
5	Löschen der zwischengespeicherte Ergebnisse E

Tabelle 5.3.: In Machina verwendetes Schema für die Nachrichtenverarbeitung

Dieses Schema kann Duplikate erzeugen, da erst die explizite Bestätigung verhindert dass sich das Versenden der Ergebnisse wiederholt. Diese Duplikate müssen während der Ausführung toleriert⁵ werden. Für idempotente Operationen ist es unnötig die Ergebnisse zwischenzuspeichern, da man diese durch erneutes Ausführen erhält. Es entfällt somit auch Schritt Fünf das Löschen der Ergebnisse.

5.3.3. Zusammenfassung

Nachstehend sind nun die notwendigen Eigenschaften, die eine Implementierung des Nachrichtenbusses erfüllen muss, zusammengefasst. Einige dieser Eigenschaften sind direkte Folgen der Anforderungen an Machina.

⁵Idempotente und testbare Operationen machen das ohnehin

Eigenschaften

1. Asynchronität: Sender und Empfänger sind unabhängig. Ermöglicht die Anforderung REQ 2.
2. Zuverlässigkeit: Sobald eine Nachricht angenommen wurde, muss diese so oft zugestellt werden, bis das Empfangen bestätigt wurde. Ermöglicht die Anforderung REQ 16.
3. Reihenfolgeerhaltend: Nachrichten vom gleichem Sender zum selben Thema kommen in der Reihenfolge des Sendens an.

Angebotene Operationen

Der Nachrichtenbus muss für die anderen Komponenten folgende Operationen anbieten.

subscribe Meldet eine Klasse beim Nachrichtenbus für ein bestimmtes Thema an. Die Klasse wird über einen Callback Mechanismus über neue Nachrichten informiert. Das Ausliefern entspricht einem „enqueue“ im Sinne der Transaktionswarteschlange.

publish Veröffentlicht eine Nachricht zu einem bestimmten Thema. Dies entspricht dem „enqueue“ einer Transaktionswarteschlange.

ack Markiert die aktuelle Nachricht als „erfolgreich ausgeführt“. Sie kann hiermit gelöscht werden. Dies entspricht dem „commit“ in einer Transaktionswarteschlange.

nack Markiert die aktuelle Nachricht als „nicht erfolgreich ausgeführt“. Es muss ein erneuter Zustellungsversuch unternommen werden. Dies entspricht dem expliziten Rollback der Transaktion.

5.4. Event-Prozessor

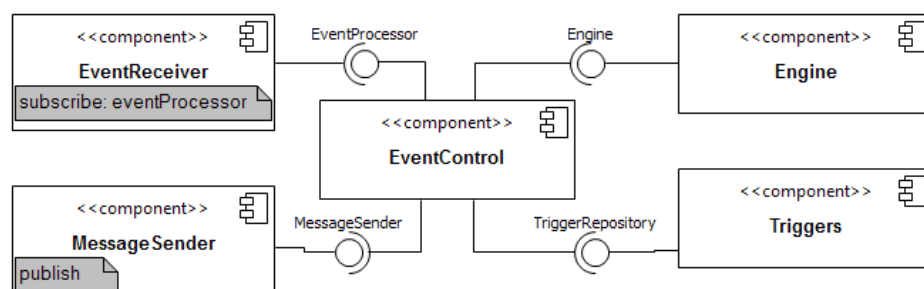


Abbildung 5.8.: Aufbau des Event-Prozessors

In der Event-Prozessor-Komponenten (Abbildung 5.8) werden die, durch die Vorverarbeitungsschritte in der *Integration*-Komponente entstandenen, generischen Events verarbeitet. Sie ist aufgeteilt in folgende Teilkomponenten:

EventReceiver Der EventReceiver empfängt Nachrichten vom Nachrichtenbus. Durch Demarshalling wird der darin enthaltene Event extrahiert.

EventControl Koordiniert die Ausführung der Events. Diese Komponente muss das Schema für die Nachrichtenverarbeitung (siehe 5.3.2) benutzen.

Triggers Stellt einen persistenten Speicher für Trigger bereit, der über das *TriggerRepository* verfügbar ist.

Engine Die Komponente führt die *TriggerCommands* aus und erzeugt hierbei neue Aktionsnachrichten. Hier muss sichergestellt sein, dass diese Ausführung keinerlei sichtbare Ergebnisse erzeugt.

MessageSender Durch Marshalling werden die Aktionen in Aktionsnachrichten umgewandelt und anschließend mit dem Nachrichtenbus versendet.

5.4.1. EventControl

Die Komponente *EventControl* ist dafür zuständig die Ausführung eines Events zu koordinieren. Hierbei muss das *EventControl* darauf achten, dass ihm diese Events möglicherweise mehr als einmal zugestellt werden. Selbst im Falle seines Ausfalls darf es nicht zu doppelt ausgeführten Events kommen. Das *EventControl* implementiert also das nach Abschnitt 5.3.2 diskutierte Schema für die Nachrichtenverarbeitung. Es ist in Tabelle 5.4 noch einmal für das *EventControl* umformuliert.

1	Empfang eines Events E
2	Ausführen des Events E durch die Komponenten <i>Engine</i> und <i>Triggers</i> . Hierbei entstehen die Nachrichten N . Die zwischengespeichert werden müssen.
3	Versand der Nachrichten N
4	Bestätigen des Empfangs des Events E
5	Löschen der zwischengespeicherten Nachrichten N

Tabelle 5.4.: Schema für Verarbeitung im EventControl

5.4.2. Triggers

Die *Triggers*-Komponente stellen die Verbindung von eingehenden Events mit den Workflows und Workflowinstanzen her. Wenn eine Aktion in einer Workflowinstanz auf ein externes Ereignis wartet, hinterlegt diese einen *TriggerCommand* mit den benötigten Informationen um die Blockierung aufzulösen. Dieser *Trigger* besitzt mindestens einen *TriggerKey*, mit dem er aufgefunden werden kann. Dieser Key bzw. diese Keys müssen so beschaffen sein, dass ein Event genug Informationen besitzt, um einen Key zu erzeugen.

Durch die Möglichkeit das mehrere Keys einen Command zugewiesen sind, bzw. da auch einem Event mehrere Keys zugeordnet werden können. Ist eine eingeschränkte Form einer unscharfen Abbildung möglich. Abbildung 5.9 zeigt die Schnittstellen, über die man diese Komponente benutzen kann.

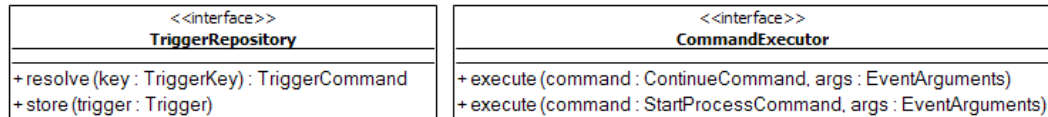


Abbildung 5.9.: TriggerRepository und CommandExecutor Schnittstelle

TriggerRepository Stellt die nach außen sichtbare Schnittstelle dieser Komponente dar. Sie ist zuständig für das Speichern und Wiederfinden der einzelnen *Trigger*-Instanzen.

CommandExecutor Realisierungen dieser Schnittstelle sind dafür zuständig *TriggerCommands* auszuführen.

Trigger

Ein *Trigger* (Aufbau siehe Abbildung 5.10) ist ein in persistentes hinterlegtes zusammengesetztes Objekt, das von der Engine ausgeführt wird. Es wird durch mindestens einen *TriggerKey* eindeutig identifiziert. Erstellt werden Trigger auf zwei unterschiedliche Arten. Während der Erzeugung einer Nachricht für eine blockierende Aktion, wird ein *Trigger* mit *ContinueCommand* erzeugt. Während der Übersetzer-Pipeline wird durch eine prozessstartende Aktion ein *Trigger* mit *StartProcessCommand* erzeugt.

TriggerKey speichert die zum Auffinden des zugehörigen *TriggerCommand* notwendigen Information ab. Es ist möglich, dass es mehrere *TriggerKeys* für einen *TriggerCommand* gibt.

ContinueCommand Dieser *TriggerCommand* speichert eine *TaskReference*, die es ermöglicht eine Blockierung durch eine Aktion in einer Workflowinstanz zu beenden.

StartProcessCommand Dieser *TriggerCommand* speichert eine *WorkflowReference*, die es ermöglicht eine neue Workflowinstanz zu starten. Dieser Command wird ausschließlich während des Übersetzungsvorgangs erzeugt und beim Deployment gespeichert.

5.4.3. Engine

Die Engine (siehe Abbildung 5.11) enthält ein Programm das *TriggerCommands* ausführt und damit neue Aktionen und Trigger erzeugt. Hierzu realisiert sie die Schnittstellen *CommandExecutor* und *MessageCache*. Für die Verwendung in Übersetzer-Pipeline ist

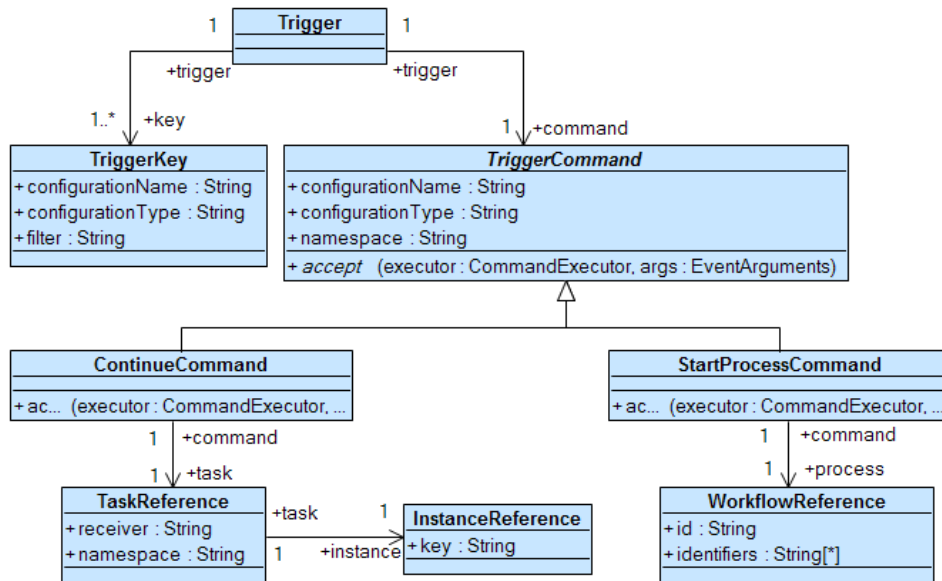


Abbildung 5.10.: Aufbau der Trigger

es zusätzlich notwendig neue Programme einbringen zu können. Hierzu wird das Schnittstelle *DeploymentHandler*⁶ realisiert.

CommandExecutor Die unter 5.4.2 beschriebene Schnittstelle zum Ausführen von *TriggerCommands*.

MessageCache Durch diese Schnittstelle stellt die Engine die erzeugten Aktionen und Trigger für das *EventControl* bereit.

⁶In Abbildung 5.11 nicht dargestellt

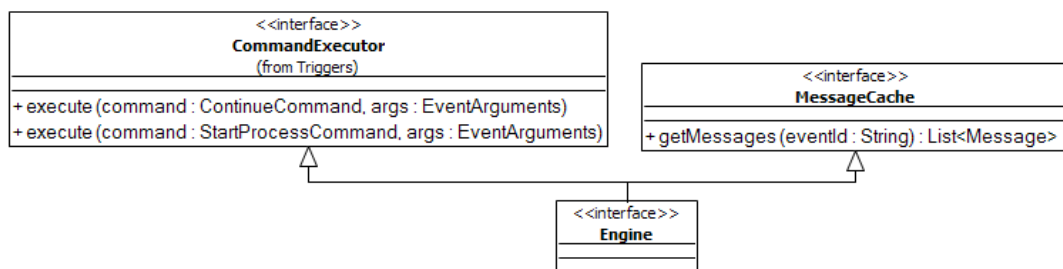


Abbildung 5.11.: Engine Schnittstelle

5.5. Übersetzer

Die Architektur des Übersetzers ist eine typische *Pipe and Filter* [BMR⁺96, S. 53 ff.] Architektur. Er besteht aus mehreren weitgehend unabhängigen *Filtern*, die hier⁷ als Komponenten dargestellt werden. Sie werden mit *Pipes* verbunden, die hier⁷ als Schnittstellen-Realisierungen bzw. Abhängigkeiten dargestellt werden.

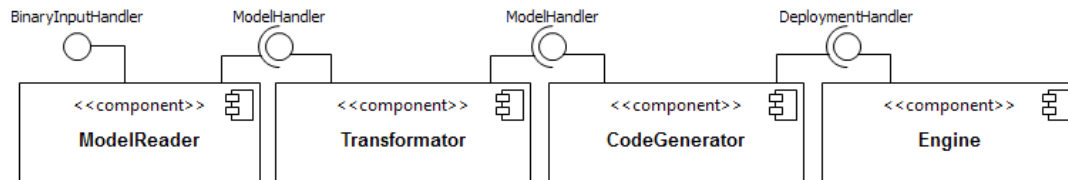


Abbildung 5.12.: „Pipe and Filter“ Struktur der Übersetzer-Pipeline

Die Hauptfilter im Überblick

ModelReader Eine Komponente die einen binären Datenstrom erhält und diesen in eine Instanz des Machina-Prozessmodells umwandelt. Der Umwandlungsprozess ist abhängig vom verwendeten Datenformat. Es lässt sich kein generisches Design erzeugen⁸, das beliebige Datenformate einlesen könnte. Ein Design für einen speziellen *ModelReader*, dem es möglich ist einen Prozess im Stages-Format einzulesen, befindet sich im Abschnitt 7.7.1. Dieser Prozess wird nach dem Einlesen den weiteren Schritten der Pipeline übergeben.

Transformator Diese Komponente steht stellvertretend für eine Reihe von identisch aufgebauten Komponenten. Diese Komponenten erhalten einen Prozess, führen eine Transformation auf diesem aus und übergeben den veränderten Prozess⁹ an den nächsten Transformator. Intern bestehen die Transformatoren aus einem oder mehreren Visitor Pattern [GHJV94, S. 331 ff.].

Codegenerator Im Anschluss an die Transformatoren wird der Workflow durch einen Codegenerator in eine für die Engine verwendbare Repräsentation¹⁰ umgewandelt.

Engine Im Zusammenspiel mit der Übersetzer-Pipeline nimmt diese Komponente den vom Codegenerator erzeugten Workflow entgegen und speichert diese dauerhaft für Benutzung in der Laufzeitumgebung. Für die weitere Benutzung ist es auch notwendig, dass die Engine passende Trigger (siehe Abschnitt 5.4.2) zum starten dieses Workflows anlegt.

⁷In Abbildung 5.12

⁸Zumindest im Rahmen dieser Arbeit

⁹Der Prozess wird zu einem Workflow, sobald der Binder ausgeführt wurde

¹⁰Im Prototypen wird BPMN erzeugt

5.5.1. Machina-Prozessmodell

Die Hauptdatenstruktur für den Übersetzer ist das Machina-Prozessmodell (Abbildung 5.13). Mit ihr wird ein Prozess während der Transformationen repräsentiert. Es ist gemäß dem Composite Pattern [GHJV94, S. 163 ff.] aufgebaut.

