

Ein Event-Subscription Framework für Prozessengines: Design und Implementierung

Autor

Gerardo Navarro Suarez

Betreuer

Prof. Dr. Mathias Weske

MSc. Matthias Kunze

MSc. Alexander Lübke

Lehrstuhl für Business Process Technology

Datum der Abgabe: 30. Juni 2011

Ich erkläre hiermit, dass ich die vorliegende Bachelorarbeit selbstständig verfasst, Zitate kenntlich gemacht und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Potsdam, 30. Juni 2011

Gerardo Navarro Suarez

Die Verarbeitung von Ereignissen ist ein wichtiger Bestandteil bei der Ausführung von Geschäftsprozessen, besonders bei der Prozessinstanziierung. Neben der einfachen Instanziierung durch ein eindeutig gekennzeichnetes Starterereignis, die von allen Prozessengines unterstützt wird, gibt es auch komplexere Möglichkeiten. Beispielsweise kann die Prozessinstanziierung durch eine Kombination aus mehreren Starterereignissen angestoßen werden.

Diese komplexe Art der Instanziierung kann von den meisten Prozessengines nicht selbstständig ausgeführt werden, weil sie nicht eigenständig auf Ereignisse reagieren können. Daher ist man gezwungen, den Geschäftsprozess umständlich hinsichtlich der Prozessengine zu verändern oder externe Anwendungen zu integrieren, welche die Verarbeitung von eintretenden Ereignissen übernimmt.

Mit der *JodaEngine* wurde eine Prozessengine entwickelt, welche die Einschränkungen anderer Prozessengines aufhebt, indem sie selbstständig Ereignisse im Verlauf der Prozessausführung verarbeiten kann. Diese Bachelorarbeit beschreibt das *Event-Subscription Framework* als Teil der *JodaEngine*, mit welchem sich auf Ereignisse bei Prozessinstanziierung und -ausführung registriert und deregistriert werden kann. Das Framework stellt eine technische Grundlage bereit, um verschiedene Ansätze und Szenarien der *Event-Subscription* technisch umzusetzen. Abschließend wird im Verlauf dieser Arbeit die Vollständigkeit des Frameworks im Hinblick auf verschiedene Anwendungsszenarien untersucht.

Inhaltsverzeichnis

1	Einleitung	1
2	Analyse	3
2.1	Anwendungsszenario	3
2.1.1	Event-Subscription bei Prozessinstanziierung	3
2.1.2	Event-Subscription bei Prozessausführung	4
2.1.3	Gesamtszenario für Event-Subscription	5
2.2	Anforderungen	6
3	Verwandte Arbeiten	8
4	Implementierung	13
4.1	Prozessinstanziierung	14
4.1.1	Lebenszyklus einer Prozessdefinition	15
4.1.2	Implementierung	16
4.2	Event-Subscription im Eventmanager	19
4.2.1	Konzept	19
4.2.2	Implementierung	21
4.3	Event-Subscription Framework	23
4.3.1	IncomingProcessEvent	24
4.3.2	ProcessEventGroup	25
4.3.3	EventSubscriptionManagement	27
4.4	Prozessinstanziierung und Event-Subscription Framework in der Praxis	27
4.4.1	Startereignisse bei Prozessinstanziierung	28
4.4.2	Zwischenereignisse bei Prozessausführung	30
5	Ausblick	34
6	Zusammenfassung	37
	Glossar	38
	Quellenverzeichnis	39

Abbildungsverzeichnis

1	Referenzprozess zur Bearbeitung eines Krankenversicherungsantrags	6
2	Ereignisverarbeitenden Infrastruktur für das CASU-Framework	12
3	Architektur der <i>JodaEngine</i>	14
4	Lebenszyklus einer Prozessdefinition	15
5	Implementierung der Prozessdefinition	17
6	Konzept des Eventmanagers	19
7	Implementierung des Eventmanager-Konzepts	21
8	Klassenmodell des <i>Event-Subscription Frameworks</i>	24
9	Integration der ProcessEventGroup ins <i>Event-Subscription Framework</i>	26
10	Referenzprozess für zukünftige Arbeiten	34

Codebeispiel

1	Implementierung der BpmnEventBasedXorGateway-Activity	31
2	trigger()-Methode in der ExclusiveIntermediateProcessEventGroup .	31
3	trigger()-Methode in der AbstractThreadSafeIntermediateProcessEventGroup	32

1 Einleitung

In den letzten Jahren hat sich Business Process Management (BPM) zu einem spannenden Feld entwickelt, sowohl in der Betriebswirtschaft als auch in der Informatik, denn es stellt Technologien und Methoden bereit, um Geschäftsprozesse zu verwalten. Da sich Unternehmen ständig neuen Marktsituationen anpassen müssen, ist es notwendig, dass sie ihre Geschäftsabläufe flexibel umstellen und automatisieren können.

Bei Geschäftsprozessen mit hohem Automatisierungsgrad werden Prozessengines eingesetzt, mit welchen es möglich ist, diese Prozesse auf technische Weise zu definieren und sie letztendlich auch auszuführen. Eine Prozessengine steuert die Ausführung des Geschäftsablaufs und entscheidet unter welchen Bedingungen welche Aufgaben und Aktivitäten stattfinden sollen [1].

Bei der Ausführung eines Geschäftsprozesses kann es vorkommen, dass die Prozessengine mit ihrer Umgebung interagiert. Einige dieser Interaktionen können als eintretende Ereignisse verstanden werden, die das Auftreten einer bestimmten Aktion aus der Umgebung repräsentieren [2][3], wie das Eingehen einer E-Mail.

Die meisten Prozessengines unterstützen jedoch nicht selbstständig die Ausführung von Geschäftsprozessen, die auf eintretende Ereignisse warten. In manchen Fällen kann daher die Ereignisverarbeitung von separaten Anwendungen übernommen werden, welche die Prozessengine beim Eintreten von Ereignissen benachrichtigt. Allerdings können Ereignisse auch in komplexen Szenarien enthalten sein, wie beim ereignisbasierten Gateway der BPMN¹. Diese können selbst mit externen ereignisverarbeitenden Anwendungen nicht ausgeführt werden können, da für diese Szenarien wichtige Informationen über die Prozessdefinition und -ausführung benötigt werden, die nur innerhalb der Prozessengine gekapselt sind. Daher sollte die Prozessengine selbstständig in der Lage sein, eingehende Ereignisse zu verarbeiten.

Im Rahmen des Bachelorprojektes *JodaEngine: Process Enactment Platform*, das von sechs Bachelorstudenten am Hasso-Plattner-Institut (HPI) Potsdam unter der Leitung des Fachgebiets für “Business Process Technology“ (BPT) durchgeführt wurde, ist mit der *JodaEngine* eine Prozessengine entwickelt worden, die durch ihre mo-

¹Business Process Model Notation, kurz BPMN, ein Standard zur Modellierung von Geschäftsprozessen

dulare Bauweise unter anderem eintretende Ereignisse empfangen und verarbeiten kann.

Dabei stellt die *JodaEngine* einen Mechanismus bereit, damit sich die Prozessausführung auf Ereignisse registrieren oder deregistrieren kann. Dieser Mechanismus wird als *Event-Subscription* bezeichnet und ist besonders im Hinblick auf die Prozessinstanziierung ein wichtiges Hilfsmittel, da verschiedene Ereignisse einen Prozess starten können. Der *Event-Subscription-Mechanismus* wird im Rahmen der *JodaEngine* als Framework bereitgestellt, das eine technische Grundlage, Schnittstellen und Konzepte anbietet, um unter anderem verschiedene Ansätze der *Event-Subscription* technisch umzusetzen.

Diese Bachelorarbeit konzentriert sich auf die *Event-Subscription* im Verlauf der Prozessausführung, die grundsätzlich beim Instanzieren eines Prozesses oder während der Bearbeitung der einzelnen Prozesselemente erfolgen kann.

Zunächst werden in Kapitel 2 Anwendungsszenarien für beide Fälle beschrieben, aus denen anschließend die Anforderungen für das *Event-Subscription Framework* erhoben werden. In Kapitel 3 werden andere Prozessengines hinsichtlich der *Event-Subscription* untersucht und es wird ein konzeptionelles Framework zur Prozessinstanziierung, das CASU-Framework, näher erläutert. Kapitel 4 zeigt die Implementierung der Prozessinstanziierung und des *Event-Subscription Frameworks*, mit welchen die *JodaEngine* in der Lage ist, selbstständig Geschäftsprozesse auszuführen, die derzeitige Prozessengines nicht unterstützen können. Vor der Zusammenfassung wird zunächst noch in Kapitel 5 ein Ausblick hinsichtlich Erweiterungsmöglichkeiten und zukünftigen Arbeiten gegeben.

2 Analyse

In diesem Kapitel sollen die verschiedenen Anwendungsszenarien beschrieben werden, die vom *Event-Subscription Framework* und der Prozessinstanziierung unterstützt werden sollen. Die Anwendungsfälle bilden die Grundlage für die anschließende Anforderungsanalyse, in welcher die Anforderungen für das *Event-Subscription Framework* und die Prozessinstanziierung erhoben werden sollen.

2.1 Anwendungsszenario

Damit die Anforderungen korrekt und vollständig erhoben werden können, müssen zunächst Geschäftsprozesse modelliert werden, die die *JodaEngine* mit Hilfe des *Event-Subscription Frameworks* letztendlich ausführen kann. Wie bereits in der Einleitung erläutert, bietet das *Event-Subscription Framework* die Registrierung auf bestimmte Ereignisse an, auf die zur Prozessinstanziierung und während der Prozessausführung reagiert werden muss. Daher befasst sich das Anwendungsszenario hauptsächlich mit den verschiedenen Stellen im Prozess, an welchen Ereignisse registriert werden können.

Das Anwendungsszenario soll als Geschäftsprozess visualisiert werden, für dessen Darstellung und Definition die Modellierungssprache BPMN ausgewählt wurde. In der BPMN können Ereignisse in Start-, Zwischen- und Endereignisse eingeteilt werden [4, Kapitel 10.4], wobei im Hinblick auf die *Event-Subscription* nur Start- und Zwischenereignisse eine hervorzuhebende Rolle spielen. In den nächsten Abschnitten werden zunächst Szenarien für beide Ereignisarten vorgestellt. Dabei wird unter anderem die Frage untersucht, wie diese Szenarien in der Praxis eingesetzt werden.

2.1.1 Event-Subscription bei Prozessinstanziierung

Geschäftsprozesse zeichnen sich durch eine klar definierte Eingabe und Ausgabe aus. Besonders bei der Eingabe müssen nicht nur die Daten vorhanden sein, sondern es muss auch eindeutig definiert sein, welche Startbedingungen und -ereignisse zur Instanziierung des Prozesses führen.

In der BPMN-Spezifikation [4, Kapitel 10.6.4] lassen sich verschiedene Möglichkeiten finden, wie BPMN-Prozesse instanziiert werden können. In den meisten Fällen besitzen BPMN-Prozesse nur ein ausgezeichnetes Starterereignis, welche den Eintrittspunkt einer neuen Prozessinstanz in den Prozess repräsentiert. Allerdings gibt es auch andere Szenarien, bei welchem mehrere Starterereignisse zueinander in Beziehung stehen und gemeinsam die Instanziierung eines Prozesses einleiten, wie beispielsweise die exklusive oder parallele Instanziierung.

Neben der einfachen direkten Prozessinstanziierung werden in der Wirtschaft auch letztere Szenarien umgesetzt. In [3, Kapitel 4] wird beispielsweise auf Teile des SAP-Referenzmodells verwiesen, welches zwar als EPK² beschrieben ist, jedoch reale, ausführbare Szenarien für die Prozessinstanziierung beschreibt. Im Buch über die Verwendung der BPMN in der Praxis [1] beschreiben die Autoren, dass die Instanziierung durch mehrere Starterereignisse ein reales Anwendungsszenario bei ihren Kundenprojekten darstellt.

2.1.2 Event-Subscription bei Prozessausführung

Während der Ausführung eines Prozesses können Zwischenereignisse auftreten und dazu genutzt werden, um die Prozessausführung zu verzögern, indem beispielsweise auf den Eingang einer Nachricht gewartet wird [5].

In der BPMN werden die Zwischenereignisse in eintretende und auslösende Zwischenereignisse gegliedert [1, 4], wobei sich diese Bachelorarbeit auf eingehende Zwischenereignisse konzentriert. Zwischenereignisse können in BPMN-Prozessen sehr vielfältig verwendet werden. Neben den vermeintlich einfachen Szenarien, in denen nur auf ein Ereignis gewartet wird, gibt es auch komplexere Anwendungsszenarien, in denen Zwischenereignisse sogar den Kontrollfluss beeinflussen, wie zum Beispiel das ereignisbasierte Gateway oder angeheftete Zwischenereignisse [4, Kapitel 13.4].

Ein Blick in die Praxis zeigt, dass auch letztere Szenarien im Rahmen einer technischen Prozessrealisierung implementiert wurden. In [6] werden verschiedene Interaktionsmöglichkeiten zwischen mehreren Systemen (Service-Interaction-Pattern) beschrieben, die unter anderem aus realen Industrieszenarien entwickelt wurden. Die Mehrheit dieser Service-Interaction-Patterns beschreiben reale Anwendungsszenari-

²Ereignisgesteuerte Prozesskette, kurz EPK, eine Modellierungssprache für Geschäftsprozesse

en, in welchen Zwischenereignisse eine große Rolle spielen. Eine Prozessengine sollte in der Lage sein, die Szenarien zu unterstützen, da diese Szenarien von der ausführbaren Konformitätsklasse (Process Execution Conformance) der BPMN [4, Kapitel 2.2] vorgegeben werden und diese von den meisten Prozessengines angestrebt werden sollte.

Im Zusammenhang von Zwischenereignissen muss die Frage untersucht werden, ob ein Zwischenereignis nur dann von der Prozessengine verarbeitet werden kann, wenn die Prozessausführung genau die Stelle im Prozess erreicht hat, in welcher auf das Zwischenereignis gewartet werden soll. Die Ausführungssemantik der BPMN bestätigt an dieser Stelle, dass Ereignisse erst registriert werden können, wenn die Prozessausführung dieses Zwischenereignis erreicht hat [4, Kapitel 13.4.2]. Im Buch von Bernd Rücker und Jakob Freund wird dieser Sachverhalt als „strenge Verpuffungssemantik“ bezeichnet, welche auf technischer Ebene zu großen Problemen führen kann [1, Kapitel 5.4.2]. So führt die „Verpuffungssemantik“ beispielsweise dazu, dass Ereignisse ignoriert werden könnten, die zeitlich vor ihrer Registrierung eintreten, obwohl sie für die Ausführung des Prozesses entscheiden sind, was in einigen Fällen zum Deadlock führen kann.

Im Rahmen der *JodaEngine* finden diese Probleme keine Betrachtung, da die BPMN-Ausführungssemantik zugrunde gelegt wird. Allerdings wird in Kapitel 5 ein Ansatz vorgestellt, der auf das „Verpuffungsproblem“ eingeht.

2.1.3 Gesamtszenario für Event-Subscription

Nachdem gezeigt wurde, wie Ereignisse in einem Prozess verwendet werden, betrachtet dieser Abschnitt nun einen Referenzprozess (siehe Abbildung 1), welcher die Anwendungsfälle aus den vorherigen Abschnitten zusammenfasst und als Orientierung bei der Implementierung der *JodaEngine* genutzt wurde.

Der Referenzprozess behandelt die Bearbeitung des Antrages für eine Krankenversicherung bei der ProKK (siehe [7, Kapitel 2.1]). Dabei wird der Prozess entweder durch den Eingang eines neuen Antrags per Email oder täglich zu einem bestimmten Zeitpunkt gestartet, welches durch das instanziiierende ereignisbasierte Gateway dargestellt wird. Nach der Prozessinstanziiierung lässt sich der Referenzprozess wie folgt zusammenfassen: Zunächst werden die wichtigen Informationen des Antrags

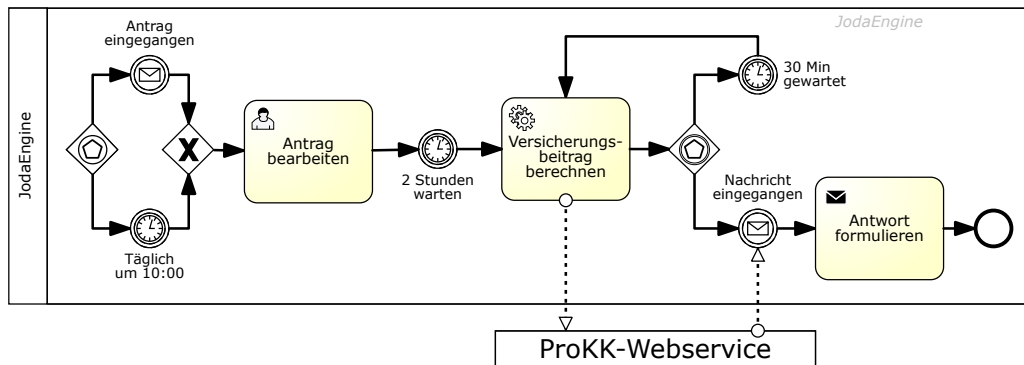


Abbildung 1: Referenzprozess zur Bearbeitung eines Krankenversicherungsantrags

von einem Mitarbeiter der ProKK bearbeitet. Die ProKK arbeitet nach dem Vieraugenprinzip, wodurch der bearbeitete Antrag anschließend zwei Stunden ruhen muss, damit dieser von einem anderen Mitarbeiter geprüft werden kann. Nach der Prüfung wird eine Anfrage an den ProKK-Webservice gesendet, der den Versicherungsbeitrag zu diesem Antrag berechnet. Da es sich um einen asynchronen aber nicht ganz zuverlässigen Webservice handelt, muss auf die Antwort explizit gewartet werden, während ein Timeout sicherstellt, dass die Anfrage nach 30 Minuten erneut gesendet wird. An dieser Stelle wird das ereignisbasierte Gateway verwendet, um das Nachrichten- und Zeitereignis in den Kontrollfluss zu integrieren. Sobald der ProKK-Webservice innerhalb der 30 Minuten antwortet, kann die Bearbeitung des Antrags beendet werden und die Antwort an den Antragsteller zurückgeschickt werden.