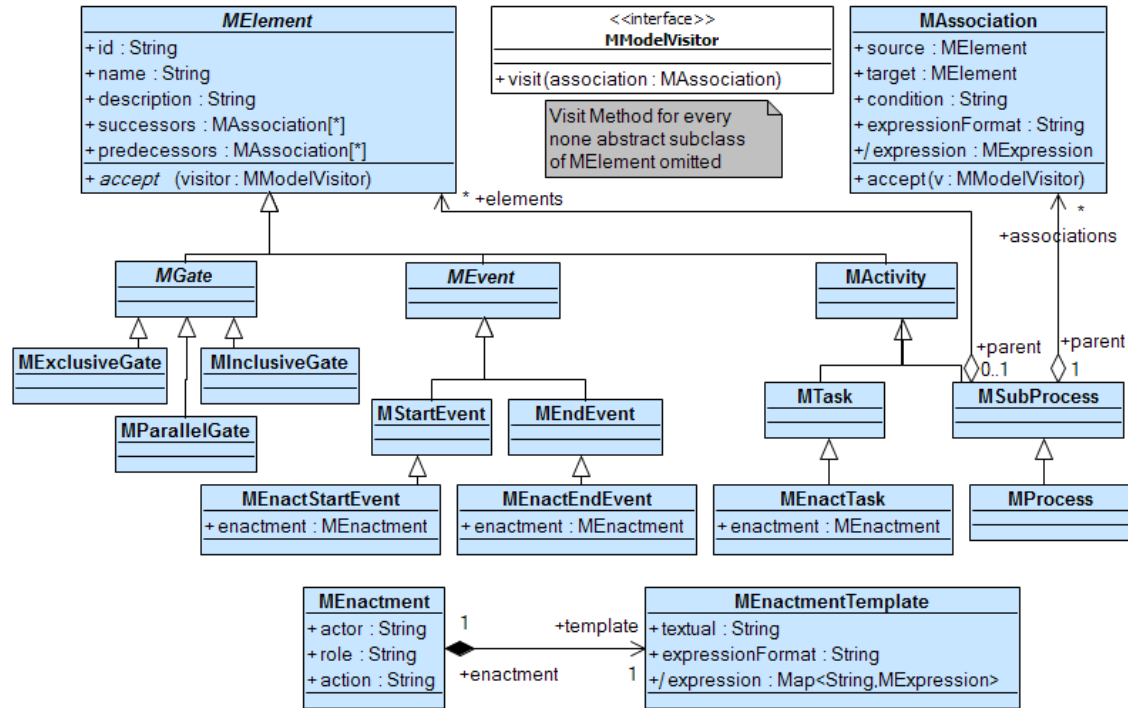


Abbildung 5.13.: Machina-Prozessmodell

Designziel dieses Modells war es BPMN¹¹ erzeugen zu können. Es war also naheliegend ein Modell zu nutzen, das diesem sehr ähnlich ist. Um die Transformationen zu vereinfachen bzw. erst zu ermöglichen, wurden einige Elemente eingeführt, die dieses nicht direkt vorsieht. Insbesondere wurden Elemente eingefügt die ein *MEnactment* haben. Ein weiteres Ziel war es mit einfachen Mitteln Instanzen des Modells erzeugen zu können.

MActivity Diese Elemente markieren die Aktivitäten und Tätigkeiten, die während eines Prozesses ausgeführt werden. Hier gibt es zum einen den *MTask* der einzelne Aktionen des Benutzers bzw. Systems beschreibt, sowie den *MSubProcess* der sich aus *MElements* zusammensetzt. Ein Spezialfall für den *MSubProcess* ist der *MProcess* das den ganzen Prozess beschreibt.

MEvent Element zum markieren von Start bzw. Ende eines *MSubProcess*. Als Variante können diese Elemente auf externe Ereignisse warten oder Ausführungen beenden.

¹¹Business Process Modelling and Notation [OMG11]

MGate Elemente die den Kontrollfluss leiten. Die Semantik wurde direkt aus BPMN übernommen.

MAssociation Stellt die Verbindung der einzelnen Element untereinander her. Sie können eine Expression besitzen, die nach einem *MGate* entscheiden ob diese Assoziation zur Laufzeit zulässig ist.

MEnactment Das *MEnactment* ist die besondere Erweiterung des Prozesses. Mit ihm wird auf konzeptioneller Ebene definiert, was eine Aktivität bedeuten soll.

Actor: Dieses Feld stellt einen prozessmodellweit eindeutigen Identifikator eines Werkzeugs dar. Enactments mit dem gleichen *Actor* werden immer auf dieselbe Konfiguration abgebildet.¹²

Role: Legt fest welche Rolle der *Actor* in diesem Enactment einnimmt. Dafür ist es notwendig, dass das abgebildete Werkzeug auch diese Rolle und damit deren Vokabular implementiert.

Action: Dieses Feld bestimmt welche der Rollenaktionen von diese Enactment ausgeführt wird.

Template: Ein über Komposition angebundenes Textfeld, das als Platzhalter für an dieses Enactment übergebene Argumente dient. Es wird von der Rolle ausgewertet. Es kann *MExpressions* enthalten und hat damit Zugriff auf Einträge im Workflowkontext.

Für die Attribute des Types *MExpression* wurde ein unabhängiges Modell (Abbildung 5.14), das Machina-Expressions-Modell, definiert. Damit kann auf Elemente des Workflowkontext zugegriffen werden.

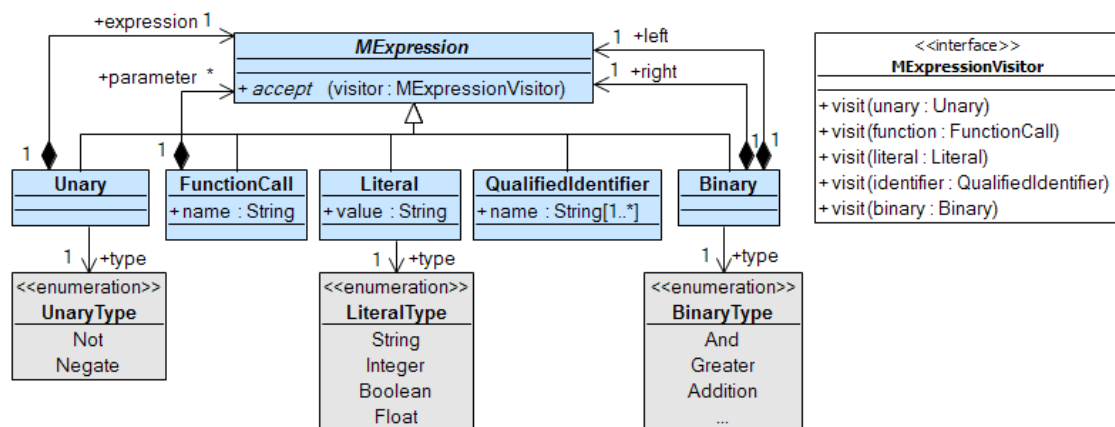


Abbildung 5.14.: Machina-Expressions-Modell

¹²Es können aber mehrere unterschiedliche Actors auf dieselbe Konfiguration abgebildet werden.

5.5.2. Transformationen

Die Transformationen führen ein Prozessmodell von einem Zustand in einen anderen über. Hierzu muss der Transformator die *ModelHandler* Schnittstelle implementieren (Abbildung 5.15).

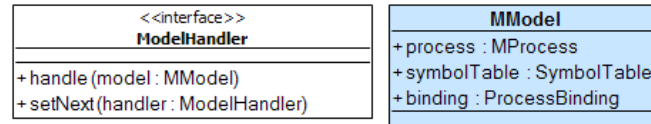


Abbildung 5.15.: ModelHandler Schnittstelle

Über die *handle(MModel)* Operation bekommt der Transformator das Prozessmodell übergeben. Er führt dann seine Transformation auf diesem Prozessmodell aus und übergibt es an den Transformator, der ihm durch die *next(ModelHandler)* Operation zugewiesen wurde.

Wichtige Transformatoren

Validierung Diese Transformation stellt sicher, dass sich der Prozess in einem zulässigen Zustand befindet. Da zu diesem Zeitpunkt in der Übersetzer-Pipeline noch nicht alle Transformationen ausgeführt wurden, lösen lediglich die Fehler einen Abbruch der Generierung aus, die hier schon sichtbar sind. Fehler, die durch nachfolgende Schritten korrigiert werden können, werden ignoriert. Beispiele für einen Validierungsfehler sind: kein vorhandener *MStartEvent* oder Prozesselemente die nicht von einem *MStartEvent* erreichbar sind.

Expression Parser Dieser Transformation sucht im Prozessmodell nach textuellen Ausdrücken und führt diese in eine Instanz des Machina-Expressions-Modells (siehe 5.14) über. Expression befinden sich an den bedingten Assoziationen (*MConditionalAssociation*) und in den *MEnactmentTemplates* der *Tasks* und *Events* mit *MEnactment*. Falls nicht alle Ausdrücke parsebar sind, wird auch hier eine Fehlermeldung erzeugt und die Generierung abgebrochen.

Binder Er ist dafür zuständig den Prozess um die Informationen in der *ProcessBinding* anzureichern und damit einen Workflow zu erzeugen. Genauer wird dieser Vorgang im Abschnitt 5.6 behandelt.

5.5.3. Codegenerator

Nachdem alle Transformatoren ausgeführt wurden, kann der Codegenerator den Workflow übernehmen und es in ein Format bringen, das die Engine verwenden kann. Ein Beispiel wie man dies durchführen kann ist im Abschnitt 7.8 zu finden.

5.5.4. Engine

Die Engine Komponente dient als Ziel der Übersetzer-Pipeline. Hierzu realisiert sie die Schnittstelle *DeploymentHandler*. Sie speichert das generierte Workflowmodell ab und erzeugt die *Trigger* mit *StartProcessCommand*. Da die Engine später die einzige Komponente ist, die mittels *StartProcessCommand* diesen Workflow finden muss, kann dieser im Feld *key* beliebige implementierungsspezifische Daten ablegen, die ihm dabei helfen diesen Workflow aufzulösen.

5.6. Binding

Beim Binding wird der Übergang von Prozess auf Workflow vollzogen. Es ist ein Teil des Übersetzers.

5.6.1. Enactment-Binding

Für diesen Vorgang wird das *MEnactment* aus dem Machina-Prozessmodell um ein *MEnactmentBinding* erweitert (siehe Abbildung 5.17). Das *MEnactment* beschreibt die angebundenen Werkzeuge auf einer konzeptionellen Ebene, während das *MEnactmentBinding* die Werkzeuge auf operative Ebene beschreibt.

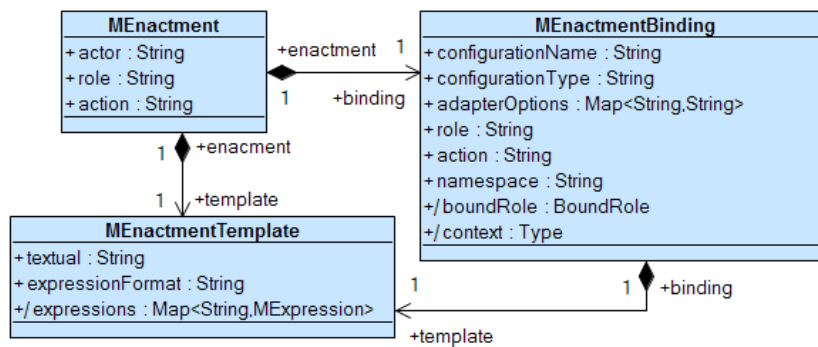


Abbildung 5.16.: MEnactment mit MEnactmentBinding

Hierzu fehlen offensichtlich noch einige Informationen, für den Übergang *Adapter* auf *BoundAdapter* sind die *AdapterOptionen* notwendig. Diese Optionen sind projektspezifisch und sollen deswegen nicht im Prozessmodell direkt abgelegt werden¹³. Aus diesem Grund wurde eine weitere Datenstruktur geschaffen das *ProzessBinding* (Abbildung 5.17). Für jedes *MEnactment* im Prozessmodell werden nun folgende Schritte ausgeführt:

1. Suchen eines *Actor* im *ProcessBinding* mit einer *id*, die dem Attribut *actor* im *MEnactment* entspricht. Damit können die Attribute *configurationName*, *configurationType* und *adapterOptions* zugewiesen werden.

¹³Projekte sind auf der operativen Ebene

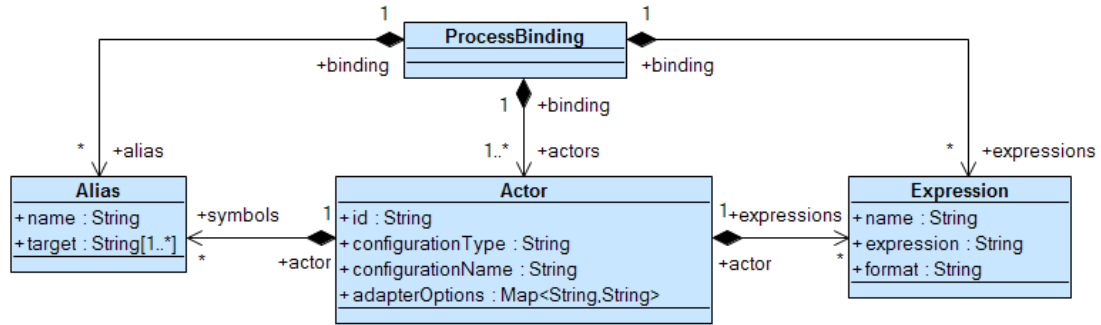


Abbildung 5.17.: Modell des Prozessbindings

2. Kopieren der Attribute *role* und *action* sowie des *ME enactmentTemplates*.
3. Aus der Konfiguration wird mit den *adapterOptions* zunächst ein *BoundAdapter* erzeugt, der wiederum eine *BoundRole* mit *role* erzeugen kann.
4. Als letzter Schritt wird der Typ dieses *ME enactmentBindings* bestimmt. Der Typ eines *ME enactments* sind die Daten, die durch dessen Ausführung vom angebundenen Werkzeug an den Workflow zurückgegeben werden. Hierzu wird zunächst bei der *boundRole* angefragt, welche Daten diese Aktion liefern kann. Die Rolle wird diese Anfrage üblicherweise ihren gebundenen Adapter weitergeben, der den Zustand des angebundenen Systems ausliest und damit die möglichen Daten ableitet.¹⁴ Repräsentiert wird diese Information dann in Form eines *StructuredTypes* aus Abbildung 5.18. Anschließend wird dieser Typ zusätzlich um die mit diesem *Actor* assoziierten *Alias* und *ME expressions* erweitert.

Ein Enactment, das nicht vollständig abbildbar ist, ist ein Modellierungsfehler. Diese muss als solcher in der Übersetzer-Pipeline erkannt und dem Modellierer mitgeteilt werden REQ 12.

5.6.2. Workflowparameter-Binding

Der Workflowtyp stellt die Zusammenfassung der einzelnen Enactment-Typen dar. Es wird durch die Struktur in 5.18 repräsentiert - einem Compositepattern [GHJV94, S. 164 ff.]. Es ist ein Baum der möglichen Daten, die von den angebundenen Werkzeugen an den Workflow übergeben werden könnten, darstellt.

Named Element

Alle *Named* Elemente haben einen Hauptnamen *name* und möglicherweise Alternativnamen *alias*, die während der Auflösung auch verwendet werden können.

¹⁴Das ist nur eine mögliche Vorgehensweise. Diese wird bei der *TrackerRole* zusammen mit dem Jira-Modul verwendet.

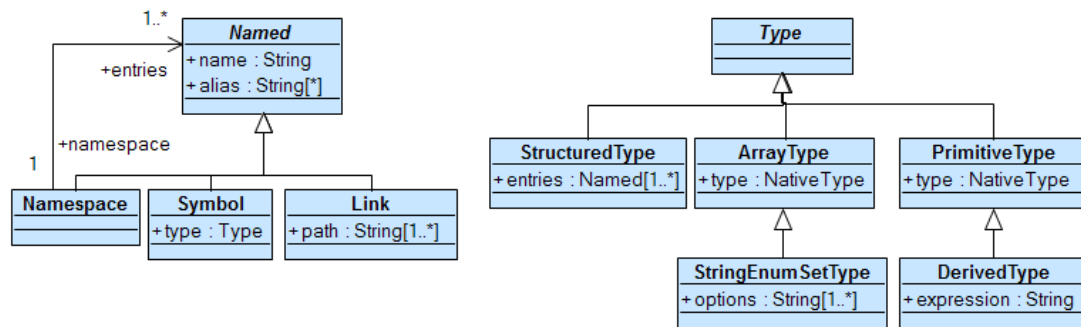


Abbildung 5.18.: Klassen zum Repräsentieren des Workflowtyps und Enactment-Typen

Symbol Bindet einen Namen direkt auf einen Typ.

Link Bildet einen Namen auf einen neuen Pfad ab.

Namespace Öffnet einen neuen Namensraum in dem sich andere *Named* Element befinden können.

Type

Mit diesen Klassen wird einem Symbol ein Typ zugewiesen.

StructuredType Ein Typ der aus *Named*-Elementen besteht. Er ist ein *Namespace* ohne eigenen Namen.

PrimitiveType Ein expliziter Type, z.B. Integer oder String.

DerivedType Ein Typ der auf auf einen Ausdruck abgebildet wird.

Gebildet wird es indem sämtliche Typen der einzelnen *MEenactmentBindings* im Prozess zusammengelegt werden. Der Workflowtyp wird als Instanz der *StructuredType*-Klasse angelegt. Jeder Typ eines *MEenactmentBindings* wird nun durch ein *Symbol* mit seinem Attribut *namespace* als *name* in diese Instanz eingetragen. Erweitert wird dieser Baum noch durch globale *Alias* und *Expressions* aus dem *ProcessBinding*. Im Listing in 5.1 wurde versucht so einen Baum darzustellen.

Syntax: [name] "alias"

```

[_TEOT0HNqEf-688ivE6zmZA] "Analyse Change Request"
  [key]
  [customfield_10001]
  [accepted] derived contains(
    "_TEOT0HNqEf-688ivE6zmZA ":"customfield_10001",
    "10000"
  )

```

```

    )
    [_5Z950CoNE-ajb-6u54SVDQ] "Feasibility Study"
    [key]
    [customfield_10002]
    [feasible] derived contains(
        "_5Z950CoNE-ajb-6u54SVDQ":"customfield_10002",
        "10010"
    )

```

Listing 5.1: Beispiel für einen Workflowtyp

Ein Workflowparameter ist ein Pfad zu einem Blatt in diesem Baum, der nur die richtigen Namen verwendet, keine Aliasnamen. Diese können in beliebigen *MExpressions* verwendet werden. Damit können auf Basis von Workflowparametern Entscheidungen in *MAssociations* getroffen werden oder die Werte über die *MEnactmentTemplates* an die Rollen für die Verwendung als Aktionsparameter übergeben werden. Während der Laufzeit wird über Instanzen von *EventArguments* und *EventArgument* (Abbildung 5.2) dieser Pfad nachgebildet und die Argumente durch Eintragen in den Workflowkontext unter dem entsprechenden Pfadnamen zurückgegeben.

5.6.3. Symboltabelle

Als letzter Schritt im Bindevorgang wird eine Symboltabelle erstellt (siehe Abbildung 5.19). Dieser stellt sicher, dass zu jedem *QualifiedIdentifier* aus den *MExpressions* entweder ein Workflowparameter im Workflowtyp existiert oder dieser *QualifiedIdentifier* durch eine *MExpression* eines *DerivedTypes* im Workflowtyp ersetzt werden kann. Die Abbildung *QualifiedIdentifier* auf Workflowparameter wird dabei in der *SymbolTable* gespeichert, um während der Codegenerierung verwendet werden zu können. Die Ersetzung wird direkt durchgeführt. Falls zu einem *QualifiedIdentifier* keine gültige Abbildung im Workflowtyp existiert, wird dies auch in der *SymbolTable* vermerkt.

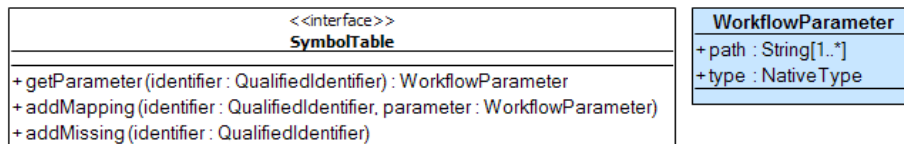


Abbildung 5.19.: SymbolTable Schnittstelle

Sollten für mindestens einen *QualifiedIdentifier* keine Abbildung gefunden werden, ist die Symboltabelle unvollständig. Dies führt zu einem Abbruch der Generierung und alle fehlenden *QualifiedIdentifier* werden dem Benutzer angezeigt (nach Anforderung REQ 12). Ansonsten ist die Symboltabelle vollständig und kann in den folgenden Schritten dazu verwendet werden *QualifiedIdentifier* direkt auf Workflowparameter zu binden.

Aufbau der Symboltabelle

- 1 Suche nach nicht gebundener *MExpression* im Prozessmodell.
- 2 In der *MExpression* wird nun wiederum nach *QualifiedIdentifier* gesucht. Falls kein nicht aufgelöster *QualifiedIdentifier* mehr gefunden wird, wird diese *MExpression* als gebunden markiert werden, weiter mit Punkt 1.
- 3 Es wurde noch ein *QualifiedIdentifier* in der *MExpression* gefunden, falls diese noch nicht in der Symboltabelle eingetragen ist, wird versucht diese im Workflowtyp aufzulösen. Es ergeben sich folgende Fälle:
 - Es wurde nichts gefunden, der *QualifiedIdentifier* wird in der Symboltabelle als fehlend eingetragen. Weiter mit Punkt 2.
 - Die Suche endet nicht in einem Blatt. Auch in diesem Fall wird dieser *QualifiedIdentifier* in der Symboltabelle als fehlend eingetragen. Weiter mit Punkt 2.
 - Es wurde ein Symbol mit einem *DerivedType* gefunden. Dieser Typ bedeutet, dass der *QualifiedIdentifier* in der Expression durch die *MExpression* im *DerivedType* ersetzt werden muss. Diese Ersetzung wird permanent¹⁵ durchgeführt.¹⁶ Weiter mit Punkte 2 und der aktuellen *MExpression*.
 - Es wurde ein *Link* gefunden. Ein Link stellt einen Verweis auf einen anderen Pfad im Workflowtyp dar. Aus dem Ziel des Links wird nun ein neuer *QualifiedIdentifier* erzeugt und wie bei einem *DerivedType* ersetzt diese neue Expression permanent den alten *QualifiedIdentifier*.¹⁷ Weiter mit Punkt 1 und der aktuellen Expression.
 - Es wurde ein Symbol mit einem *PrimitiveType* oder *StringEnumSetType* gefunden. Weiter mit Punkt 2.
- 4 Der gefundene Pfad ist ein Workflowparameter, eine Abbildung des *QualifiedIdentifier* auf diesen Parameter wird nun in der Symboltabelle abgelegt. Weiter mit Punkt 2.

¹⁵Diese Ersetzung bleibt für die folgende Schritte in der Übersetzer-Pipeline sichtbar.

¹⁶Es muss darauf geachtet werden, dass die eingefügte *MExpression* auch gebunden wird. Eine Zyklenkennung ist notwendig, aber im aktuellen Prototyp noch nicht implementiert.

¹⁷Eine Zyklenkennung ist notwendig, aber im aktuellen Prototyp nicht implementiert.

6. Technologien

Machina benutzt diverse Bibliotheken und Frameworks. Die Wichtigsten sind hier aufgeführt. Bis auf Activiti und JIRA sind die anderen Technologien leicht durch andere ähnliche Implementierungen zu ersetzen. Keine hat einen entscheidenden Vorteil gegenüber ähnlichen Implementierungen, es ist lediglich notwendig etwas zu verwenden.

6.1. Activiti

Activiti¹ ist eine Opensource Workflowengine die 2010 von Tom Baeyens und Joram Barrez gestartet wurde [Rad12, vgl. S. 5].

6.1.1. Business Process Modelling and Notation

Als Sprache verwendet Activiti die *Business Process Modelling and Notation* (BPMN) in der Version 2.0, es ist ein Standard und erfüllt somit die Anforderung REQ 1. Es ist eine Sprache zur Beschreibung von Geschäftsprozessen. Eine Besonderheit und damit ein Vorteil, ist die Möglichkeit es zum Modellieren auf der konzeptionellen Ebene für Prozesse sowie auf der operativen Ebene für Workflows einsetzen zu können. Eine vollständige Beschreibung findet sich in der von der OMG² veröffentlichten Spezifikation [OMG11]. Hier werden nur die Teile beschrieben, die auch in Machina Verwendung finden.

Modellierungselemente

In BPMN gibt es vier Hauptelemente, die zur Modellierung eingesetzt werden können, die auch für Machina relevant sind. Eine angenehme Eigenschaft von BPMN ist, dass diese Elemente nicht nur durch ihr Aussehen definiert sind, sondern auch durch ihre XML Repräsentation. Dies erlaubt eine einfache Generierung.

SequenceFlow Sequenzen verbinden die einzelnen Elemente. Durch *SequenceFlow* Elemente die zusätzliche eine *ConditionalExpression* enthalten, kann nach einem Gateway entschieden werden, welche von mehreren möglichen Sequenzen zulässig ist. Graphisch werden Sequenzen als Linien dargestellt. Die XML-Repräsentation ist in Listing 6.1 dargestellt. Für die genaue Semantik sei auf [OMG11, S. 427] verwiesen.

```
<sequenceFlow id="seq1" sourceRef="source" targetRef="target" />
```

¹activiti.org

²Object Management Group - <http://www.omg.org/>

```

<!-- Sequence with Conditional Expression -->
<sequenceFlow id="seq2" sourceRef="source" targetRef="target">
  <conditionExpression xsi:type="tFormalExpression">
    </conditionExpression>
  </sequenceFlow>

```

Listing 6.1: XML-Repräsentation von *SequenceFlow*

Events Mit Events kann ein in BPMN modellierter Workflow auf externe oder interne Ereignisse reagieren oder diese auslösen. Hier sind die Ereignisse *StartEvent* und *EndEvent* relevant (siehe Abbildung 6.1 und Listing 6.2). Diese haben eine leicht unterschiedliche Bedeutung, abhängig von ihrem Kontext. Auf oberster Ebene in einem Workflow sind sie dafür zuständig, den Start bzw. das Ende der Ausführung einer Workflowinstanz zu markieren. Innerhalb eines *SubProcesses* sind die dafür zuständig den Start bzw. das Ende des jeweiligen *SubProcess* zu markieren. Für die genaue Semantik sei auf [OMG11, S. 439 ff.] verwiesen.



Abbildung 6.1.: StartEvent und EndEvent

```

<startEvent id="startId" />
<endEvent id="endId" />

```

Listing 6.2: XML-Repräsentation von StartEvent und EndEvent

Gateways Durch Gateways wird der Aktivitätsfluss in einem Workflow geleitet (siehe Abbildung 6.2 und Listing 6.3). Sie können den Kontrollfluss aufspalten aber auch zusammenführen. Für die genaue Semantik sei auf [OMG11, S. 434 ff.] verwiesen.



Abbildung 6.2.: Die verschiedenen Gateways

```

<exclusiveGateway id="exclusive" />
<parallelGateway id="parallel" />
<inclusiveGateway id="inclusive" />

```

Listing 6.3: XML-Repräsentation der verschiedenen Gateways

Activities Mit Elementen des Typs *Activity* bzw. deren Untertypen werden die tatsächlich auszuführenden Aktivitäten beschrieben. Als Beispiele wurden in Abbildung 6.3 und in Listing 6.4 ein *Task* und ein *ServiceTask* dargestellt. Für die genaue Semantik sei auf [OMG11, S. 426 ff.] verwiesen.



Abbildung 6.3.: Task und ServiceTask

```
<task id="task" name="Name_of_Task" />
<serviceTask id="serviceTask" />
```

Listing 6.4: XML-Repräsentation von Task und ServiceTask

Process und SubProcess Diese speziellen Aktivitäten fassen mehrere Elemente zusammen (siehe Listing 6.5). Sie können selbst wieder alle anderen Modellierungselemente enthalten. Anmerkung: Da BPMN für konzeptionelle und operative Modellierung verwendet werden kann, wird hier den allgemeinere Begriff Prozess verwendet.

```
<process id="process">
  <!-- other process elements -->
  <subprocess id="subprocess">
    <!-- other sub process elements -->
  </subprocess>
</process>
```

Listing 6.5: XML-Repräsentation von Process und Subprocess

6.1.2. BPMN Erweiterungen

Zusätzlich zur Standard BPMN Funktionalität wurden auch Erweiterungen von Activiti benutzt.

DelegateTasks Dieser Task ist ein spezieller *ServiceTask* (siehe Listing 6.6). Anstelle eines Webservices wird hier eine Java Klasse direkt aufgerufen. Dies erleichtert den Aufbau erheblich.

```
<serviceTask id="delegateTask"
  activiti:class="com.example.ClassName">
  <extensionElements>
    <activiti:field name="role">
      <activiti:string>tracker</activiti:string>
    </activiti:field>
```

```

        <activiti:field name="data">
            <activiti:expression>#{juelExpressions} ↗
        </activiti:expression> ↘
        </activiti:field>
    </extensionElements>
</serviceTask>

```

Listing 6.6: XML-Repräsentation eines ServiceTasks mit DelegateTask

Durch spezielle Felder lassen sich beliebige Werte übergeben. Auch ganze Ausdrücke sind möglich und bieten damit Zugriff auf sämtliche Daten im Kontext dieser Workflowinstanz.

Ausdrücke Activiti benutzt für die Ausdrücke den Syntax der „Java Unified Expression Language“ (JUEL). Die Spezifikation für JUEL ist unter [Mic06] zu finden. Diese Ausdrücke können unter anderem in *ConditionalSequences* und in den *DelegateTasks* verwendet werden.

6.2. HornetQ - JMS

HornetQ³ ist eine Implementierung der Java Messaging Spezifikation [Mic02] von JBoss.

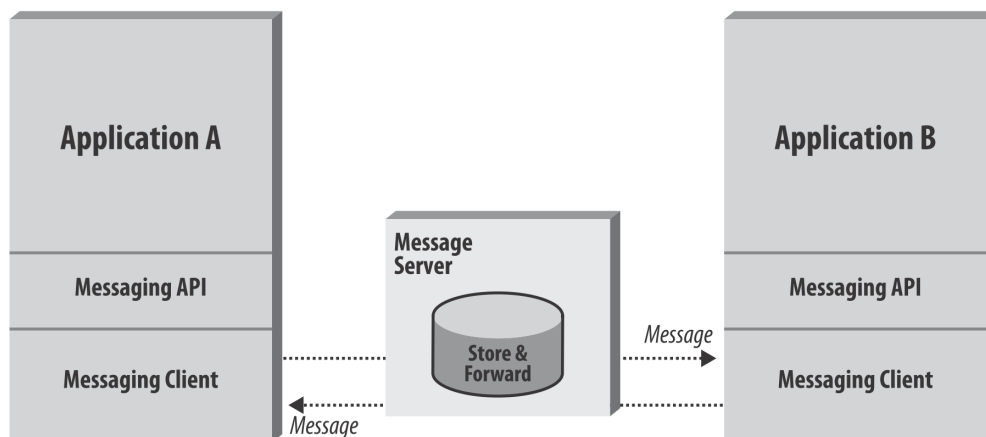


Abbildung 6.4.: „Underlying store-and-forward mechanism guarantee delivery of messages“ [RMHC09, S. 24]

Durch den zugrundeliegenden Store-and-forward Mechanismus (Abbildung 6.4) ist es garantiert, dass Nachrichten nicht verloren gehen. Hierbei wird eine Nachricht erst angenommen, wenn sie sich im persistenten Speicher befindet. Die Nachricht wird erst aus dem Speicher gelöscht, wenn sie mit Sicherheit zugestellt wurde. Genau dieses Verhalten benötigen wir für die Implementierung des Nachrichtenbusses. Darüber hinaus bietet

³<http://www.jboss.org/hornetq>

JMS noch viel mehr an, dass nicht direkt benötigt wird. Da es aber ein Industriestandard ist, lassen sich die Implementierungen beliebig austauschen, falls sich HornetQ als Problem herausstellen sollte.

6.3. ANTLR

ANTLR⁴ ist ein Parsergenerator, der von Terence Parr seit 1993 beständig weiter entwickelt wird [Par07, vgl. S.14 ff.]. Für eine vollständige Einführung in dieses umfangreiche Werkzeug sei hier auf das Buch [Par07] verwiesen. Ein einfaches Tutorial für die in dieser Arbeit verwendeten Teile ist unter [Ros07] verfügbar. Wie andere Parsergeneratoren auch, gibt man diesem Werkzeug eine Grammatik und bekommt damit einen fertigen Parser. Dem Parser gibt man einen Quelltext und erhält diesen in einer generischen Syntaxbaumrepräsentation.