Im weiteren Verlauf dieser Bachelorarbeit wird mit diesem Referenzprozess gearbeitet, um Konzepte und Implementierung zu veranschaulichen (vergleiche Kapitel 4.4).

2.2 Anforderungen

Aus den beschriebenen Szenarien ergeben sich die folgenden Anforderungen für die Prozessinstanziierung und das *Event-Subscription Framework*.

Allgemeine Anforderungen an die *JodaEngine*: Da das *Event-Subscription Framework* im Rahmen der *JodaEngine* implementiert wurde, mussten einige

Anforderungen der *JodaEngine*, wie Erweiterbarkeit und Flexibilität, ebenfalls mit berücksichtigt werden. In der Bachelorarbeit von Jannik Streek werden die Anforderungen an die *JodaEngine* detaillierter diskutiert [7, Kapitel 2].

Flexible Prozessinstanziierung: Aus dem Abschnitt bezüglich Szenarien für Startereignisse wird deutlich, dass viele Möglichkeiten zur Prozessinstanziierung existieren. Darüber hinaus wird in Kapitel 3 ein konzeptionelles Framework vorgestellt, das weitere Mechanismen zur Prozessinstanziierung beschreibt. Daher muss die Implementierung der Prozessinstanziierung flexibel und erweiterbar sein, da sie ebenfalls einen zusätzlichen Erweiterungspunkt in der Architektur der *JodaEngine* darstellt (siehe Kapitel 4.1).

Einfache Ereignisregistrierung: Unabhängig von der Art der Ereignisse, müssen im Rahmen der Prozessausführung häufig Ereignisse registriert werden. Daher muss ein leicht verständliches Konzept bereitgestellt werden, mit denen verschiedene Ereignisse definiert und registriert werden können. Hinsichtlich dessen wird in Kapitel 4.3 das *Event-Subscription Framework* vorgestellt.

Beziehungen zwischen Ereignissen: Nicht selten stehen bereits registrierte Ereignisse in einer gewissen Relation zueinander, wie im Fall des ereignisbasierten Gateways im Referenzprozess. Tritt eines der beiden Ereignisse ein, die mit dem Gateway verbunden sind, so muss das andere Ereignis deregistriert werden. Daher muss eine Möglichkeit angeboten werden, um unterschiedliche Relationen zwischen mehreren Ereignissen zu definieren und das Verhalten dieser Relationen zu implementieren (vergleiche Kapitel 4.3).

Korrekte Reihenfolge konkurrierender Ereignisse: Stehen mehrere Ereignisse in einer Relation zueinander, dann ist die Reihenfolge entscheidend, in der diese eintreten. Angenommen sie treten kurz nacheinander ein, dann konkurrieren die Ereignisse darum, dass sie von der Prozessengine bearbeitet werden. Somit befinden sich die Bearbeitungsstränge der Ereignisse in einem kritischen Wettlauf (Race-Condition), in welchem es möglich ist, dass ein späteres Ereignis ein früheres überholt. Dadurch wäre die Reihenfolge verändert. Das *Event-Subscription Framework* muss jedoch sicherstellen, dass die korrekte Reihenfolge erhalten bleibt. In Kapitel 4.4 wird eine Implementierung dieser Anforderung erläutert.

3 Verwandte Arbeiten

Die Verwendung und Bearbeitung von Ereignissen (Event-Processing) ist ein wichtiger Forschungs- und Anwendungsaspekt in der Informatik. Die Bearbeitung von eingescannten RFID-Tags³ [8], die an verschiedenen Waren angeheftet sind, um diese im Verlauf der Wertschöpfungskette verfolgen zu können, oder eine ereignisbasierte Unternehmensarchitektur [2] sind nur zwei aus vielen Anwendungsdomänen, in welchem das Event-Processing eine große Rolle spielt.

Allerdings ist diese Verwendung und Bearbeitung von Ereignissen innerhalb der meisten Prozessengines nur schwach integriert, obwohl in der Praxis durchaus ausführbare Geschäftsprozesse existieren (siehe Kapitel 2.1), die den Umgang mit Ereignissen bei einer Prozessengine voraussetzen. Der Grund dafür ist, dass Prozessengines zu technisch festlegen müssten, inwiefern Ereignisse eingehen [1]. Dies kann allerdings auf zu vielen verschiedenen Wegen passieren, wie beispielsweise JMS⁴, Webservices, E-Mails oder auch durch eingescannte RFID-Tags. Da Prozessengines, wie Activiti oder jBPM, nicht zu viele Annahmen über ihre Ausführungsumgebung treffen wollen, wird der Umgang mit Ereignissen auf externe Anwendungen ausgelagert, aber somit auch von der Prozessengine entkoppelt [1]. Tobias Pfeiffer untersucht in seiner Bachelorarbeit [9] die Unterstützung von Ereignissen am Beispiel von Activiti und kommt zum Schluss, dass Activiti hierbei wenige Szenarien unterstützen kann. Im Rahmen des Activiti-Projektes wurde daher das PSI-Framework (Process Service Invocation) ins Leben gerufen, welches als externe Anwendung eingehende Nachrichten verarbeiten und mit Prozessinstanzen korrelieren kann [10]. Sobald die Entwicklung des PSI-Frameworks abgeschlossen ist, wird die Activiti-Engine in der Lage sein, Nachrichten-Ereignisse zu unterstützen.

Ob eine Prozessengine verschiedene Instanziierungsszenarien unterstützen kann (vergleiche Kapitel 2.1.1), ist eng an die Unterstützung von Ereignissen gekoppelt. Werden nur wenige Ereignisse unterstützt, so können auch nur wenige Instanziierungsmechanismen unterstützt werden. Im Folgenden wird nun das konzeptionelle CASU-Framework vorgestellt, welches auf Basis diverser Modellierungssprachen verschiede-

³Radio-Frequency Identification, kurz RFID, ist ein winziger Chip, der Informationen über eine Ware enthält und ausgelesen werden kann

⁴Java Message Service, kurz JMS, ist eine Schnittstelle, um mit nachrichtenorientierten Systemen zu interagieren

ne Instanziierungsmechanismen beschreibt und in Kapitel 4.4.1 für die *JodaEngine* implementiert wird.

CASU-Framework Ereignisse spielen im Rahmen der Prozessinstanziierung eine große Rolle, da sie als Auslöser genau spezifizieren, wann eine Prozessinstanz gestartet wird. Eine klare Instanziierungssemantik von Prozessen ist sehr wichtig, um eine eindeutige Interpretation und Ausführung des Prozessmodells zu gewährleisten. Vergleicht man verschiedene Prozessmodellierungssprachen miteinander, so bemerkt man, dass die Prozessinstanziierung jeweils unterschiedlichen Konzepten und Semantiken folgt.

Die Arbeiten [3][11] von Gero Decker und Jan Mendling beschäftigen sich mit einem konzeptionellen Framework zur Prozessinstanziierung, welches als CASU-Framework bezeichnet wird. Beide haben verschiedene Modellierungssprachen hinsichtlich ihrer Prozessinstanziierung untersucht und dabei allgemeine Instanziierungsmuster identifiziert. Durch diese Instanziierungsmuster ist nun eine Grundlage gegeben, mit welcher sich die einzelnen Modellierungssprachen hinsichtlich ihrer Instanziierungssemantik einteilen und vergleichen lassen.

Das CASU-Framework ist in vier Instanziierungsaspekte gegliedert, die in der Instanziierungssemantik einer Prozessmodellierungssprache enthalten sein müssen. Dabei ist jeder Aspekt erneut in konkretere Muster (Patterns) eingeteilt. Im Folgenden werden die einzelnen Aspekte und ihre konkrete Muster überblicksartig erklärt. Für Beispiele und detaillierte Informationen zu den einzelnen Aspekten und Mustern sei auf beiden Arbeiten zum CASU-Framework verwiesen [3][11]. An dieser Stelle sei vorgemerkt, dass die konkreten Muster (Pattern) in Kapitel 4.4 über ihr Kürzel referenziert werden.

Creation (C): Dieser Instanziierungsaspekt beschäftigt sich mit der Frage, wann eine neue Prozessinstanz erstellt werden soll [3, Kapitel 3.1]. Dabei werden die folgenden konkreten Muster unterschieden:

- *Ignorance* (C-1) beschreibt, dass ein Prozess nicht über Startereignisse oder -bedingungen instanziiert wird. Stattdessen wird die Instanziierung von der Prozessumgebung gesteuert.

- *Single Condition Filter* (C-2) und *Multi Condition Filter* (C-3) spezifizieren, dass ein Prozess durch eine oder mehrere Startbedingungen instanziiert wird. Startbedingungen im Rahmen der *JodaEngine* entsprechen Bedingungsereignissen, die eintreten, sobald die Bedingung wahr ist (siehe [9]).
- *Single Event Trigger* (C-4) und *Multi Event Trigger* (C-5) beschreibt, dass die Prozessinstanziierung anhand von Ereignissen geschieht, die beim Start konsumiert werden.

Activation (A): Sobald eine Prozessinstanz erzeugt wird, muss festgelegt werden, welche Eintrittspunkte in den Prozess aktiviert werden [3, Kapitel 3.2]. Die Prozessinstanz beginnt die Ausführung des Prozesses an den aktivierten Eintrittspunkten. Folgende Muster können dabei unterschieden werden:

- *Initial State* (A-1) sagt aus, dass der initiale Zustand der Prozessinstanz explizit im Prozessmodell definiert wurde.
- Bei *All Start Places* (A-2) ist der initiale Zustand durch alle Startknoten vorgegeben, die anschließend alle bei der Prozessinstanziierung aktiviert werden.
- *True Conditions* (A-3), *Occurred Events* (A-4) und *Occurred Events and Conditions* (A-5) beschreiben, dass nur die Startknoten im Prozessmodell aktiviert werden, bei denen die Startbedingung positiv ausgewertet wurde oder das Startereignis eingetreten ist. Dadurch entsteht eine logische Kopplung zwischen der Instanziierung und Aktivierung.

Subscription (S): Die Ausführungssemantik von manchen Modellierungssprachen erlaubt, dass Startereignisse, die zur Instanziierung noch nicht eingetreten sind, bei der Ausführung eines Prozesses betrachtet werden können. Dieser Instanziierungsaspekt gibt Muster vor, die beschreiben, für welche dieser noch nicht eingetretenen Startereignisse sich die Prozessinstanz registriert [3, Kapitel 3.3]:

- *All Subscriptions* (S-1) gibt vor, dass sich die Prozessinstanz für alle noch nicht eingetretenen Startereignisse registriert.
- *No Subscriptions* (S-2) untersagt, dass sich eine Prozessinstanz nach ihrer Erzeugung für andere Startereignisse registriert. Beispielsweise handelt es

sich hierbei um die Semantik, die von der BPMN vorgegeben ist und daher im Kapitel 4.4.1 implementiert wurde.

- *Reachable Subscriptions* (S-3) spezifiziert, dass nur bestimmte Startereignisse für die Prozessinstanz registriert werden. Es handelt sich dabei um die Startereignisse, die benötigt werden, damit die Prozessinstanz erfolgreich beendet werden kann.

Unsubscription (U): Wie lange Registrierungen der Subscription-Patterns bestehen bleiben, wird in diesen Instanzierungsaspekt behandelt [3, Kapitel 3.4]. Die Unsubscription bietet Muster an, die spezifizieren, wann die registrierten Ereignisse deregistriert werden können:

- *Until Consumption* (U-1) beschreibt, dass eine Prozessinstanz erst beendet werden kann, sobald die registrierten Ereignisse eingetreten sind. Dies bedeutet, dass sämtliche Registrierungen von Startereignissen gehalten und nicht deaktiviert werden.
- *Until Termination* (U-2) spezifiziert, dass die registrierten Startereignisse deregistriert werden, sobald die Prozessinstanz beendet ist.
- *Timer-based* (U-3) sagt aus, dass die Registrierungen von Ereignissen nach einer bestimmten Zeitspanne aufgehoben werden.
- *Event-based* (U-4) erläutert einen Sachverhalt, in welchem mehrere exklusive Startereignisse registriert sind und eines diese Ereignisse eintritt. Damit werden die anderen Ereignisregistrierungen aufgehoben.
- Bei *Proper Completion* (U-5) können die registrierten Startereignisse verworfen werden, sobald die Prozessinstanz ohne dieses registrierte Ereignis einen gültigen Endzustand erreicht.

Gero Decker und Jan Mendling stellen in [3, 11] eindeutig klar, dass es sich beim CASU-Framework nur um eine konzeptionelle Arbeit handelt. Im Rahmen dieser konzeptionellen Arbeit wurde das CASU-Framework nicht implementiert. Allerdings wird dabei beschrieben, welche Infrastruktur für eine Implementierung des CASU-Frameworks gegeben sein müsste (siehe Abbildung 2, welche aus [3] entnommen wurde).

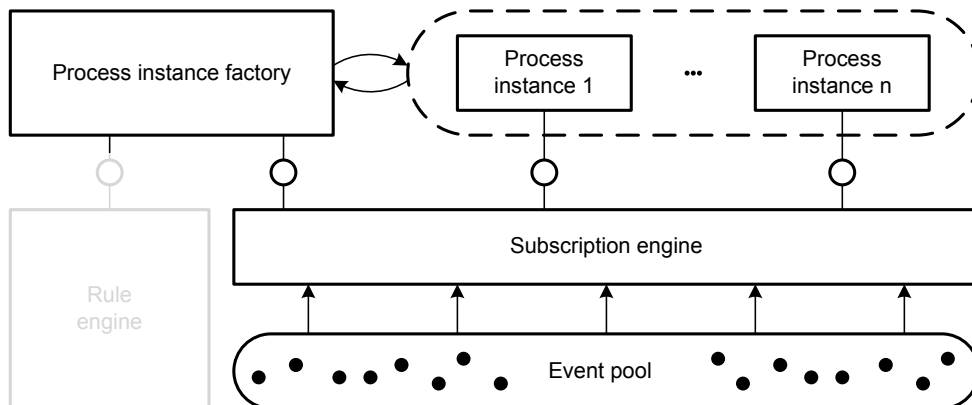


Abbildung 2: Ereignisverarbeitenden Infrastruktur für das CASU-Framework

Die vorausgesetzte Infrastruktur besteht im Wesentlichen aus vier Komponenten. Der Event-Pool erzeugt Ereignisse, welche von der Subscription-Engine beobachtet werden. Die Process-Instance-Factory und die Process-Instances können sich für bestimmte Ereignisse bei der Subscription-Engine registrieren. Die Registrierungen von der Process-Instance-Factory bleiben dabei dauerhaft erhalten, da sie nicht an eine konkrete Prozessinstanz gebunden sind. Wie aus dem Namen hervorgeht, erzeugt die Process-Instance-Factory neue Process-Instances, sobald Ereignisse eintreten. Erst nach dieser Instanziierung kann sich eine Process-Instance für bestimmte Ereignisse registrieren. Sobald ein Ereignis eintritt, überprüft die Subscription-Engine, ob dieses bereits registriert ist und benachrichtigt daraufhin die Komponente, die sich auf das Ereignis registriert hat. Die Rule-Engine, die Regeln im Rahmen der Instanziierung auswertet, bleibt von der Ereignisregistrierung unberührt.

Zusammenfassend stellt das CASU-Framework bis dato ein Konzept hinsichtlich der Prozessinstanziierung bereit, welches aus einer Analyse der verschiedenen Prozessmodellierungssprachen entwickelt wurde. Im Rahmen dieser Arbeit wird die technische Realisierung des CASU-Frameworks untersucht.

4 Implementierung

Ein Hauptbestandteil dieser Bachelorarbeit besteht im Design und der Implementierung des *Event-Subscription Frameworks*. Im folgenden Abschnitt wird kurz erläutert, wie sich das *Event-Subscription Framework* in die Gesamtarchitektur der *JodaEngine* integriert. Im folgenden Kapitel wird zunächst ein näherer Blick auf die Prozessinstanziierung in der *JodaEngine* geworfen, da es die Grundlage für die Implementierung des CASU-Frameworks darstellt. Anschließend wird das *Event-Subscription Framework* vorgestellt, und wie sich dieses in das Konzept des Eventmanagers integriert. Abschließend wird gezeigt, wie Prozessinstanziierung und *Event-Subscription Frameworks* zusammenarbeiten, um den Referenzprozess aus Kapitel 2.1.3 zu realisieren.

Architektur der *JodaEngine*. Die *JodaEngine* besteht, wie in Abbildung 3 und in [7] dargestellt, aus fünf Kerndiensten sowie einem Erweiterungsdienst (`ExtensionService`), mit welchem zusätzliche Dienste, wie ein `DebuggerService` (siehe Bachelorarbeit von Jan Rehwaldt [12]) integriert werden können.

Für die Instanziierung eines Prozesses und die Integration des *Event-Subscription Frameworks* spielen jedoch nur zwei Komponenten eine wichtige Rolle, der `Navigator-` und `EventService`. Der `IdentityService`, `WorklistService` (vgl. Bachelorarbeit von Tobias Metzke [13]) und `ExtensionService` werden im Rahmen dieser Arbeit nicht behandelt.

Der `RepositoryService` verwaltet die Prozessdefinitionen, die von der *JodaEngine* ausgeführt werden sollen. Eine Prozessdefinition repräsentiert einen Geschäftsprozess und muss zunächst deployt werden, wie in der Bachelorarbeit von Thorben Lindhauer beschrieben [14]. Anschließend kann man über den `RepositoryService` die Prozessdefinition aktivieren, sodass diese bereit zur Instanziierung und Ausführung ist. Im Zuge dieser Aktivierung werden ebenfalls die Starterereignisse des Prozesses registriert.

Der `NavigatorService` ist für die Ausführung von Prozessen verantwortlich. Dabei navigiert er durch den Prozess und führt die einzelnen Aktivitäten aus. Auf abstrakter Ebene kann man sagen, dass sich der `NavigatorService` auf Ereignisse registriert, sobald eine Ereignisaktivität, wie z.B. das Zeit-Zwischenereignis in Abbildung 1, von

der Prozessausführung erreicht wurde (für Einzelheiten siehe [14]). Im Rahmen dieser Arbeit wird die Prozessdefinition vorgestellt, die ein Prozess mit allen Informationen repräsentiert und hauptsächlich für die Prozessinstanziierung verantwortlich ist.