6.4. JIRA

JIRA ist ein Issue-Tracking-System der Firma Atlassian. Es lässt sich durch eine Vielzahl von Plugins erweitern. Eine für diese Arbeit sehr hilfreiche Eigenschaft ist die Existenz einer sehr gut dokumentierten Api. Mit dieser Api kann man JIRA nach seinen Wünschen anpassen. Für einen detaillierten Einstieg in dieses Thema sei auf die Bücher [Li11] und [Kur11] verwiesen.

6.5. Stages

Stages⁵ ist ein Prozessmodellierungswerkzeug der Firma Methodpark Software AG. Es war eine Anforderung dieses Format zu unterstützen (siehe Anforderung REQ 9). Die Besonderheiten dieses Formats werden in Abschnitt 7.7.1 genauer behandelt.

6.6. EclipseLink - JPA

EclipseLink ist die Referenzimplementierung der Java Persistence Api (JPA) Spezifikation [Mic09]. JPA ist ein Industriestandard für Object Relational Mapper (ORM). Diese erlauben es Objekte und Objektgraphen in einer relationalen Datenbank abzulegen. Es wird für die Anbindung an eine Datenbank benutzt, falls Daten persistent gespeichert werden müssen.

⁴steht für „ANother Tool for Language Recognition“ und ist unter <http://www.antlr.org/> verfügbar

⁵<http://stages.methodpark.de/>

6.7. Spring

Spring⁶ ist ein Framework das die Prinzipien des *Dependency Injection* bzw. *Inversion of Control* umsetzt. Diese Prinzipien werden in [Fow04] beschrieben. Für eine detaillierte Beschreibung sei hier auf die umfangreiche Dokumentation verwiesen [Sou10].

Durch den Einsatz von *Dependency Injection* ist es leicht möglich Klassen in Isolation zu testen. Für den produktiven Betrieb werden diese dann durch eine auf XML basierende Konfiguration zusammengestellt. Diese Konfiguration erlaubt es komfortabel die Architektur des Systems zu steuern. Zusätzlich bietet Spring einige Erweiterungen an, mit denen man JPA und JMS mittels Springkonfiguration konfigurieren kann.

⁶<http://www.springsource.org/>

7. Implementierung

Einige der Komponenten sind durch das Design so weit ausformuliert, dass deren Implementierungen ein reines „ausprogrammieren“ darstellt. Hier werden die Implementierungsdetails der Komponenten besprochen die darüber hinausgehen.

7.1. Konfiguration

Die Konfigurationen werden in diesem Prototyp durch eine statische *HashMap* implementiert. Diese *HashMap* wird beim Starten von Machina aus einer Datei mit den Konfigurationsdaten befüllt. Da diese Datei im Klartext im Dateisystem liegt, ist ihr einziger Schutz die Beschränkungen des Dateisystems. Diese Implementierung erfüllt damit nicht die Anforderung von Sicherheit REQ 4. In einer produktiven Version ist es unerlässlich die Konfigurationen durch zusätzliche Sicherheitsmaßnahmen abzusichern.

7.2. Trigger

Die *Trigger* müssen von Machina zuverlässig dauerhaft gespeichert werden. Deswegen ist die naheliegende Implementierung, diese in einer Datenbank zu abulegen. Es wurde JPA¹ verwendet um diese Teile der *Triggers*-Komponente zu implementieren.

7.3. Jira-Modul

Die Implementierung eines Werkzeugmoduls wird hier exemplarisch am Jira-Modul ausgeführt. Ein angebundenes Werkzeug besteht zunächst aus zwei Teilen: Der entfernt laufenden Werkzeuginstanz sowie dem Werkzeugmodul in Machina. Die Werkzeuginstanz muss möglicherweise um ein Plugin erweitert werden, damit es die benötigten Informationen an Machina liefern kann. Das Werkzeugmodul ist ein Plugin für das die machinaspezifische Daten in werkzeugspezifischen Daten überführt. Hierfür muss das Modul einige der bereits angesprochenen Schnittstellen implementieren. Abbildung 7.1 zeigt den Aufbau des Jira-Moduls.

MessageBus Durch verwenden dieser Schnittstelle ist es dem Jira-Modul möglich Eventnachrichten an den Event-Prozessor zu schicken und Aktionsnachrichten zu erhalten.

¹Java Persistence Api: Details im Abschnitt 6.6

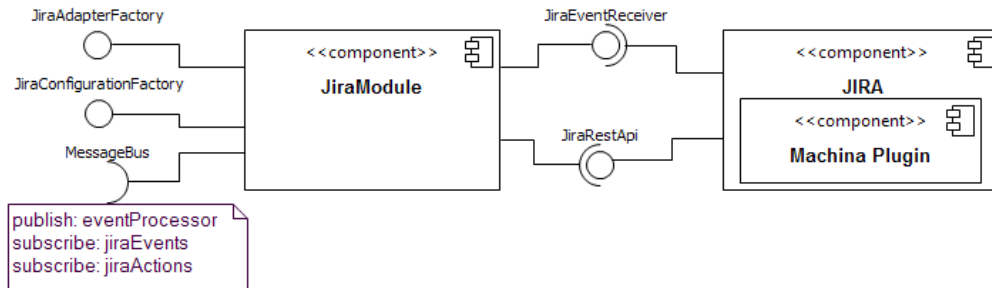


Abbildung 7.1.: Komponentendiagramm des Jira-Moduls

ConfigurationFactory und AdapterFactory Diese beiden Schnittstellen müssen realisiert werden, um für die *Integration*-Komponenten die Konfigurationen und Adapter zur Verfügung stellen zu können.

JIRA

Im Zusammenspiel mit JIRA und dem darin enthaltenen Plugin müssen folgende Schnittstellen vorhanden sein.

JiraEventReceiver Über diese Schnittstelle werden die JIRA internen Events an das Jira-Modul übermittelt. Sie ist auch durch den Nachrichtenbus realisiert.

JiraRestApi Über diese Schnittstelle werden die Effekte in JIRA ausgelöst. Wie der Name schon andeutet wird sie durch die von JIRA angebotene RESTful Api realisiert. Während des Bindevorgangs werden über diese Schnittstelle die verfügbaren IssueTypes und deren Attribute ausgelesen.

7.3.1. Plugin

Um die Events von JIRA zu erhalten muss man ein Plugin für JIRA erzeugen. In diesem Plugin kann man den Event `com.atlassian.jira.event.issue.IssueEvent` abonnieren und verarbeiten. Mit diesem Event erhält man Zugriff auf alle neu angelegten und veränderten Issues. Dieser Event wird zunächst serialisiert und dann lokal persistiert². Anschließend wird der Event an den Nachrichtenbus übergeben und dieser Event als gesendet markiert. Das Schema in Tabelle 7.1 beschreibt dieses Verhalten noch einmal in der Übersicht.

Dieses Schema kann nicht vollständig verhindern, dass Events verloren gehen. Es gibt eine Zeitspanne, in der der Issue schon verändert wurde, aber die Eventnachricht noch nicht in der lokalen Warteschlange eingereicht ist. Deswegen muss beim Starten von JIRA überprüft werden, ob es Issues gibt, zu denen noch keine Eventnachricht erzeugt wurde. Um diesen Test durchführen zu können kann die Eventnachricht im Schritt Vier nur als gesendet markiert werden, sie kann jedoch nicht vollständig gelöscht werden³.

²Lokale Warteschlange für den Fall der Nichtverfügbarkeit des Nachrichtenbusses.

³Dieser Startup Test wurde im Prototyp nicht implementiert.

1	Empfang eines <i>IssueEvents</i> von JIRA <i>I</i>
2	Zwischenspeichern des Events <i>I</i>
3	Übergeben des Events <i>I</i> an den Nachrichtenbus
4	Bestätigen des Events <i>I</i> indem im Zwischenspeicher als versendet markiert wird.

Tabelle 7.1.: Schema zum versenden der *IssueEvents* in JIRA

7.3.2. Eventnachrichtenverarbeitung

Für eingehende Eventnachrichten entsteht der Kommunikationsfluss, der durch Abbildung 7.2 beschrieben ist. Diese Transformation des Events ist idempotent, es ist also nicht notwendig etwaige Ergebnisse zu speichern.

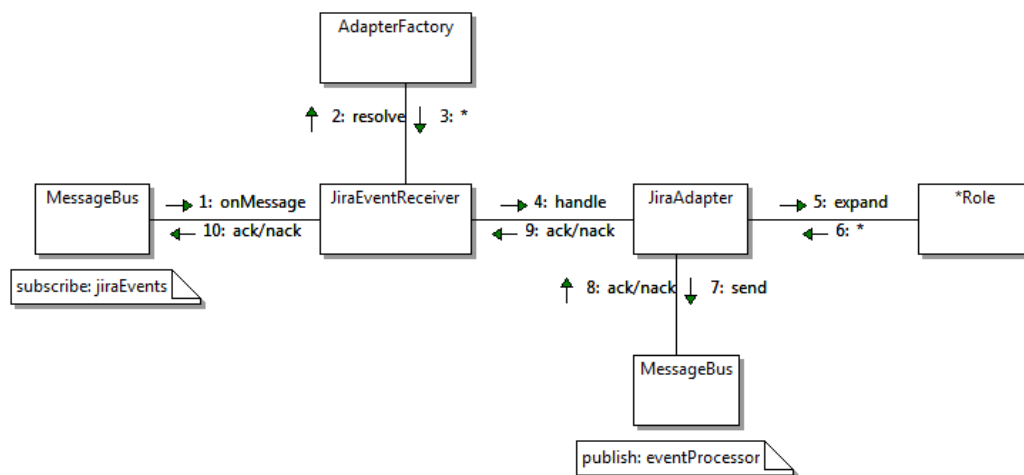


Abbildung 7.2.: Kommunikationsdiagramm für die Verarbeitung der Eventnachrichten

- 1 Empfangen einer Nachricht. Diese Nachricht wird im *JiraEventReceiver* de-serialisiert, und die Konfigurationsdaten werden extrahiert.
- 2 Laden eines konfigurierten *JiraAdapter* mithilfe der *AdapterFactory*.
- 3 Es ergeben sich drei Fälle:
 1. Es wurde ein passender Adapter gefunden: weiter mit Punkt 4.
 2. Es wurde kein Adapter gefunden: bestätige die Nachricht mit einem 'ack', was dem Ignorieren dieses Events entspricht.
 3. Es wurde ein Adapter eines falschen Moduls gefunden: diese Nachricht wird als fehlerhaft markiert, wird aber dennoch mit einem 'ack' bestätigt. (Diese Nachricht stellt eine vergiftete Nachricht dar; weitere Versuche des Ausführens

sind zwecklos und würden zur Starvation/Verhungern anderer Nachrichten führen.)

4 Übergeben des *JiraEvents* an den *JiraAdaper*.

5, 6 Übergeben des *JiraEvents* an die vom *JiraAdapter* implementierten Rollen. Diese Rollen erzeugen aus dem jiraspezifischen *JiraEvent* generische *RoleEvents*. Der *JiraAdapter* aggregiert die einzelnen *RoleEvents* in einem allgemeinen *Event* (siehe Abbildung 5.2).

7 Dieser *Event* wird an den Nachrichtenbus übergeben und dort unter dem Thema „eventProcessor“ veröffentlicht.

8,9,10 Es ergeben sich zwei Fälle:

1. Der Nachrichtenbus meldet das erfolgreiche versenden des Event. Es werden 'acks' zurückgeben und der *JiraEvent* als verarbeitet bestätigt.
2. Der Nachrichtenbus kann den Event gerade nicht versenden. Die aktuelle Ausführung wird abgebrochen. Der Nachrichtenbus wird mittels 'nack' angewiesen die ursprüngliche Eventnachricht erneut auszustellen.

7.3.3. Aktionsnachrichtenverarbeitung

Für die tatsächliche Verarbeitung des Events sei hier auf die Implementierung der Engine verwiesen (Abschnitt 7.5). Das Jira-Modul erhält nachdem der *Event* verarbeitet wurde Aktionsnachrichten mit der es Effekte in der zugeordneten JIRA-Instanz auslöst. Abbildung 7.3 beschreibt diesen Schritt.

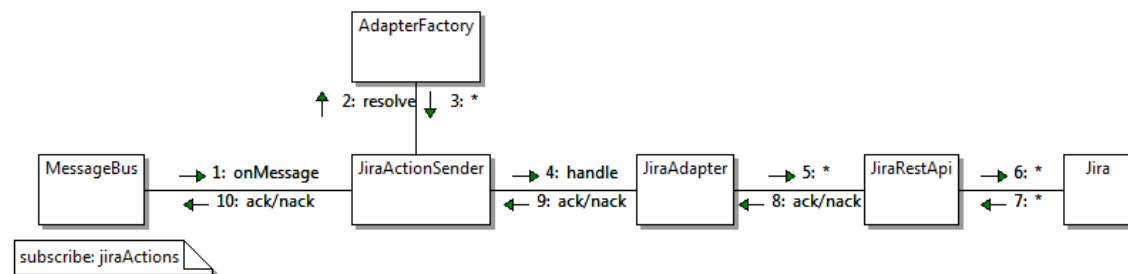


Abbildung 7.3.: Kommunikationsdiagramm für die Verarbeitung der Aktionsnachrichten

- 1 Empfangen einer Nachricht. Diese wird vom *JiraEventSender* de-serialisiert, und die Konfigurationsdaten werden extrahiert.
- 2 Laden des konfigurierten *JiraAdapter* mithilfe der *AdapterFactory*.
- 3 Es ergeben sich drei Fälle:

1. Es wurde ein passender Adapter gefunden: weiter mit Punkt 4.
 2. Es wurde kein Adapter gefunden: Bestätigen der Nachricht mit einem 'ack'. Was dem ignorieren dieser Nachricht entspricht.
 3. Es wurde ein Adapter eines falschen Moduls gefunden: diese Nachricht wird als fehlerhaft markiert, wird aber dennoch mit einem 'ack' bestätigt.
- 4** Übergeben der auszuführenden Aktion an den *JiraAdapter*. Der *JiraAdapter* dient hier als Synchronisationspunkt. Ein *JiraAdapter* darf aufgrund von möglichen Lese-Schreib-Wettlaufsituationen nicht nebenläufig mehrere⁴ Aktionen ausführen.
- 5,6,7** Der *JiraRestClient* muss nun dafür Sorgen, dass die Aktion nur einmal ausgeführt wird. Hierzu müssen aktionspezifische Strategien angewandt werden.
- 8,9,10** Es ergeben sich zwei Fälle:
1. Der *JiraRestClient* meldet das erfolgreiche Ausführen der Aktion. Es werden 'acks' zurückgeben.
 2. Der *JiraRestClient* hat einen möglicherweise nicht permanenten Fehler festgestellt. Die aktuelle Ausführung wird abgebrochen. Der Nachrichtenbus wird mittels 'nack' angewiesen die ursprüngliche Aktionsnachricht erneut zuzustellen.

Derzeit unterstützt der *JiraAdapter*, bedingt durch die implementierte *TrackerRole*, zwei Aktionen: das Anlegen und das Schließen von Issues.

IssueCloseAction Diese Aktion ist auch noch auf JIRA Seite idempotent. Man kann sie so lange wiederholen bis sie erfolgreich war, ohne negative Konsequenzen befürchten zu müssen.

IssueCreateAction Diese Aktion ist nur testbar. Hier muss also überprüft werden, ob dieser Issue schon erzeugt wurde und darf erst nachdem eindeutig ist, dass dieser Issue noch nicht existiert einen Versuch starten diesen anzulegen. Hier muss auf die Lese-Schreib-Wettlaufsituation besonders geachtet werden. Für eine Konfiguration darf global nur ein solcher Befehl ausgeführt werden, dies wird durch einfache Synchronisation⁵ auf der *JiraAdapter* Instanz erreicht. Allerdings könnte durch Netzwerkverzögerung auch diese Synchronisation nicht ausreichend sein. Hier muss eine absichtliche Verzögerung in die Ausführung eingefügt werden, um auch Netzwerkverzögerungen tolerieren zu können.

7.4. Nachrichtenbus

Der Nachrichtenbus wurde mit der JMS Implementierung HornetQ⁶ realisiert. Es wird aber nicht direkt HornetQ genutzt sondern nur die JMS Erweiterungen von Spring. Damit

⁴Es könnte ein Duplikat der gleichen Aktion sein.