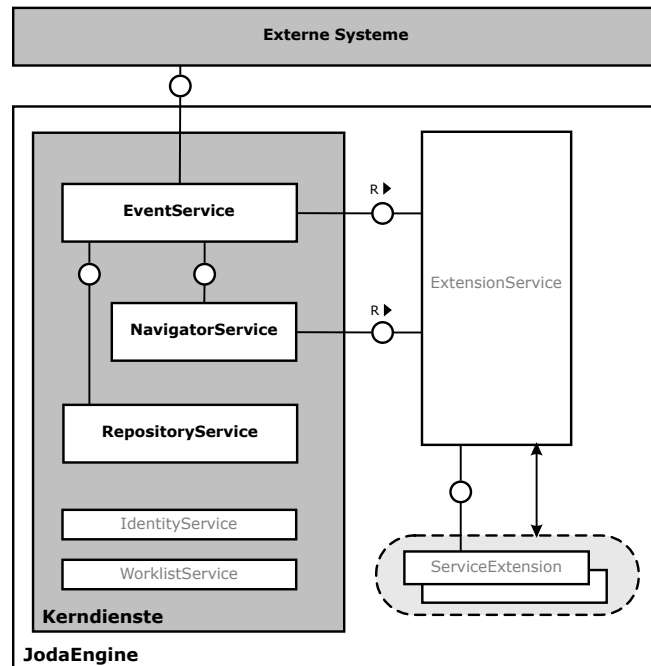


Abbildung 3: Architektur der *JodaEngine*

Der `EventService`, auch Eventmanager genannt, ist ein wichtiger Bestandteil dieser Arbeit, da in ihm das *Event-Subscription Framework* integriert wurde. Daher stellt der Eventmanager die Möglichkeit bereit, sich auf Ereignisse zu registrieren. Weiterhin ist der `EventService` sowohl für die eingehende als auch die ausgehende Kommunikation mit externen Systemen verantwortlich, die in der Bachelorarbeit von Tobias Pfeiffer besondere Beachtung findet [9]. In Kapitel 4.2 wird die Implementierung des Eventmanagers hinsichtlich der Registrierung von Ereignissen näher erläutert.

4.1 Prozessinstanziierung

Im Hinblick auf die Instanziierung des Referenzprozesses in Kapitel 4.4.1, wird in diesem Kapitel das grundlegende Konzept zur Prozessinstanziierung in der *Joda-*

Engine beschrieben. Wie bereits in den Anforderungen in Kapitel 2.2 beschrieben, sollte die Prozessinstanziierung flexibel und erweiterbar sein, damit möglichst viele Mechanismen integriert werden können.

4.1.1 Lebenszyklus einer Prozessdefinition

Beim Deployment eines ausführbaren Geschäftsprozesses [14] wird eine **ProcessDefinition** (Prozessdefinition) erstellt, welche den deployten Prozess repräsentiert und wichtige Informationen für die Ausführung des Prozesses enthält. Bevor die **ProcessDefinition** jedoch instanziiert werden kann, muss sie aktiviert werden.

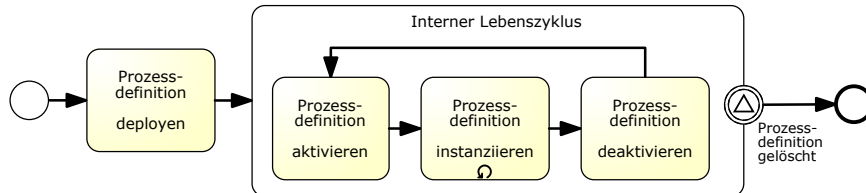


Abbildung 4: Lebenszyklus einer Prozessdefinition

Dies ist im internen Lebenszyklus einer Prozessdefinition definiert, welcher sich in drei Phasen einteilen lässt. Abbildung 4 zeigt ein BPMN-Modell der einzelnen Phasen im gesamten Lebenszyklus einer Prozessdefinition. Nachdem ein Prozess deployt wurde, muss die neu erzeugte Prozessdefinition aktiviert werden. Anschließend können solange beliebig viele Instanzen erstellt werden bis die Prozessdefinition deaktiviert oder gelöscht wird. Der aufgeklappte Teilprozess zeigt die drei internen Phasen einer Prozessdefinition, die auch als Methoden von jeder **ProcessDefinition** für Benutzer (Clients) bereitgestellt werden müssen.

ProcessDefinition.activate(): Der Benutzer kann hierüber die Prozessdefinition aktivieren. Die Unterscheidung zwischen aktivierten und deaktivierten Prozessdefinitionen ist durchaus sinnvoll, da der Prozessadministrator zur Laufzeit der *JodaEngine* die Kontrolle darüber haben muss, welche deployten Prozesse gestartet werden können und welche nicht. Die meisten Prozessengines bieten nur die Möglichkeit zum Löschen einer Prozessdefinition und nicht zur Prozessaktivierung an, weil die Prozessinstanziierung von externen Anwendungen

angestoßen wird und diese gegebenenfalls ausgeschaltet werden können. Da jedoch die *JodaEngine* in der Lage sein soll, die Prozessdefinition selbstständig durch definierte Startereignisse zu instanzieren, muss eine Möglichkeit bereitgestellt werden, die Prozessinstanziierung zeitweise zu unterbinden, ohne den deployten Prozess zu löschen.

ProcessDefinition.createInstance(): Mit dieser Methode können Benutzer eine Instanz des deployten Prozesses erzeugen. Da sich die verschiedenen Modellierungssprachen hinsichtlich der Instanziierungssemantik unterscheiden, wird es für jede konkrete `ProcessDefinition`, wie der `BpmnProcessDefinition`, eine unterschiedliche Implementierung dieser Methode geben. Im folgenden Abschnitt wird eine Technik erläutert, mit der die Instanziierungslogik unabhängig von einer konkreten `ProcessDefinition` ist und daher wiederverwendet werden kann.

ProcessDefinition.deactivate(): Die Deaktivierung eines deployten Prozesses erfolgt über diese Methode (vgl. `activate()`-Methode).

Diese drei Methoden sind besonders wichtig, da sich die unterschiedlichen Modellierungssprachen hinsichtlich dieser drei Phasen unterscheiden und daher auch unterschiedlich realisiert werden müssen. Beispielsweise müssen in BPMN-Prozessen im Rahmen der Aktivierung die einzelnen Startereignisse registriert werden, während Petrinetze keine konkrete Logik bei der Aktivierung ausführen müssen. Besonders hinsichtlich der Prozessinstanziierung durch verschiedene Startereignisse ist es wichtig, dass die Prozessdefinition selbst die verschiedenen Startereignisse registriert und weitere Schritte hinsichtlich der Prozessinstanziierung ausführt. Die drei Methoden sind in der Schnittstelle `ProcessDefinitionInside` zusammengefasst, welche von konkreten `ProcessDefinitions` realisiert wird.

4.1.2 Implementierung

Die Implementierung des Lebenszyklus geschieht zunächst in der Klasse `AbstractProcessDefinition`, welche in Abbildung 5 näher dargestellt ist.

Die Klasse `AbstractProcessDefinition` stellt eine grundlegende Implementierung bereit, auf welcher die konkreten Klassen der unterschiedlichen Prozessdefinitionen aufbauen können. Die Klasse besitzt eine Liste mit Verweisen auf die einzelnen Startknoten des Prozesses, die ausreichend sind, um den ganzen Prozessgraphen zu refe-

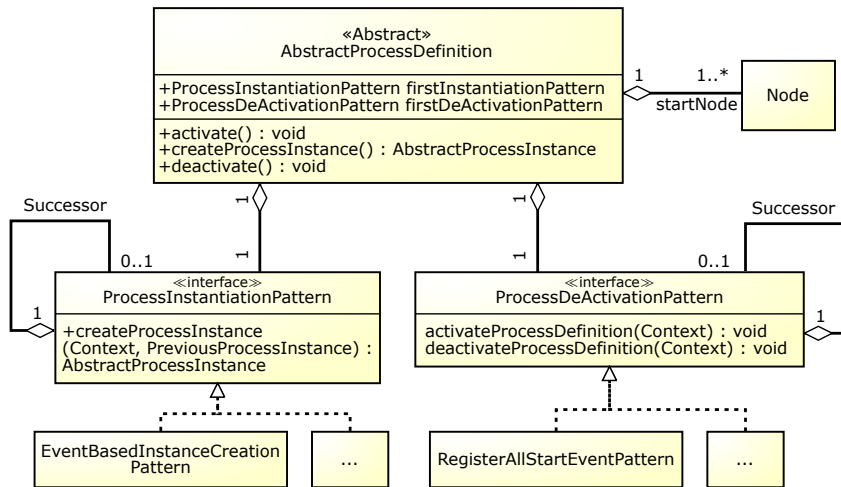


Abbildung 5: Implementierung der Prozessdefinition

renzieren. In den Arbeiten [7] und [14] ist die erwähnte Repräsentation eines Prozessgraphen und dessen Implementierung näher beschrieben.