⁵Mittels des Java Keyword synchronized.

⁶Genauer in Abschnitt 6.2

ist es möglich durch Änderungen an der Spring Konfiguration die JMS Implementierung zu ändern. Da sich schon bei den Grundlagen über Integration für textuelle Austauschformate beschränkt wurde, ist die Einschränkung von JMS, nur primitive Datentypen und serialisierbare Objekte übertragen zu können, irrelevant. Das Verhalten von HornetQ lässt sich durch umfangreiche Konfigurationsoptionen einstellen. Das Bestätigen von Nachrichten wurde auf explizit gesetzt, hierdurch wird im Zweifelsfall immer eine erneute Auslieferung der Nachricht durchgeführt.

Umsetzung der verschiedenen Operationen:

publish Das Veröffentlichen einer Nachricht findet über die *org.springframework.jms.JmsTemplate* Schnittstelle statt. Durch einen Eintrag in der Springkonfiguration wird diese mit einem Thema verbunden.

subscribe Ein *subscribe* wurde zusammen mit den JMS Erweiterungen von Spring über eine Implementierung der *org.springframework.jms.listener.SessionAwareMessageListener* Schnittstelle erreicht. Diese Implementierung wird durch einen Eintrag in der Springkonfiguration mit einem Thema verbunden.

ack Ein expliziter *ack* wird durch einen Methodenaufruf von *javax.jms.Message.acknowledge()* erreicht. Da HornetQ so konfiguriert ist, dass alle Empfangsbestätigungen explizit sein müssen, gibt es keine implizite Variante.

nack Ein *nack* kann entweder implizit durch nicht Akzeptieren der Nachricht geschehen oder durch das Werfen einer *JmsException*.

7.5. EventControl

Das *EventControl* ist eine Kollaboration von zwei Hauptklassen und einigen Hilfsklassen. Die Klasse *EventMachine* realisiert einen Zustandsautomaten. Dabei benutzt sie die Klasse *EventLogRepository* um ihren Fortschritt persistent zu protokollieren. Mit dem *EventLogRepository* wird immer der zuletzt erreichte Zustand des Events gespeichert. Somit kann der Zustandsautomat selbst im Falle eines Ausfalls an der entsprechenden Stelle fortgesetzt werden.

7.5.1. EventLogRepository

Eine Implementierung der *EventLogRepository* Schnittstelle (in Abbildung 7.4) stellt die Möglichkeit für die *EventMachine* bereit, den letzten erreichten Zustand eines Events zu speichern und auszulesen. Hierdurch vereinfacht sich die Fehlersuche beim Starten nach einem Ausfall. Da im Falle eines Ausfalls lediglich die ausgehende Transition vom letzten bekannten Zustand nach Artefakten von vorherigen Ausführungen durchsucht werden müssen. Anschließend kann die Ausführung fortgesetzt werden. Jeder Methodenaufruf löst eine Datenbanktransaktion aus. Damit ist garantiert dass der Zustand auch über Ausfälle hinweg erhalten bleibt. Diese Klasse wird mittels JPA⁷ implementiert.

⁷Java Persistence Api: Details im Abschnitt 6.6

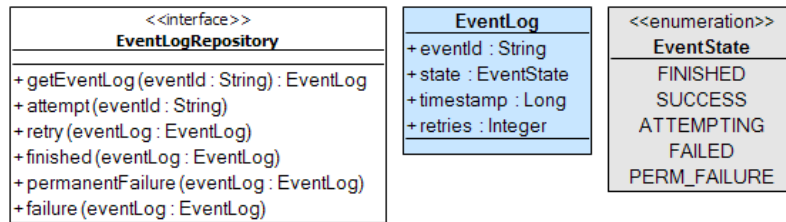


Abbildung 7.4.: EventLogRepository Schnittstelle

7.5.2. Zustandsautomat

Der Zustandsautomat in Abbildung 7.5 wird durch die Klasse *EventMachine* implementiert. Er ist dafür zuständig die einzelnen Events abzuarbeiten. Die Herausforderung dabei ist sicherzustellen, dass zum Einen duplizierte Events toleriert werden können und zum Anderen Relikte, die durch Ausfälle entstanden sind, sicher zu erkennen und zu korrigieren.

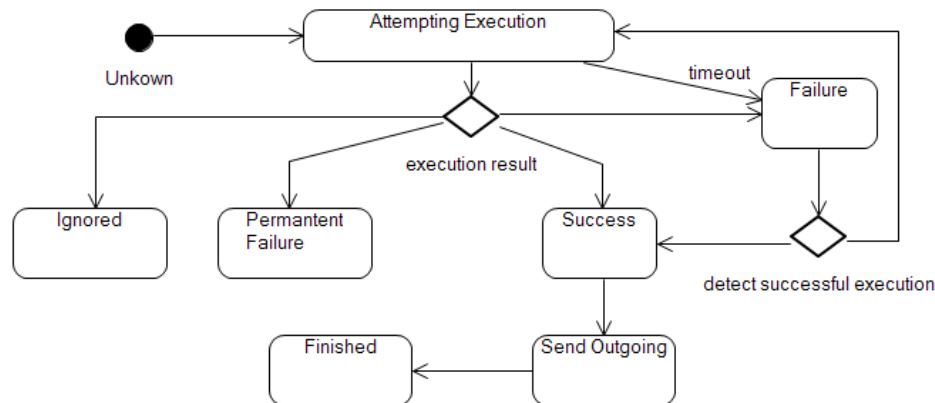


Abbildung 7.5.: Zustandsautomat in der Klasse *EventMachine*

Unkown Markiert einen Event, der bisher unbekannt ist.

Attempting Execution Der Event wird gerade von der Engine ausgeführt. Soll die *EventMachine* einen Event in diesem Zustand erneut ausführen, wird angenommen, dass dieser bereits von einem anderen Thread ausgeführt wird. Die aktuelle Ausführung wird abgebrochen und der Event später erneut zugestellt. Im Falle eines Ausfalls würde das jedoch zur Starvation/Verhungern dieses Events führen. Deswegen wird zusätzlich noch ein Timeout⁸ beachtet. Nach einer gewissen Zeit im Zustand „at-

⁸Gespeichert im Feld *timestamp* des *EventLog*

tempting“ wird automatisch auf *Failure* übergegangen, danach kann eine erneute Ausführung versucht werden.

Failure Die vorausgehende Eventausführung wurde mit einem Fehler abgebrochen. Im Falle eines *StartCommands* wird überprüft, ob bereits eine erfolgreich gestartete Workflowinstanz existiert. Falls diese existiert, wird der Event in den Zustand „Success“ überführt. Sonst wird die Ausführung erneut gestartet und der Event geht in den Zustand „Attempting Execution“ über.

Success Die Ausführung war erfolgreich. Es wurden aber noch keine Ergebnisse externalisiert.

Send Messages Sendet die zwischengespeicherten Aktionen. Damit werden die Ergebnisse der Ausführung externalisiert.

Finished Ein Endzustand, der das erfolgreiche Ausführen des Events markiert.

Ignore Ein Endzustand, der einen nicht relevanten Event markiert.

Permanent Failure Ein Endzustand, der einen nicht ausführbaren Event markiert. Beispiele hierfür sind: zu viele Ausführungsversuche oder Trigger mit falschen Daten.

Sobald ein Event einen Endzustand erreicht hat, werden Wiederholungen dieses Events quitiert, es findet aber keinerlei Aktion mehr statt. Wie man sieht ist hier das Nachrichtenverarbeitungsschema eingehalten. Die Ausführung ist komplett abgeschlossen bevor Ergebnisse sichtbar werden.

7.6. Engine

Das *EventControl* benötigt eine Implementierung von *Engine* um ihre Aufgabe erfüllen zu können. Als Kern der Engine wurde die Prozessengine von Activiti⁹ verwendet. Die *Engine* Schnittstelle wird als Adapter [GHJV94, S. 139 ff.] implementiert und delegiert die Aufrufe an Activiti. Neben der Engine Schnittstelle, die sich um die *TriggerCommand*-Verarbeitung kümmert, sind noch einige andere Klassen zu implementieren, die unter *JavaDelegates* zusammengefasst sind, die sich um die Erstellung der Aktionen kümmern.

7.6.1. CommandExecutor

Durch die Schnittstelle *CommandExecutor* ist für jeden *TriggerCommand* eine Methode zu implementieren. Das Ausführen eines jeden *TriggerCommands* hat einen Aktivitätsschub in der Workflowinstanz zur Folge. Während dieses Schubs werden *JavaDelegates* angesprochen, die wiederum Aktionsnachrichten im *MessageRepository* speichern.

⁹Details im Abschnitt 6.1

execute(*StartProcessCommand*) Dieser *TriggerCommand* ist dafür zuständig einen Workflow zu finden und damit von Activiti eine neue Workflowinstanz erzeugen zu lassen. Er wurde während des Deployment Vorgangs in der Übersetzer-Pipeline von Activiti erzeugt. Im Property *id* wurde der interne Workflow Identifier von Activiti hinterlegt. Zum Ausführen des *StartProcessCommands* sind folgende Schritte notwendig:

1. Suchen des Workflows in Activiti mit der hinterlegten *id*. Falls kein Workflow mit dieser *id* gefunden wurde, wird dies durch eine *Exception* als permanenter Fehler an die *EventMachine* signalisiert.
2. Erzeugen eines Workflowkontext, indem die alle verwendeten Variablen mit *null* vorbelegt werden.
3. Eintragen der Workflowargumente dieses Events in den erzeugten Workflowkontext.
4. Starten einer Workflowinstanz des gefundenen Workflows mit dem erzeugten Workflowkontext.

execute(*ContinueCommand*) Dieser *TriggerCommand* wurde während des Abarbeiten eines vorhergehenden Events im ersten Teil einer blockierenden Aktion erzeugt. In ihm wurden Activiti spezifische Daten zum Auflösen dieser Blockierung hinterlegt. Es wurden die notwendigen Information gespeichert um einen *ActivitiReceiveTask* zu finden. Der Event, der diesen Command auslöst kann *EventArguments* besitzen, diese müssen in den Workflowkontext als *WorkflowArgumente* eingetragen werden. Dies geschieht in folgenden Schritten:

1. Erweitern der *EventArguments* um den Namensraum¹⁰ dieses *TriggerCommands*.
2. Suchen des *ReceiveTasks* aus der Workflowinstanz der fortgesetzt werden soll. Als Resultat gibt es drei Möglichkeiten:
 - a) Der Task wurde gefunden und kann ausgelöst werden: Auslösen.
 - b) Der Task wurde gefunden, ist aber nicht mehr auslösbar: Dieser Task wurde durch eine vorhergehende Ausführung eines Events bereits ausgelöst. Rückkehr als wäre der Command normal ausgelöst worden.
 - c) Der Task existiert nicht: per Exception permanenten Fehler an die *EventMachine* melden.

7.6.2. MessageCache - MessageRepository

Da das Ausführen in der Engine nur testbar ist und ein wiederholtes Ausführen nicht möglich ist oder zu Inkonsistenzen führen würde, müssen alle erzeugten Aktionsnachrichten zwischengespeichert werden. Für dieses Zwischenspeichern ist eine Implementierung der *MessageRepository* Schnittstelle (siehe Abbildung 7.6) zuständig, die mittels JPA¹¹ die

¹⁰Dieser Namensraum wurde im *ContinueCommands* gespeichert

¹¹Java Persistence Api: Details im Abschnitt 6.6

Nachrichten in einer Datenbank ablegt. Durch den Einsatz einer Datenbank sind selbst nach einem Ausfall die Aktionsnachrichten noch vorhanden. Nach dem erfolgreichen Ausführen des *TriggerCommands* stehen die Aktionsnachrichten über die Methode *retrieveMessages(String)* für die *EventMachine* zur Verfügung.

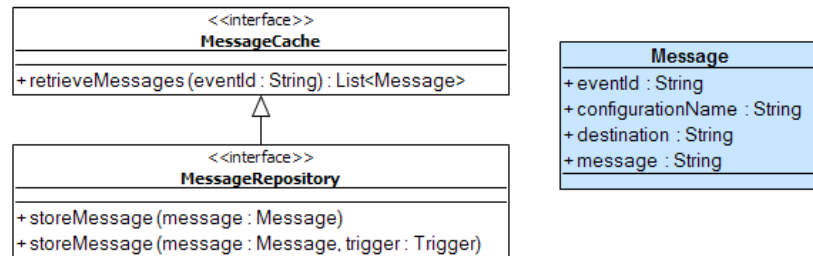


Abbildung 7.6.: Zwischenspeicher für Aktionsnachrichten

Eine Aktionsnachricht besteht aus der *eventId* des Events aus dem sie erzeugt wurde, dem Namen der Konfiguration der die Nachricht ausführen soll, einem Nachrichtenbus-Thema unter dem ein Werkzeugmodul auf die Aktionsnachricht wartet sowie der serialisierten eigentlichen Aktion. Eine Verbindung zur Rolle ist nicht notwendig.

7.6.3. JavaDelegates

Activiti ermöglicht es an der Stelle von *ServiceTasks* auch Java Klassen direkt aufzurufen. Das wird verwendet um die Aktionsnachrichten zu erzeugen. Abhängig vom Typ auszuführenden Aktion¹² wurde während der Übersetzer-Pipeline der richtige Delegate gewählt, und in das erzeugten BPMN generiert. Während der Codegenerierung werden auch die *AdapterOptionen* aus dem *ME enactmentBinding* gespeichert.

BlockingTaskDelegate

Die Verarbeitung einer blockierenden Aktion wird in Abbildung 7.7 beschrieben, es entsteht dabei eine Aktionsnachricht. Es wird zunächst eine *BoundRole* erzeugt. Anschließend lässt man von dieser Rolle die entsprechende Aktionsnachricht erzeugen. Zusätzlich zur Aktion wird auch ein *Trigger* mit *ContinueCommand* erzeugt, der die Informationen enthält, wie diese blockierende Aktion fortgesetzt wird. Anschließend wird die Nachricht zusammen mit dem *Trigger* im *MessageRepository* zwischengespeichert, bis alle Aktionen des aktuellen Aktivitätsschubs ausgeführt wurden.

NonBlockingTaskDelegate

Die Nachrichten für nicht-blockierende Aktionen werden wie die für blockierende Aktionen erzeugt. Der Unterschied ist, dass diese keinen *Trigger* benötigen.

¹²Blockierend oder Nicht-blockierend

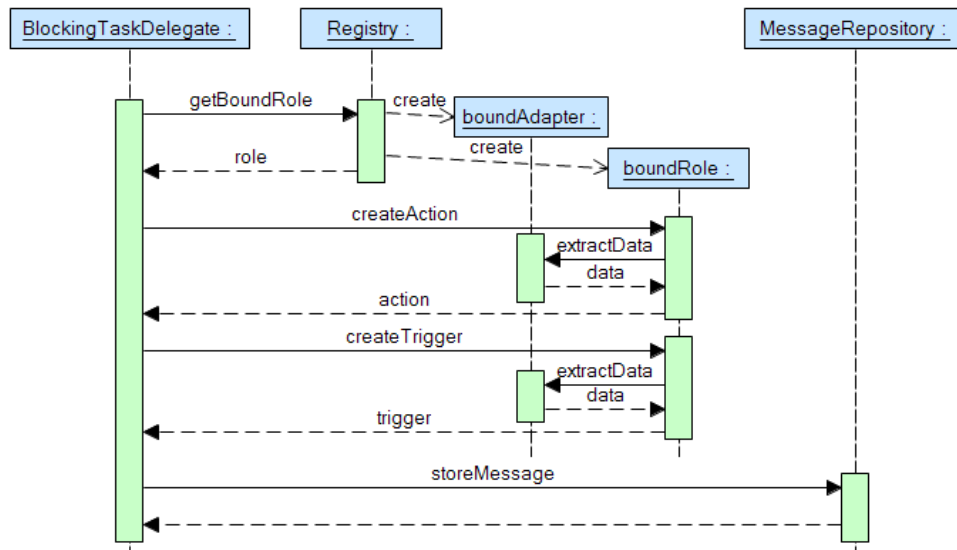


Abbildung 7.7.: Erzeugen der Aktionsnachricht in einem BlockingTaskDelegate

7.7. Übersetzer

7.7.1. StagesModelReader

Diese Komponente wurde durch die Anforderung REQ 9 notwendig. Sie wandelt ein Prozessmodell im Stages-Format in eine Instanz des Machina-Prozessmodells für die Übersetzer-Pipeline um.