Außerdem werden auch die drei Methoden der Schnittstelle `ProcessDefinitionInside` in der Klasse `AbstractProcessDefinition` implementiert. Dabei wird auf eine abgewandelte Implementierung des Entwurfsmuster „Chain of Responsibility“ und dem „Command Pattern“ [15] zurückgegriffen. Das „Chain of Responsibility“-Pattern sieht eine verkettete Liste von Verhaltensobjekten vor, die eine Anfrage (z.B. `activate()` oder `createInstance()`) bearbeiten können. Wie in Abbildung 5 erkennbar, handelt es sich bei den Verhaltensobjekten jeweils um die `ProcessDeActivationPatterns` und `ProcessInstantiationPatterns`, die in separaten Ketten enthalten sind.

Die Schnittstelle `ProcessDeActivationPattern` repräsentiert ein Verhaltensobjekt, welches für die Aktivierung und Deaktivierung der Prozessdefinition verantwortlich ist. Den zwei Methoden der Schnittstelle wird als Parameter ein Kontextobjekt (aus dem „Chain of Responsibility“-Pattern) übergeben, das Verweise auf alle notwendigen Informationen und Dienste enthält. Zusätzlich hält jedes Pattern ein Verweis auf seinen Nachfolger, wie es im Konzept der „Chain of Responsibility“ vorgegeben ist.

Die Schnittstelle `ProcessInstantiationPattern` (Instanziierungsmuster) behandelt die Instanziierung der Prozessdefinition. Dazu musste die Implementierung der „Chain

of Responsibility“ abgewandelt werden, da das Entwurfsmuster empfiehlt, dass die Methoden der Schnittstelle keinen Rückgabewert besitzen. In dieser Implementierung gibt jedoch die Methode `createProcessInstance()` eine Prozessinstanz zurück und bekommt als Parameter neben dem Kontextobjekt auch die Prozessinstanz, die das vorherige Pattern in der Kette zurückgegeben hat.

Sobald eine Prozessinstanz erzeugt werden soll, ruft die `AbstractProcessDefinition` auf dem ersten Instanzierungsmuster in der Kette die Methode `createProcessInstance()` auf. Dieses erzeugt eine neue Prozessinstanz, welche dann rekursiv dem nächsten Instanzierungsmuster weitergegeben wird. Nachdem die Prozessinstanz erzeugt worden ist, können die folgenden `ProcessInstantiationPatterns` zusätzliche Aufgaben übernehmen, wie das Registrieren von zukünftigen Ereignissen. Da die Reihenfolge der einzelnen Patterns entscheidend ist, muss dies vom Programmierer einer konkreten `AbstractProcessDefinition` festgelegt werden. Für die Prozessaktivierung bzw. -deaktivierung wird die gleiche Herangehensweise bei der Implementierung verwendet.

Die Verwendung des abgewandelten Entwurfsmusters „Chain of Responsibility“ hat zur Folge, dass die Implementierung der Prozessaktivierung, -instanziierung und -deaktivierung erweiterbar ist, da nun weitere Patterns leicht hinzugefügt werden können. Außerdem sind die einzelnen Patterns für eine bestimmte Aufgabe verantwortlich, wodurch sie besser gekapselt sind, daher die Kopplung reduzieren und wiederverwendet werden können. Neben der Instanziierung von Prozessen können die Patterns auch andere Aufgaben übernehmen, wie Persistierung und Loggen der Prozessinstanziierung, wodurch diese Patterns ein zusätzliches Erweiterungskonzept in der *JodaEngine* darstellen.

Nachdem die Klasse `AbstractProcessDefinition` eine grundlegende Implementierung bereitstellt, können nun die konkreten `ProcessDefinitions` einfacher implementiert werden. Hinsichtlich der Prozessaktivierung, -instanziierung und -deaktivierung müssen die konkreten Prozessdefinitionen nur noch festlegen, welche Patterns benutzt werden sollen. Dabei können sie auf eine Sammlung von verschiedenen Patterns zurückgreifen.

4.2 Event-Subscription im Eventmanager

Im Kapitel 4 wurde bereits erwähnt, dass die *JodaEngine* eine eigenständige Komponente für die Verwendung von Ereignissen bei der Prozessausführung besitzt. Diese Komponente wird als Eventmanager oder `EventService` bezeichnet und soll in diesem Abschnitt näher vorgestellt werden.

4.2.1 Konzept

Abbildung 6 zeigt einen Überblick des Eventmanagers in der *JodaEngine*, dessen Konzept sich hinsichtlich der *Event-Subscription* in drei Elemente einteilt.

Das `EventSubscriptionManagement` ist die zentrale Anlaufstelle für die Prozessausführung (`NavigatorService`), um Ereignisse zu registrieren (`subscribe`) oder zu deregistrieren (`unsubscribe`). Dieser Teilbereich des Eventmanagers verarbeitet die Subscription (Registrierung) von Ereignissen und speichert diese in einen `ProcessEventStore` (Prozessereignisspeicher) ab, damit sie später vom `EventCorrelator` abgerufen werden können. Damit sich die Prozessausführung für ein Ereignis registrieren kann, übergibt sie dabei dem `EventSubscriptionManagement` ein konfiguriertes `ProcessEvent` (Prozessereignis), welches ein bestimmtes Ereignis repräsentiert, auf dessen Eintreten die Pro-

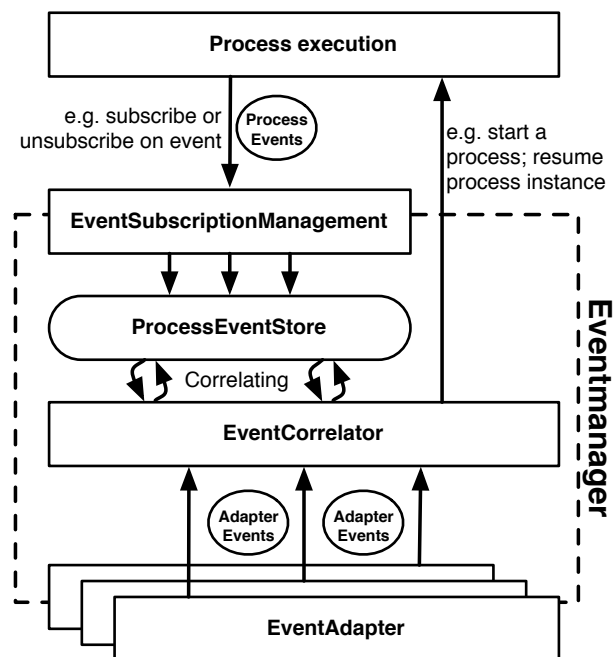


Abbildung 6: Konzept des Eventmanagers

zessausführung wartet. `ProcessEvents` werden vom *Event-Subscription* Framework bereitgestellt und werden im Abschnitt 4.3 näher beleuchtet.

Die `EventAdapter` sind für das Generieren von Ereignissen aus der Umgebung (z.B. externe IT-Systeme) zuständig. Dabei hat ein `EventAdapter` die Aufgabe eines Übersetzers, der die Ereignisse aus der Umgebung in sogenannte `AdapterEvents` überführt, mit welchen der Eventmanager der *JodaEngine* weiterarbeiten kann. Ein `AdapterEvent` repräsentiert demnach ein übersetztes Umgebungsereignis, welches ungefiltert alle Informationen der tatsächlichen Begebenheit enthält. Beispielsweise gibt es einen Adapter, der beim Eingehen von neuen E-Mails ein `AdapterEvent` erzeugt. In diesem Fall enthält das erzeugte `AdapterEvent` alle Informationen der eingegangenen Email, wie Absender und Betreff. Des Weiteren können externe Anwendungen einen `EventAdapter` als Schnittstelle benutzen, um eigene Ereignisse in der *JodaEngine* auszulösen.

Dabei ist zu beachten, dass die erzeugten `AdapterEvents` nicht mit den `ProcessEvents` gleichzusetzen sind, die bei der Registrierung von Ereignissen übergeben werden. Während `AdapterEvents` ein tatsächliches Ereignis aus der Umgebung repräsentieren, werden die `ProcessEvents` von der Prozessausführung genau spezifiziert und beschreiben, auf welches Umgebungsereignis gewartet wird.

Der `EventCorrelator` ist für die Korrelation zwischen `AdapterEvent` und `ProcessEvent` (Prozessereignis) verantwortlich. Dieser greift auf die im `ProcessEventStore` hinterlegten Prozessereignisse zu und vergleicht die erzeugten `AdapterEvents` mit diesen Prozessereignissen. Sobald ein passendes `ProcessEvent` gefunden wird, löst der `EventCorrelator` das `ProcessEvent` aus. Daraufhin kann beispielsweise ein Prozess instanziiert werden oder die ausgesetzte Prozessausführung fortgesetzt werden.

Auf technischer Ebene wird das Konzept des Eventmanagers in gleichnamigen Schnittstellen definiert. Dadurch kann es unterschiedliche Implementierungen geben. Bei allen Implementierungen erhält dabei der `ProcessEventStore` eine besondere Beachtung. Dieser wird in einigen Szenarien sehr beansprucht, da gleichzeitig `ProcessEvents` registriert werden und ständig mit neuen erzeugten `AdapterEvents` korreliert werden müssen. Durch intelligentes Einfügen von `ProcessEvents` in den `ProcessEventStore` und parallelisierte Korrelation kann der Eventmanager effizient implementiert werden.

Wie bereits erwähnt, befasst sich der vorgestellte Überblick mit den Teilbereichen des `EventServices`, die im Rahmen der *Event-Subscription* verwendet werden. Allerdings bietet der `EventService` auch die Möglichkeit an, Ereignisse aus der *JodaEngine* an die Umgebung zu schicken. Das Konzept für ausgehende Ereignisse basiert teilweise auf den vorgestellten Komponenten. Im Rahmen der Bachelorarbeit von Tobias Pfeiffer [9] wird dieser Teil des `EventServices` vollständig vorgestellt.

4.2.2 Implementierung

Nachdem nun das Konzept des Eventmanagers hinsichtlich der *Event-Subscription* vorgestellt wurde, soll nun dessen Implementierung näher erklärt werden. Dabei wird gezeigt, an welchen Stellen in der derzeitigen Implementierung die Schnittstellen des Eventmanagers realisiert sind.

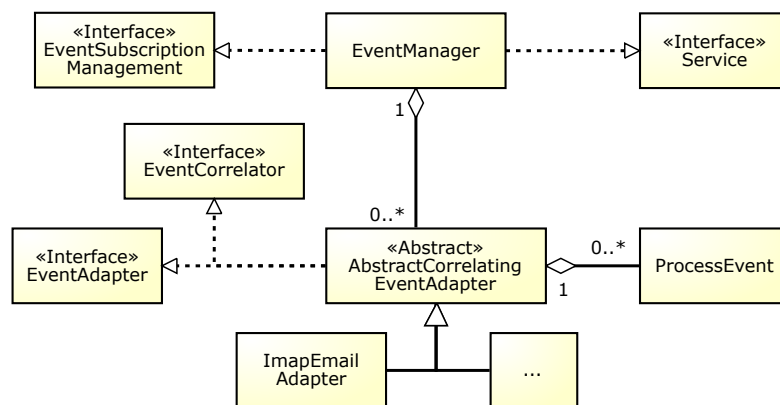


Abbildung 7: Implementierung des Eventmanager-Konzepts

Die Klasse `EventManager` realisiert die Schnittstelle `EventSubscriptionManagement` (siehe Abbildung 7), die als Anlaufstelle für andere Komponenten gilt, um Ereignisse zu registrieren. Dabei ist der `EventManager` ebenfalls als `Service` der *JodaEngine* implementiert, sodass dieser auch von anderen Diensten genutzt werden kann und beim Starten der *JodaEngine* bereitsteht. Des Weiteren ist die Klasse `EventManager` auch für die Verwaltung der verschiedenen `EventAdapter` verantwortlich. Wenn ein Prozessereignis registriert wird, erzeugt der `EventManager` einen passenden Adapter und hält diesen in einer Liste vor, damit für ähnliche Prozessereignisse der gleiche

Adapter verwendet werden kann. Für weitere Informationen hinsichtlich der Verwaltung von Adaptern sei an dieser Stelle auf die Bachelorarbeit von Tobias Pfeiffer [9] verwiesen.

Der `AbstractCorrelatingEventAdapter` ist die grundlegende Implementierung eines Adapters, der sowohl die `EventAdapter`- als auch das `EventCorrelator`-Schnittstelle realisiert, wie in Abbildung 7 dargestellt. Dadurch ist dieser Adapter in der Lage, die erzeugten `AdapterEvents` sogleich selbstständig zu korrelieren. Weiterhin besitzt jeder dieser konkreten `EventAdapter` eine `ProcessEvent`-Liste, welche den `ProcessEventStore` repräsentiert. Sobald ein `AdapterEvent` erzeugt wird, vergleicht dieser Adapter im Rahmen der Korrelation das erzeugte `AdapterEvent` mit den `ProcessEvents` in der Liste. Bei einer Korrelation wird das jeweilige `ProcessEvent` ausgelöst.

Wie bereits erwähnt, spielt der `ProcessEventStore` (Prozessereignisspeicher aus Abbildung 6) eine besondere Rolle im Konzept des Eventmanagers, weil dieser gleichzeitig vom `EventSubscriptionManagement` und dem `EventCorrelator` verwendet wird. Durch die Zusammenführung der `EventAdapter`- und `EventCorrelator`-Schnittstelle in der derzeitigen `EventAdapter`-Implementierung ist es möglich, den Prozessereignisspeicher auf die erstellten `AbstractCorrelatingEventAdapter` zu verteilen. Da nun jeder dieser Adapter eine eigene Liste von `ProcessEvents` enthält, ist die Anzahl der `ProcessEvents`, die bei der Korrelation betrachtet werden müssen, kleiner als bei einem zentralen Prozessereignisspeicher, der alle Prozessereignisse gespeichert hätte. Das bedeutet, dass die passenden Prozessereignisse schneller und effizienter gefunden werden können.

Durch die Verteilung des `ProcessEventStores` kann die Korrelation auf allen Adaptern parallel ausgeführt werden. In einem zentralen `ProcessEventStore` hätten sich andere `EventAdapter`-Implementierungen bei der Korrelation gegenseitig behindert. Daher wären komplizierte Synchronisationsmechanismen nötig gewesen, die jedoch auch die Ausführung verlangsamen.

An dieser Stelle bleibt der Vergleich und die Evaluation mit anderen Implementierungen eines Eventmanagers aus. Allerdings wurde durch diese Implementierung gezeigt, dass das Eventmanager-Konzept nur einen Rahmen definiert, welcher jedoch die Implementierung desselben nicht eingrenzt.

4.3 Event-Subscription Framework

Ein Hauptfokus dieser Bachelorarbeit ist das *Event-Subscription Framework*, welches im Rahmen des Eventmanagers implementiert wurde. Wie bereits in der Einleitung beschrieben, stellt das *Event-Subscription Framework* die Möglichkeit bereit, dass sich Komponenten der *JodaEngine* auf bestimmte Ereignisse registrieren. Weiterhin werden bestimmte Klassen und Schnittstellen angeboten, die diese Registrierung vereinfachen. Das *Event-Subscription Framework* ist dabei in drei Teile eingeteilt:

- Ein `IncomingProcessEvent` repräsentiert ein bestimmtes eintretendes Ereignis, auf dessen Eintreten die Prozessausführung wartet, wie zum Beispiel ein Zeitereignis, das nach einer definierten Zeitspanne eintreten wird. Vom Framework werden mehrere unterschiedliche `IncomingProcessEvents` zur Verfügung gestellt, mit denen die Prozessausführung arbeiten kann. Im vorherigen Kapitel wurde unter anderem das Konzept der `ProcessEvents` erklärt, die im Rahmen des *Event-Subscription Framework* jedoch als `IncomingProcessEvents` bezeichnet werden.
- Eine `ProcessEventGroup` stellt eine Gruppe von `IncomingProcessEvents` dar und kann dazu verwendet werden, um Relationen zwischen mehreren `IncomingProcessEvents` zu definieren. In bestimmten Fällen können mehrere `IncomingProcessEvents` voneinander abhängen. Beispielsweise kann das Eintreten von einem `IncomingProcessEvent` zur Folge haben, dass ein anderes `IncomingProcessEvent` deregistriert werden muss, wie beim ereignisbasierten Gateway (siehe Kapitel 2.1.3). In [2, Kapitel 8] sind weitere Anwendungsfälle beschrieben, in welchen Ereignisse logisch aggregiert werden müssen. Die Beziehungen zwischen `IncomingProcessEvents` können sehr vielfältig sein und erfordern daher das Konzept der `ProcessEventGroup`.
- Das `EventSubscriptionManagement` ist die zentrale Schnittstelle für die Registrierung und die Deregistrierung. Wie bereits in Kapitel 4.2 vorgestellt, ist das `EventSubscriptionManagement` Teil des Eventmanager.

In den folgenden Unterkapiteln werden die einzelnen Teile des *Event-Subscription Frameworks* auch aus technischer Sicht näher beleuchtet.

4.3.1 IncomingProcessEvent

Wie bereits dargestellt, handelt es sich bei den `IncomingProcessEvents` (eintretende Prozessereignisse) um konfigurierte Ereignisse, auf dessen Eingang gewartet werden soll. Das *Event-Subscription Framework* stellt ein Datenmodell in Form von Schnittstellen und Klassen bereit, welche die Verwendung von `IncomingProcessEvents` vereinfachen und eine Grundlage bereitstellt, um eigene `IncomingProcessEvents` zu entwickeln. Abbildung 8 zeigt hierzu die Klassenhierarchie und weitere Implementierungsdetails als UML-Klassendiagramm.