Stages-Format

Das Stages-Format besteht im wesentlichen aus einer Instanz des Modells in Abbildung 7.8. Es besteht aus einer Liste von *ProcessElement* die durch *Association* verbunden sind. Zur verbesserten Übersicht wurden hier nur die Typen *Activity* und *Resource* erwähnt, da die anderen Typen nicht für eine mögliche Ausführbarkeit relevant sind. In Abbildung 7.9 ist das Szenario aus Abschnitt 2.2 in der graphischen Version von Stages abgebildet.

Semantik

Das Stages-Prozessmodell war bisher nicht für eine Ausführbarkeit ausgelegt, deswegen besitzen diese Typen nicht genug formal festgelegte Bedeutung um dies zu ermöglichen. Es musste also eine Möglichkeit geschaffen werden eine Semantik für die einzelnen Stages-Prozesselemente zu definieren. Es ist ebenso notwendig¹³, dass diese Semantik vom einzelnen Projekt umdefiniert werden kann. Erreicht wurde diese indem die Instantiierungslogik als Semantikbeschreibung dient. Um die Instantiierungslogik zu implementieren wurde

¹³Wegen Anforderung REQ 10

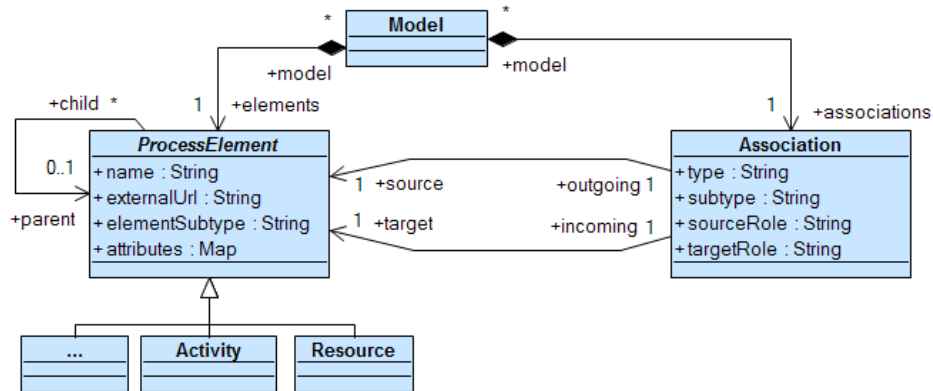


Abbildung 7.8.: Gekürztes Stages-Prozessmodell

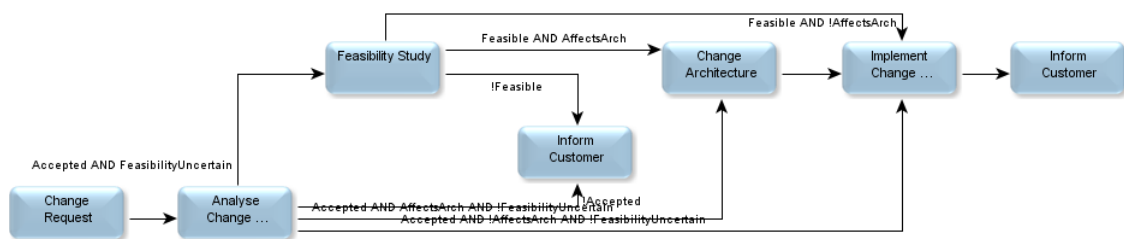


Abbildung 7.9.: Graphische Repräsentation des Szenarios

JavaScript¹⁴ verwendet, es ist Turing-vollständig und kann somit sämtliche Transformationen beschreiben.

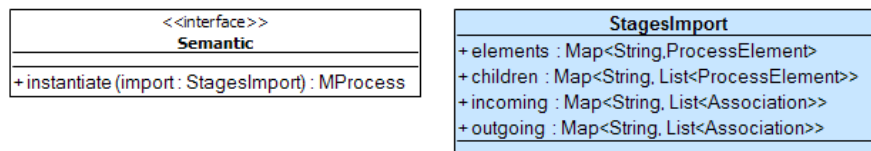


Abbildung 7.10.: Semantik Schnittstelle

Eine Semantik muss die Schnittstelle in Abbildung 7.10 implementieren. Über diese Schnittstelle wird ein Stages-Prozess¹⁵ übergeben. Dieser Prozess wird nun von der Semantikimplementierung in eine Instanz des Machina-Prozessmodells umgewandelt. Der entscheidende Vorteil von JavaScript¹⁶ ist, dass man es direkt in den Stages-Prozess als

¹⁴Diese Sprache war naheliegend, da das Rollout-Team diese schon verwendet.

¹⁵Als eine Instanz von *StagesImport*

¹⁶Diesen Vorteil teilt es mit allen Skriptsprachen, die in Java einbettbar sind.

Resource hinterlegen kann. Damit kann für jeden Prozess eine eigene Semantik definiert werden, was die Anforderung REQ 10 erfüllt. Man stellt also dem Stages-Prozess die Möglichkeit zur Verfügung sich selbst nach Machina zu importieren.

Erweiterbare Standardsemantik für das Stages-Prozessmodell

Es ist zu erwarten, dass sich die meisten Semantiken lediglich in Kleinigkeiten unterscheiden, ist es unnötig zu jedem Projekt eine eigenständige Semantik zu implementieren. Das würde den Wartungsaufwand nur unnötig erhöhen. Deswegen wurde eine Standardsemantik erstellt. Diese Semantik ist an einigen Stellen durch eigene Klassen erweiterbar. Das führt im Idealfall dazu, dass nur wenige Methoden in JavaScript überschrieben werden müssen. Diese Semantik soll nicht alle Möglichkeiten abbilden können, sie soll lediglich kleinere prozessspezifische Abweichungen erlauben, ohne eine komplette Semantik zu implementieren und pflegen zu müssen. Die vollständige Struktur wird im Anhang unter A.3 abgebildet.

Hier die vier Hauptklassen die man projektspezifisch erweitern kann (siehe Abbildungen 7.11 und 7.12). Je zwei Klassen sind für Assoziationen und Prozesselemente zuständig. Die **Selector* Klasse wählt eine Fabrikmethode aus, die *Factory** Klasse implementiert dann die Fabrikmethoden.

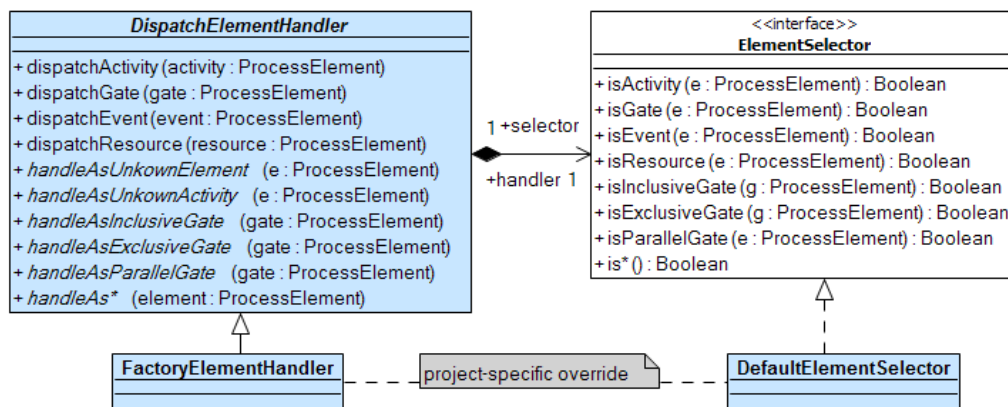


Abbildung 7.11.: Teil der Standardsemantik zum Ändern der Prozesselementerzeugung

DefaultElementSelector Diese Klasse implementiert die Entscheidungsregeln für Prozesselemente, welche Fabrikmethode aufgerufen wird.

FactoryElementHandler Diese Klasse implementiert die Fabrikmethoden, die die Elemente erzeugt.

DefaultAssociationSelector Diese Klasse implementiert die Entscheidungsregeln für Assoziationen, welche Fabrikmethode aufgerufen wird.

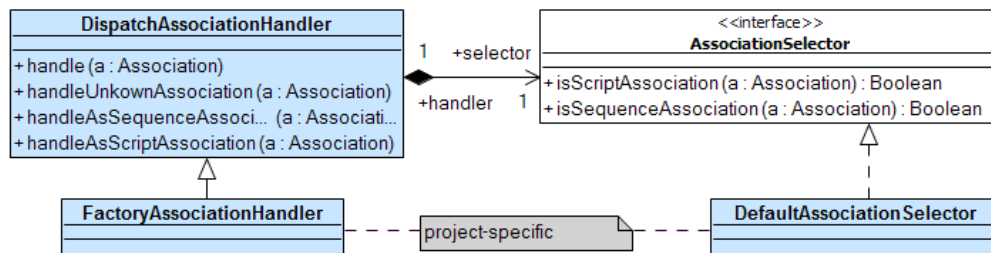


Abbildung 7.12.: Teil der Standardsemantik zum Ändern der Assoziationserzeugung

FactoryAssociationHandler Diese Klasse implementiert die Fabrikmethoden die Assoziationen erzeugt.

Mit diesen vier Klassen hat man somit Zugriff auf den Elemententstehungsprozess und kann diesen erheblich beeinflussen. Sollte es nötig sein mehr als den Elemententstehungsprozess zu kontrollieren, kann man beliebige andere Teile der Defaultsemantik durch eigene Implementierung¹⁷ überschreiben. Ein Beispiel hierfür ist im Anhang unter A.1 zu finden.

7.7.2. Transformatoren

Die einzelnen Transformatoren werden durch eine Spring Konfiguration in Reihe geschaltet.

7.7.3. Expressions-Parser

Die Ausdrücke in den *MExpressions* sind zunächst nur textuell vorhanden. Sie müssen erst in eine Instanz des Machina-Expression-Modells¹⁸ umgewandelt werden. Hierfür ist ein Parser notwendig. Dieser Parser wurde mit Werkzeug ANTLR erzeugt. Als Basis für die Grammatik wurde der Artikel [Ros07] verwendet. Diese in diesem Artikel beschriebene Grammatik musste noch um *QualifiedIdentifier* erweitert werden.

Für die Entscheidung eine eigene Grammatik¹⁹ zu benutzen gab es einige Gründe. Zunächst werden lediglich arithmetisch und logische Ausdrücke benötigt, diese sind so grundlegend, dass diese den meisten schon bekannt sein sollten. Aber auch das Einführen projektspezifischer Erweiterungen²⁰ werden hierdurch erleichtert. Falls es nötig sein sollte mehrere Versionen einer Grammatik oder aber auch mehrere verschiedene Grammatiken²¹ zu unterstützen, wurde vorsorglich ein Feld im Machina-Prozessmodell angelegt in dem diese Information hinterlegt ist.

¹⁷Direkt in JavaScript

¹⁸Siehe Abschnitt 5.14

¹⁹Im Gegensatz zu Standards wie OCL

²⁰Verwendung des deutschen Worts „und“ anstelle des englischen „and“.

²¹Auch Standard Grammatiken wären hier denkbar.

7.7.4. Binder

Die Binder-Transformation wurde bereits im Design in Abschnitt 5.6 besprochen. Nachdem er erfolgreich ausgeführt wurde sind sämtliche *QualifiedIdentifiers* entweder durch *MExpressions* der *DerivedTypes* ersetzt oder durch einen Workflowparameter in der Symboltabelle gebunden.

7.8. Activiti BPMN Synthese

Mit dieser Komponente ist es dem Übersetzer möglich BPMN für Activiti zu erzeugen.

7.8.1. Activiti-Synthesemodell

Für dieses Back-End wurde ein eigenes Synthesemodell erzeugt. Hierdurch konnte es weitgehend von der Übersetzer-Pipeline entkoppelt werden, auch war es damit nicht notwendig Elemente in das Machina-Prozessmodell einzutragen, die nur für Activiti relevant sind. Als angenehmen Nebeneffekt wurde diese Komponente hiermit in zwei Teile gespalten; eine Model-To-Model Transformation und eine Model-To-Text Transformation. Das vereinfacht die jeweilige Transformation erheblich.

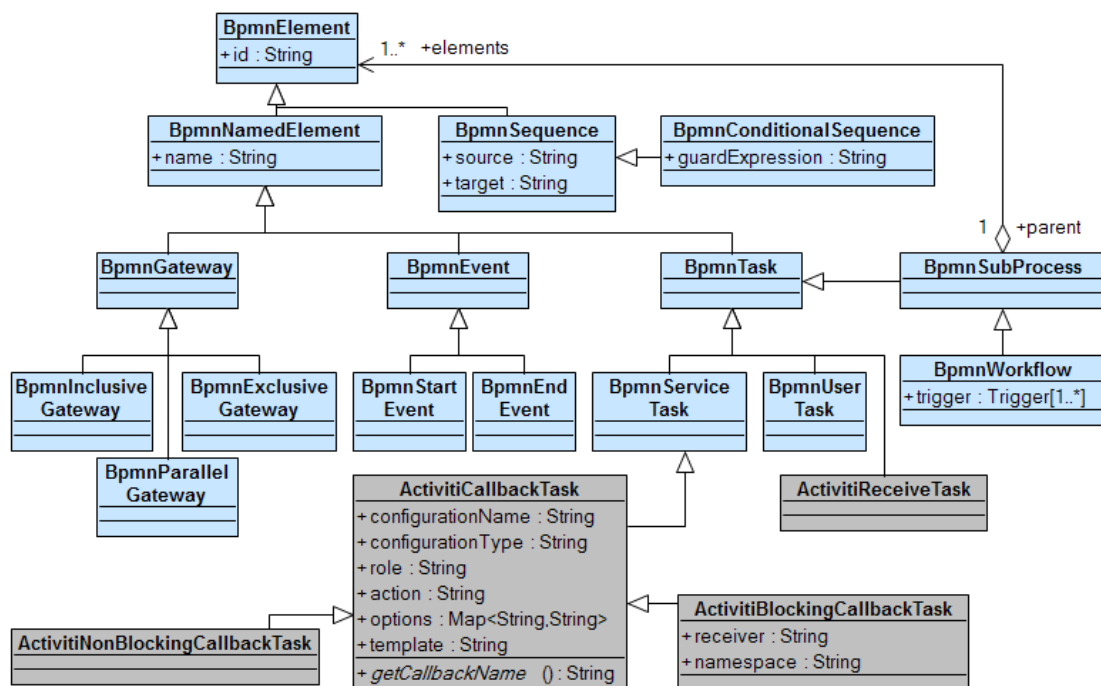


Abbildung 7.13.: Activiti-Synthesemodell

Abbildung 7.13 beschreibt das Activiti-Synthesemodell. Gegenüber dem Machina-Prozessmodell wurde es weiter vereinfacht. Es werden nur die für eine Umwandlung in BPMN notwendi-

gen Artefakte behalten. Zusätzlich wurden *Activiti** Elemente eingeführt, die die BPMN Erweiterungen von Activiti repräsentieren.

ActivitiCallbackTask Dieses Element speichert die Daten aus einem *MEnactmentBinding*. Diese Daten werden direkt in das entstehende BPMN generiert und während der Ausführung an die von durch *getCallbackName* festgelegte Implementierungsklasse²² übergeben. Diese Implementierungen führen dann diese Aktivität aus.

AcitivitiReceiverTask Dieses Element markiert einen Task, der auf einen *Trigger* mit *ContinueCommand* wartet. Diese werden immer im Zusammenspiel mit einem *AcitivitiBlockingCallbackTask* benutzt.

7.8.2. Model-To-Model Transformation

In dieser Transformation wird eine Instanz des Machina-Prozessmodells in eine Instanz des Activiti-Synthesemodell übertragen. Hier die Transformationsvorschriften für die einzelnen Elemente. Der größte semantische Unterschied ist durch die Entfernung der Assoziationen bedingt. Diese waren nur notwendig um eine einfachere Transformation zu erlauben. Der Elementgraph vereinfacht sich dadurch zu einem Elementbaum.

Einheitliche Identifier

BPMN verlangt, dass die eingesetzten *ids* in einem Workflow eindeutig sind. Der *ModelReader* erzeugt eindeutige Identifier. Deswegen wurde während der Transformationen darauf geachtet, dass diese Eindeutigkeit nicht verloren geht. Damit erhält man eine injektive Zuordnung von Elementen aus dem Ursprungsprozess zu den Elementen im Workflow.

MProcess und MSubProcess

Aus diesen werden *BpmnWorkflow* bzw. *BpmnSubProcess*. Als Besonderheit müssen diese rekursiv abgearbeitet werden, dabei transformiert man ihre jeweiligen *MElements* einzelnen. Während die *MAssociation* im Machina-Prozessmodell noch einen Sonderstatus als nicht *MElements* hatten, werden diese nun auch direkt in Subklassen *BpmnElemente* umgewandelt.

MStartEvent und MEndEvent

Diese Events werden direkt auf ihr BPMN Gegenstück abgebildet.

MParallelGate, MInclusiveGate und MExclusiveGate

Diese Gates werden direkt auf ihr BPMN Gegenstück abgebildet.

²² *BlockingTaskDelegate* oder *NonBlockingTaskDelegate*

MEactTask und MEactStartEvent

Bei diesem Elementen hängt es davon ab welchen Typ die Aktion in ihrem *MEactmentBinding* hat.

Blockierende Aktion Um einen blockierende Aktion auszuführen benötigt man mehr als ein Element. Abbildung 7.14 zeigt deren Zusammensetzung.

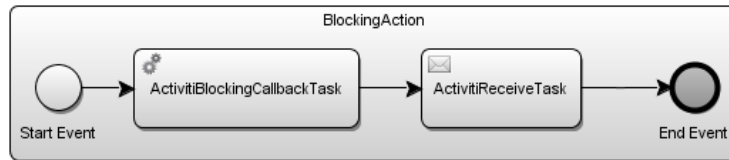


Abbildung 7.14.: Generiertes BPMN Fragment einer blockierenden Aktion

Zunächst wird ein *BpmnSubProcess* erzeugt, der die *id* des ursprünglichen *MEactTask* übernimmt. Dieser SubProcess wurde eingefügt, um die Assoziationen nicht verändern zu müssen. In diesem SubProzess werden nun, durch *BpmnSequence* verbunden, ein *BpmnStartEvent*, *ActivitiBlockingCallbackTask*, *ActivitiReceiveTask* und *BpmnEndEvent* erzeugt.

Prozessstartende Aktion Diese Aktion ist immer Teil eines *MEactStartEvent* und wird auf einen *BpmnStartEvent* abgebildet. Zusätzlich wird aus dem *MEactmentBinding* ein *Trigger* mit *StartProcessCommand* erzeugt und im *BpmnWorkflow* abgelegt.

Nicht-blockierende Aktion Nicht-blockierende Aktionen können direkt auf einen *ActivitiNonBlockingCallbackTask* abgebildet werden.

MEactEndEvent

Ein *MEactEndEvent* hat immer eine nicht-blockierende Aktion. Er wird als *ActivitiNonBlockingCallback* und anschließendem *BpmnEndEvent* abgebildet.

MAssociation

Bei der Abbildung der *MAssociation* wird unterschieden ob diese eine *MExpression* tragen. Falls sie keine Expression haben, wird direkt auf *BpmnSequence* abgebildet. Die mit *MExpression* werden auf *BpmnConditionalSequence* abgebildet.

MExpression

Da Activiti JUEL als Sprache für Ausdrücke verwendet, musste diese während dieser Transformation aus den *MExpressions* generiert werden. JUEL benötigt für Variablen

gültige Java-Bezeichner²³, damit war die größte Herausforderung die Workflowparameter auf diese Bezeichner abzubilden. Die Lösung wurde mit Base32²⁴ gefunden. Base32 verwendet nur Zeichen die in Java-Bezeichnern erlaubt sind. Um zu verhindern, dass ein Bezeichner mit einer Zahl beginnt oder es ein Java Keyword ergibt, wurde zusätzlich ein Unterstrich als Prefix bzw. Pfadtrennzeichen eingefügt. Hierdurch ist gewährleistet das alle Bezeichner gültig sind. Zusätzlich garantiert die Bijektivität von Base32, dass nicht mehrere Unterschiedliche Workflowparameter auf den gleichen Bezeichner abgebildet werden.

7.8.3. Model-To-Text Transformation

Im Anschluss an diese Transformationen muss lediglich der Baum als XML herausgeschrieben werden. Dieses Herausschreiben wird durch ein Visitor-Pattern [GHJV94, S. 331 ff.] implementiert. Die XML Repräsentation der einzelnen BPMN Elemente wurde bereits in Abschnitt 6.1.1 beschrieben. Ein Beispiel für eine generierte Blockierende Aktion ist in Quelltextfragment A.2 im Anhang zu finden.

²³Für die Regeln siehe [GJS⁺13, S. 38]

²⁴Format wird unter [Jos03] beschrieben

8. Bewertung

8.1. Vergleich mit Anforderungen

Da die Anforderungen an Machina lediglich gewünschte Funktionalität waren, wird hier analysiert, welche Anforderungen tatsächlich wie weit umgesetzt werden konnten.

Allgemeine Funktionale Anforderungen

REQ 1 - teilweise Als Standards wurde BPMN für die Workflowbeschreibung eingesetzt. Leider ist es notwendig die Erweiterungen von Activiti zu benutzen. Zusätzlich wurden einige der Java EE Industriestandards eingesetzt.

REQ 2 - erfüllt Durch den Einsatz eines Nachrichtenbusses besteht keine Verfügbarkeitskopplung der Werkzeuge und Machina.

REQ 3 - erfüllt Die Reaktionszeit wurde durch einige einfache Tests¹ mit $<0,1$ Sekunden bestimmt. Das ist deutlich von der selbst auferlegten Grenze von einer Sekunde entfernt.

Angebundene Werkzeuge

REQ 4 - nicht erfüllt Die Werkzeugkonfigurationen werden noch direkt im Dateisystem abgelegt.

REQ 5 - erfüllt Es besteht eine einfache Möglichkeit neue Werkzeuge hinzuzufügen. Im Selbstversuch konnte ein E-Mail-Modul mit einer *NotificationRole* in etwa drei Stunden implementiert werden.

REQ 6 - teilweise Es sind einige Erweiterungen an den Werkzeugen notwendig, insbesondere die Erweiterung um einen Mechanismus der Events generiert und versendet.

REQ 7 - unbekannt Machina versucht sich den gewohnten Arbeitsabläufen anzupassen, ob dies gelingt ist ohne realen Anwendungsfall nicht beweis- oder widerlegbar.

REQ 8 - teilweise erfüllt Ein direkter Zugriff auf den Kontext des Workflows wurde geschaffen, es fehlt jedoch an Funktionalität um einen wirklichen Nutzen für den Kunden zu bieten.

¹allerdings nicht wissenschaftlich rigoros

Modellierung

REQ 9 - erfüllt Das Datenformat wird unterstützt.

REQ 10 - erfüllt Durch die Implementierungen der *Semantic* Schnittstelle sind beliebige projektspezifische Semantiken möglich.

REQ 11 - erfüllt Das Prozessmodell muss um *MEnactments* erweitert werden. Das Enactment-Konzept enthält nur Daten über Rollen. Es besteht kein direkter Bezug zum Laufzeitsystem.

REQ 12 - teilweise Fehlermeldungen werden nur als Stacktrace in den Logausgaben generiert. Es ist aber ohne großen Aufwand möglich diese an den Modellierer weiterzugeben.

Laufzeit

REQ 13 - erfüllt Werkzeuge sind voneinander unabhängig.

REQ 14 - erfüllt Mehrere Werkzeuginstanzen des gleichen Werkzeugs sind möglich.

REQ 15 - erfüllt Das Speichern von Laufzeitinformationen über die Workflowinstanzen wird von Activiti direkt angeboten.

REQ 16 - erfüllt Theoretisch erfüllt durch das Nachrichtenschema, es wurden aber noch keine Rauchtests² durchgeführt.

REQ 17 - nicht erfüllt Eskalation an den Administrator wurde noch nicht implementiert.

8.2. Quelltextmetrik

Die einzelnen Komponenten von Machina sind in unterschiedlichen Graden der Vollständigkeit implementiert. Deswegen ist diese Metrik nur eine grobe Abschätzung für ein fertiges Produkt. Sie kann lediglich Tendenzen aufzeigen. Insgesamt wurden etwa 16.000³ Quelltextzeilen erzeugt, die sich folgendermaßen auf die einzelnen Komponenten verteilen:

- Übersetzer: etwa 6.000³ Zeilen verteilt auf etwa 170 Klassen
- Expression-Parser: etwa 2.500³ Zeilen von ANTLR generiert verteilt auf 2 Klassen
- Jira-Modul: etwa 2.000³ Zeilen verteilt auf etwa 40 Klassen
- Activiti-Adapter: etwa 600³ Zeilen verteilt auf 15 Klassen

²In diesem Fall: Experimente, die versuchen Inkonsistenzen zu erzeugen.

³Gemessen nach SLOC

- GUI: etwa 900³ Zeilen verteilt auf 31 Klassen
- Rest: etwa 4.000³ Zeilen mit etwa 130 Klassen, davon etwa 600³ für das Rollenkonzept.

Zusätzlich zu diesen produktiven Quelltexten wurden Tests für etwa 35% Codecoverage erzeugt. Was sich in dieser Metrik deutlich abzeichnet, ist die geringe Anzahl an Quelltextzeilen für das Rollenkonzept. Die erhoffte Einsparung ist damit nur minimal ausgefallen.

8.3. Offene Punkte

Wie bei einem Prototypen zu erwarten, sind noch viele Punkte offen geblieben. Diese offenen Punkte wurden in drei Kategorien eingeteilt. „Detailverbesserungen“, sind offene Punkte, die keine offensichtlichen Fragen mehr beinhalten. Hier fehlt es lediglich an der handwerklichen Umsetzung. „Offene Fragen“, sind die Punkte bei denen eine Umsetzung möglicherweise Probleme aufwirft die sich im ganzen Design niederschlagen. Unter „Zukünftiger Arbeit“ sind diejenigen Punkte gesammelt, die definitiv Probleme aufwerfen werden die gelöst werden müssen.

8.3.1. Detailverbesserungen

Diese offenen Punkte beinhalten keine offensichtlichen Fragen mehr. Es fehlt lediglich an der handwerklichen Umsetzung.

- Zu wenige Tests. Eine Testabdeckung von 35% ist zu gering.
- Die Validierungsschritte in der Übersetzer-Pipeline sind unvollständig.
- Die Übersetzer-Pipeline ist derzeit statisch festgelegt. Hier wäre eine Erweiterung interessant, die es der projektspezifischen Semantik erlaubt noch zusätzliche Transformationen hinzuzufügen.
- Typinference in den Expressions ist nur teilweise umgesetzt. Dies muss vervollständigt werden, um sinnvolle Fehlermeldungen zu erzeugen.

8.3.2. Offene Fragen

Diese offenen Punkte gehen über eine reine handwerkliche Umsetzung hinaus. Es sind zusätzliche Analysen notwendig, um eine sinnvolle Umsetzung zu finden. Hierdurch könnten neue Fragen aufgeworfen werden oder Änderungen am Design notwendig sein.

- Eine Funktionalität, die Aufgaben direkt an einzelne Benutzer zuweist, ist derzeit nicht implementiert und wirft neue Fragen in Bezug auf Integration und Binding auf.

- Die Übersetzer-Pipeline sollte zunächst Standard BPMN erzeugen. Erst in einem zweiten unabhängigen Lauf sollte dieses Standard BPMN in Activiti spezifisches BPMN für Machina umgewandelt werden. Dies würde den Übersetzer einerseits vereinfachen, aber auch das Einbinden von Prozessmodellen die, nicht von Stages kommen, leichter machen.
- Es ist leider nicht bekannt, wie viele der Projekte sich mit dieser in Abschnitt 7.7.1 beschriebenen Semantik umsetzen lassen. Die hierfür notwendigen Projekte aus der Praxis fehlen.
- Es sind nur zwei Rollen prototypisch definiert. Es müssen mehrere Werkzeuge verglichen werden, um sinnvolle Abstraktionen erzeugen zu können.
- Die Timeout Transition in der Zustandsmaschine im *EventControl* (siehe Abschnitt 7.5) ist noch nicht implementiert. Es ist besonders darauf zu achten, dass ein Timeout nicht zu schnell passiert und der Event versehentlich zweimal ausführt wird. Ein Lease-Mechanismus könnte dies ermöglichen. Ein anderer Ansatz wäre, dass man keinen Timeout benutzt, sondern nur beim Starten von Machina noch laufend Ausführungsversuche zurücksetzt.
- Das Löschen der *EventLog* Einträge in der Datenbank ist noch nicht umgesetzt. Eine geregeltes Löschen dieser Einträge ist schwierig, da man garantieren muss, dass definitiv keine Duplikate dieses Events mehr in irgendeinem der Zwischenspeicher vorhanden sind. Ein Checkpointsystem [GR93, S. 604 ff.], wie es für das Journal eines Datenbankensystems Anwendung findet, ist hierfür ein vielversprechender Ansatz.
- Der Mechanismus für Anforderung REQ 17, wie der Administrator Nachrichten zur Korrektur von Problemen in den Workflowinstanzen erzeugen könnte, ist noch unklar.

8.3.3. Zukünftige Arbeit

Diese offenen Punkte werden wahrscheinlich große Herausforderungen beinhalten. Es werden zwar schon erste Ideen genannt wie man diese angeht, diese könnten sich aber während der Bearbeitung als falsch herausstellen.

- Ein Update der Anwendung auf eine neue Version ist derzeit sehr schwierig durchzuführen. Die durch ein Update möglicherweise entstehenden, nicht mehr verarbeitbaren Nachrichten, würden zu Fehlern in der Ausführung führen. Diese müssten vom Administrator händisch behoben werden. Durch einen speziellen Modus, der ein geregeltes Herunterfahren ermöglicht könnte dieser Aufwand reduziert werden.
- Die statische Bindung macht es unmöglich inkompatible Änderungen an den Werkzeugen vorzunehmen. Das Umstellen von statischem Binding auf dynamisches Binding könnte dies lösen, wirft aber sehr viele Fragen auf.

- Update einer laufenden Workflowinstanz auf eine neuere Version des Workflows wäre für potentiell lang⁴ laufende Prozesse sehr wichtig. Aber auch kürzer laufende Prozesse könnten hiervon profitieren.
- Derzeit ist nur eine Machina-Instanz möglich. Durch einen Cluster aus mehreren Instanzen könnte die Verfügbarkeit gesteigert werden. Die Performance könnte hiervon auch profitieren. Die Auswirkungen auf das Nachrichtenverarbeitungsschema und die interne Synchronisation sind schwer abzuschätzen.

⁴Monate und darüber hinaus

9. Fazit

Machina konnte im Großen und Ganzen seine Anforderungen erfüllen. Als größte Stärke aber auch als größte Schwäche sehe ich das Rollenkonzept. Es erlaubt das vom operativen vollständig unabhängige Modellieren der Prozesse und ist damit nach [Gad10] die richtige Vorgehensweise. Allerdings sind die Einschränkungen durch ein generisches Zwischenformat massiv und möglicherweise für eine praktische Anwendung ungeeignet.

Rollenkonzept Das Rollenkonzept wurde nötig um Anforderung REQ 11 erfüllen zu können. Gäbe man diese Anforderung auf, kann man schon im Prozessmodell festlegen was für ein Werkzeug sich hinter bestimmten Aktivitäten befindet. Man benötigt also kein abstraktes Vokabular für eine Rolle, das von allen Werkzeugen implementiert werden muss die diese Rolle unterstützen wollen. Dies erhöht erheblich die mögliche Funktionalität, die in den Werkzeugen angesprochen werden kann. Damit einfach Umzusetzen wäre die Aufnahme turing-vollständige Skripte in den Prozess, die dann direkt von JIRA als Effekt einer Aktion ausgeführt werden.

Wie die Quelltext-Metriken derzeit aussagen, wurde der erhoffte Vorteil¹ des Rollenkonzepts nicht erfüllt. Es wurden lediglich 600 Quelltextzeilen in Rollen ausgelagert, das sind weniger als 30% der Zeilen eines Werkzeugmoduls. Ein abschließendes Urteil ist allerdings noch zu früh. Es ist bisher nur eine Rolle/Werkzeug umgesetzt und das auch nur prototypisch. Diese Werte könnten sich für eine produktive Variante stark verändern.

JMS als Nachrichtenbus Der Einsatz von JMS als Nachrichtenbus bleibt zweifelhaft. Es ist nicht möglich zu überprüfen ob Nachrichten mit bestimmten Werten bereits gesendet wurden. Dies hat weitreichende Konsequenzen für das Nachrichtenverarbeitungsschema, weil dadurch der Nachrichtenbus nicht testbar ist, was zur Duplikatsvermeidung notwendig gewesen wäre. Da der Nachrichtenbus lediglich als persistente verteilte Nachrichtenwarteschlange² benutzt wird, hätte eine Implementierung basierend auf JPA³ dies auch erfüllen können. Die auf JPA basierende Implementierung hätte jedoch den Vorteil, dass man diese leicht testbar machen kann.

Deklaration der Symbole im Prozessmodell Die Symbole im Prozessmodell werden nicht explizit deklariert. Sie werden durch Verwendung in einer *MExpression* implizit deklariert und durch Typinference typisiert. Diese Entscheidung hat den Vorteil, dass ein Standard-Binding leicht möglich ist. Allerdings verliert der Modellierer hierdurch

¹Nutzen von Synergien durch Verschieben von gemeinsamen Code aus den Werkzeugmodulen

²Ohne schwer zu erfüllende Performance Anforderungen

³Java Persistence Api: Details im Abschnitt 6.6

leichter den Überblick. Welcher Ansatz für alle Beteiligten angenehmer ist, muss sich in der Praxis herausstellen. Ein Umstellen auf ein explizites Typmodell ist derzeit noch möglich. Sobald bestehende Projekte mit Machina laufen, wird eine Migration schwierig.

A. Anhang

A.1. Beispiel Semantik

```
var elementSelector = {
  isTask : function(element) {
    return element.getType() == ProcessElementType.ACTIVITY //
      && ("activity".equals(element.getElementSubtype()) || null ↯
        ↯ = element.getElementSubtype())
      && !(mna.isStartAction(element.getAttribute(roleToken)));
  },

  isEvent : function(element) {
    return this.isStartEvent(element);
  },

  isStartEvent : function(element) {
    return element.getType() == ProcessElementType.ACTIVITY //
      && ("activity".equals(element.getElementSubtype()) || null ↯
        ↯ = element.getElementSubtype())
      && (mna.isStartAction(element.getAttribute(roleToken))); // ↯
        ↯ See if the enactment action of that element is a ↯
        ↯ process-starting action
  },
};

var associationSelector = {
  isScriptAssociation : function(association) {
    return association.isEnactmentActive() && null != ↯
      ↯ association.getResourceExportGuid();
  }
};

var elementSelect = mna.create(ElementSelector, new ↯
  ↯ ElementSelector(elementSelector));
var associationSelect = mna.create(AssociationSelector, new ↯
  ↯ AssociationSelector(associationSelector));

var builder = {
  base : function () {
    return new FactoryElementHandler(elementSelect);
  }
};

// Add specific phases
var semantic = new DefaultSemantic();
```

```

semantic.addPhase(mna.create(ProcessElementHandler, new ↵
    ↵ ProcessElementHandler(builder)).createPhase());
semantic.addPhase(new ↵
    ↵ FactoryAssociationHandler(associationSelect).createPhase());
semantic.addPhase(new Segmentation().createPhase());
semantic.addPhase(new ProcessBuilder().createPhase());
semantic;

```

Listing A.1: Beispiel semantic.js

A.2. Analyse Change Request Fragment

```

<subProcess id="L5KEKT2UGBEE44KFMWWTMOBYNF3EKNT2NVNEC">
  <startEvent id="L5KEKT2UGBEE44KFMWWTMOBYNF3EKNT2NVNEC_start"
    name="Analyse_Change_Request_start">
  </startEvent>
  <endEvent id="L5KEKT2UGBEE44KFMWWTMOBYNF3EKNT2NVNEC_end"
    name="Analyse_Change_Request_end">
  </endEvent>
  <serviceTask id="L5KEKT2UGBEE44KFMWWTMOBYNF3EKNT2NVNEC_send"
    name="Analyse_Change_Request"
    activiti:class="de.methodpark.machina.activiti.callback.BlockingTaskCallback">
    <extensionElements>
      <activiti:field name="role">
        <activiti:string>tracker</activiti:string>
      </activiti:field>
      <activiti:field name="action">
        <activiti:string>createIssue</activiti:string>
      </activiti:field>
      <activiti:field name="configurationName">
        <activiti:string>privateJira</activiti:string>
      </activiti:field>
      <activiti:field name="configurationType">
        <activiti:string>jira</activiti:string>
      </activiti:field>
      <activiti:field name="configurationOptions">
        <activiti:string>{"machinaurlfieldid":"10000",
          "issuetype":"Change_Activity"}</activiti:string>
      </activiti:field>
      <activiti:field name="payloadType">
        <activiti:string>json</activiti:string>
      </activiti:field>
      <activiti:field name="payloadData">
        <activiti:expression>{
          "elementUrl":"${"http://127.0.0.1:8080/pkit/go/pelement.do ↵
            ↵ ?id=56036&type=Activity"}",
          "parent":"${_L5BHCRZYMNEDO6KFMN4VIMDPMVUWWWMSEKVBHO_NNSXS}", "summary" ↵
            ↵ : "${"Analyse Change Request"} ↵
            ↵ ${_L5BHCRZYMNEDO6KFMN4VIMDPMVUWWWMSEKVBHO_NNSXS}: ↵
            ↵ ${_L5BHCRZYMNEDO6KFMN4VIMDPMVUWWWMSEKVBHO_ON2W23LBOJ4Q}"
          }
        </activiti:expression>
      </activiti:field>
      <activiti:field name="receiver">

```

```

        <activiti:string>L5KEKT2UGBEE44KFMYWTMOBYNF3EKNT2NVNEC_receive ↵
        ↵ </activiti:string>
    </activiti:field>
    <activiti:field name="namespace">
        <activiti:string>L5KEKT2UGBEE44KFMYWTMOBYNF3EKNT2NVNEC ↵
        ↵ </activiti:string>
    </activiti:field>
</extensionElements>
</serviceTask>
<userTask id="L5KEKT2UGBEE44KFMYWTMOBYNF3EKNT2NVNEC_receive"
    name="Analyse_Change_Request">
</userTask>
<sequenceFlow
    id="L5KEKT2UGBEE44KFMYWTMOBYNF3EKNT2NVNEC_start_↵ ↵
    ↵ L5KEKT2UGBEE44KFMYWTMOBYNF3EKNT2NVNEC_send"
    sourceRef="L5KEKT2UGBEE44KFMYWTMOBYNF3EKNT2NVNEC_start" ↵ ↵
    ↵ targetRef="L5KEKT2UGBEE44KFMYWTMOBYNF3EKNT2NVNEC_send" />
<sequenceFlow
    id="L5KEKT2UGBEE44KFMYWTMOBYNF3EKNT2NVNEC_send_↵ ↵
    ↵ L5KEKT2UGBEE44KFMYWTMOBYNF3EKNT2NVNEC_receive"
    sourceRef="L5KEKT2UGBEE44KFMYWTMOBYNF3EKNT2NVNEC_send" ↵ ↵
    ↵ targetRef="L5KEKT2UGBEE44KFMYWTMOBYNF3EKNT2NVNEC_receive" />
<sequenceFlow
    id="L5KEKT2UGBEE44KFMYWTMOBYNF3EKNT2NVNEC_receive_↵ ↵
    ↵ L5KEKT2UGBEE44KFMYWTMOBYNF3EKNT2NVNEC_end"
    sourceRef="L5KEKT2UGBEE44KFMYWTMOBYNF3EKNT2NVNEC_receive"
    targetRef="L5KEKT2UGBEE44KFMYWTMOBYNF3EKNT2NVNEC_end" />
</subProcess>

```

Listing A.2: Generiertes BPMN Fragment für Analyse Change Request

A.3. Erweiterbare Standardsemantik für Stages

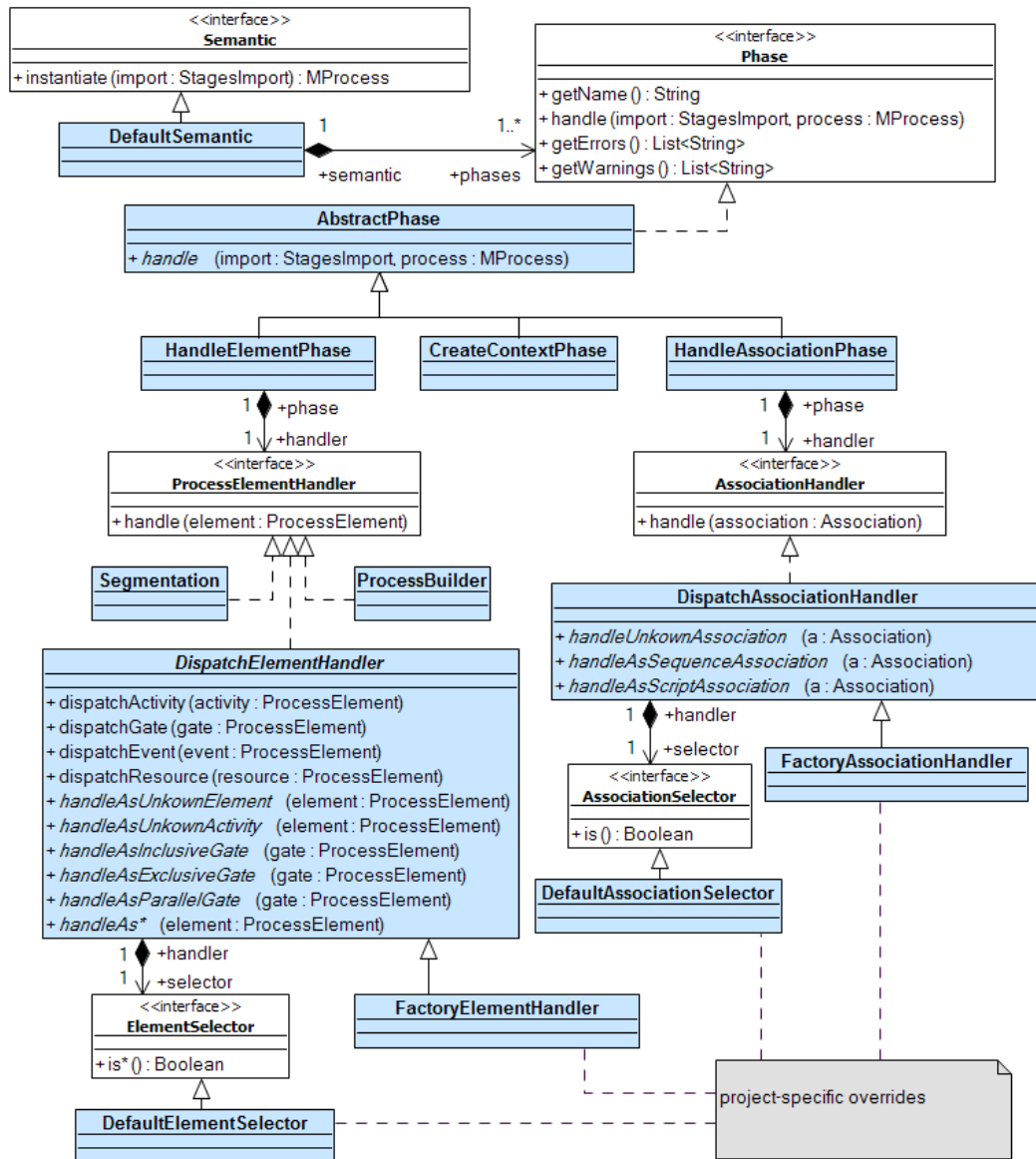


Abbildung A.1.: Erweiterbare Standardsemantik für Stages-Prozesse

Literaturverzeichnis

- [BMR⁺96] BUSCHMANN, Frank ; MEUNIER, Regine ; ROHNERT, Hans ; SOMMERLAD, Peter ; STAL, Michael: *Pattern-Oriented Software Architecture: A System of Patterns*. Chichester : Wiley, 1996
- [BN09] BERNSTEIN, Philip ; NEWCOMER, Eric: *Principles of Transaction Processing*. San Francisco : Morgan Kaufmann, 2009
- [Fow04] FOWLER, Martin: *Inversion of Control Containers and the Dependency Injection pattern*. <http://www.martinfowler.com/articles/injection.html>. Version: January 2004
- [Gad10] GADATSCH, Andreas: *Grundkurs Geschäftsprozess-Management: Methoden und Werkzeuge für die IT-Praxis: Eine Einführung für Studenten und Praktiker*. 6.Auflage. Wiesbaden : Vieweg+Teubner Verlag, 2010
- [GHJV94] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns: elements of Reusable Object-Oriented Software*. Boston : Addison Wesley, 1994
- [GJS⁺13] GOSLING, James ; JOY, Bill ; STEELE, Guy ; BRACHA, Gilad ; BUCKLEY, Alex: *The Java(TM) Language Specification Java SE 7 Edition*. <http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf>. Version: February 2013
- [GR93] GRAY, Jim ; REUTER, Andreas: *Transaction Processing: Concepts and Techniques*. San Francisco : Morgan Kaufmann Publishers, 1993
- [HW04] HOHPE, Gregor ; WOOLF, Bobby: *Enterprise Integration Patterns*. Boston : Addison Wesley, 2004
- [Jos03] JOSEFSSON, S.: *The Base16, Base32, and Base64 Data Encodings*. <http://www.ietf.org/rfc/rfc3548.txt>. Version: July 2003
- [Kur11] KURUVILLA, Jobin: *JIRA Development Cookbook*. Birmingham : Packt Publishing, 2011
- [LBK07] LENZ, Richard ; BEYER, Mario ; KUHN, Klaus A.: Semantic integration in healthcare networks. In: *International Journal of Medical Informatics* 76 (2007), S. 201–207
- [Li11] LI, Patrick: *JIRA 4 Essentials*. Birmingham : Packt Publishing, 2011

- [Mic02] MICROSYSTEMS, Sun: *Java(TM) Message Service Specification Final Release 1.1*. <http://download.oracle.com/otndocs/jcp/7195-jms-1.1-fr-spec-oth-JSpec/>. Version: April 2002
- [Mic06] MICROSYSTEMS, Sun: *JSR 245: JavaServer(TM) Pages 2.1 Expression Language Specification*. <http://www.jcp.org/en/jsr/detail?id=245>. Version: May 2006
- [Mic09] MICROSYSTEMS, Sun: *JSR 317: Java(TM) Persistence API, Version 2.0*. <http://download.oracle.com/otndocs/jcp/persistence-2.0-fr-eval-oth-JSpec/>. Version: November 2009
- [OMG11] OMG: *Business Process Model and Notation (BPMN) Version 2.0*. <http://www.omg.org/spec/BPMN/2.0/PDF>. Version: January 2011
- [Par07] PARR, Terence: *The Definitive ANTLR Reference*. Raleigh : Pragmatic Bookshelf, 2007
- [Rad12] RADEMAKERS, Tijs: *Activiti in Action*. Shelter Island : Manning, 2012
- [RMHC09] RICHARDS, Mark ; MONSON-HAEFEL, Richard ; CHAPPELL, David A.: *Java Message Service, Second Edition*. Sebastopol : O'Reilly, 2009
- [Ros07] ROS, Sebastien: *State of the Art Expression Evaluation*. <http://www.codeproject.com/Articles/18880/State-of-the-Art-Expression-Evaluation>. Version: November 2007
- [Sou10] SOURCE, Spring: *Spring Framework Reference Documentation*. <http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/>. Version: 2010
- [Sti13] STIEHL, Volker: *Prozessgesteuerte Anwendungen entwickeln und ausführen mit BPMN*. Heidelberg : dpunkt.verlag, 2013