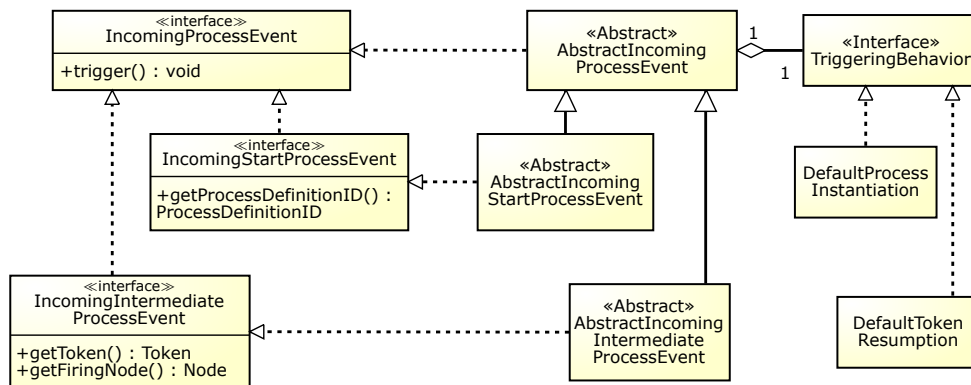


Abbildung 8: Klassenmodell des *Event-Subscription Frameworks*

Die Grundlage des Klassenmodells vom Event-Subscription Framework ist die Schnittstelle `IncomingProcessEvent`, welches als Grundgerüst für alle konkreten, eintretenden Prozessereignisse dient. Dabei besitzt ein `IncomingProcessEvent` eine Adapterkonfiguration für die Spezifikation des benötigten Adapters und eine Bedingungen für die Korrelation (siehe Bachelorarbeit von Tobias Pfeiffer [9]).

Eine wichtige Methode, die von `IncomingProcessEvents` implementiert werden muss, ist die `trigger()`-Methode, die aufgerufen wird, sobald ein eintretendes Prozessereignis korreliert und ausgelöst wird. In der grundlegenden Implementierung der Schnittstelle `IncomingProcessEvent` wird das Konzept der `TriggeringBehavior` (Eintrittsverhalten) eingeführt, die das Verhalten eines ausgelösten bzw. eingetretenen Prozessereignisses repräsentiert und kapselt. Dadurch ist das Eintrittsverhalten unabhängig

von der konkreten `IncomingProcessEvent`-Klasse und kann daher ausgetauscht und wiederverwendet werden.

Im Rahmen der Prozessausführung gibt es zwei Zeitpunkte, in welchen Ereignisse eintreten können. Auf der einen Seite können Startereignisse (`IncomingStartProcessEvents`) verwendet werden, um die Instanziierung eines Prozesses auszulösen. Andererseits gibt es Zwischenereignisse (`IncomingIntermediateProcessEvents`), die während der Ausführung eines Prozesses registriert und eintreten können. Das Klassenmodell des *Event-Subscription Frameworks* macht in der Implementierung von `IncomingProcessEvents` ebenfalls diese Unterscheidung. Dabei werden zwei neue Schnittstellen eingeführt, die spezifische Methoden für den jeweiligen Typ definieren (siehe Abbildung 8).

Ein weiterer Unterschied zwischen beiden Typen von `IncomingProcessEvents` ist die Tatsache, dass `IncomingIntermediateProcessEvents` nur einmal eintreten können. Sobald sie eintreten, werden sie gelöscht und können nicht mehr korreliert werden. `IncomingStartProcessEvents` werden nach dem Eintreten nicht gelöscht und können fortlaufend korreliert werden, solange sie nicht explizit über das `EventSubscriptionManagement` deregistriert werden

Das *Event-Subscription Framework* stellt für alle Schnittstellen bereits grundlegende Implementierungen bereit (vgl. die Realisierung der Schnittstellen in Abbildung 8). Konkrete `IncomingProcessEvents` können auf dieser Implementierung aufbauen, wodurch der Aufwand für die Implementierung von den konkreten `IncomingProcessEvents` möglichst klein gehalten wird.

4.3.2 `ProcessEventGroup`

Im Rahmen der Prozessausführung kann es durchaus vorkommen, dass mehrere `IncomingProcessEvents` unmittelbar zueinander in Beziehung stehen, wie in Kapitel 2.1 beschreiben. Aus diesem Grund wurde das Konzept der Gruppierung von Ereignissen entwickelt und als Bestandteil des *Event-Subscription Frameworks* implementiert. Die Abbildung 9 zeigt, wie sich das Konzept der `ProcessEventGroup` in das Klassenmodell des *Event-Subscription Frameworks* integriert.

Bei der Implementierung der `ProcessEventGroup` wird das Konzept der `TriggeringBehavior` verwendet. Dadurch ist es möglich, das Verhalten der `IncomingProcessEvents`

mit dem Verhalten der `ProcessEventGroup` zu überschreiben. Sobald die `trigger()`-Methode auf einem `IncomingProcessEvent` aufgerufen wird, das zu einer Gruppe gehört, so wird stattdessen die `trigger()`-Methode der `ProcessEventGroup` aufgerufen.

Eine grundlegende Implementierung wird von der Klasse `AbstractProcessEventGroup` bereitgestellt. Die Klasse hält die Verweise auf die `IncomingProcessEvents`, die zur Gruppe gehören, und bietet eine Methode an, mit der man die einzelnen Prozessereignisse zur Ereignisgruppe hinzufügen kann. Weiterhin übernimmt diese Methode das Überschreiben der `TriggeringBehavior` der hinzugefügten Prozessereignisse. Konkrete `ProcessEventGroups`, wie die `ExclusiveIntermediateProcessEventGroup`, können von dieser Klasse erben und selbstständig die `trigger()`-Methode realisieren.

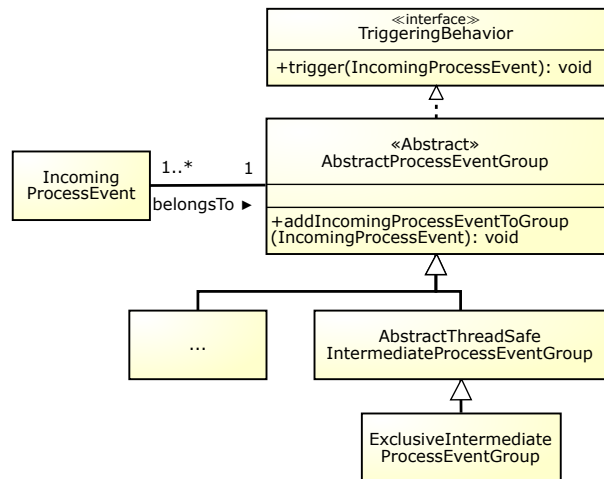


Abbildung 9: Integration der `ProcessEventGroup` ins *Event-Subscription Framework*

Ein Vorteil des `ProcessEventGroup`-Konzeptes ist die einfache Möglichkeit zur Definition von Beziehungen zwischen `IncomingProcessEvents`. Zunächst instanziiert man die gewünschte `ProcessEventGroup` und im zweiten Schritt wird die Methode `addIncomingProcessEventToGroup()` genutzt, um die einzelnen `IncomingProcessEvents` hinzuzufügen. In Kapitel 4.4.2 wird die Verwendung und Implementierung der `ExclusiveIntermediateProcessEventGroup` anhand von Codebeispielen detaillierter erläutert.

4.3.3 EventSubscriptionManagement

Der letzte Bestandteil des *Event-Subscription Frameworks* ist die Schnittstelle `EventSubscriptionManagement` des Eventmanagers bzw. `EventServices`, die bereits in Kapitel 4.2 angesprochen wurde. Diese Schnittstelle ist Teil des Eventmanagers (vgl. Abbildungen 7) und stellt Methoden sowohl für das Registrieren (Subscription) als auch das Deregistrieren (Unsubscription) von `IncomingProcessEvents` bereit.

EventService.subscribeToStartEvent(StartProcessEvent): Durch diese Methode ist es möglich, Startereignisse für einen Prozess zu registrieren. Als Gegenstück wird auch die Methode `unsubscribeFromStartEvent(StartProcessEvent)`, damit Startereignisse auch deregistriert werden können.

EventService.subscribeToIntermediateEvent(IntermediateProcessEvent): Sobald sich die Prozessausführung auf Zwischenereignisse registrieren muss, kann diese Methode verwendet werden. Die Methode `unsubscribeFromIncomingIntermediateEvent(IntermediateProcessEvent)` ist an dieser Stelle das passende Gegenstück.

4.4 Prozessinstanziierung und Event-Subscription Framework in der Praxis

Nachdem in den vorherigen Kapiteln die Implementierung der Prozessinstanziierung und des *Event-Subscription Frameworks* erklärt wurden, soll nun in diesem Kapitel gezeigt werden, wie sich diese beiden Implementierungen in die Prozessausführung integrieren. Dabei soll erläutert werden, wie einzelne Teilaspekte des Szenarios aus Kapitel 2.1.3 realisiert wurden.

Im folgenden Abschnitt wird zunächst die Instanziierung des Referenzprozesses durch die Startereignisse untersucht. Ein Schwerpunkt in diesem Abschnitt behandelt die Implementierung von verschiedenen Patterns des CASU-Frameworks, welches in Kapitel 3 vorgestellt wurde.

Der darauffolgende Abschnitt befasst sich mit den verschiedenen Situationen, in welchen Zwischenereignisse verwendet werden. Besonderes Augenmerk liegt auf der Implementierung des ereignisbasierten Gateways, ein spezieller Kontrollfluss in der

BPMN [4, Kapitel 13.3.4], welches mit Hilfe des *Event-Subscription Frameworks* realisiert werden muss.

4.4.1 Startereignisse bei Prozessinstanziierung

Jede Modellierungssprache für Geschäftsprozesse bietet eine Reihe von unterschiedlichen Methoden zur Prozessinstanziierung an. Mit Hilfe des CASU-Frameworks aus Kapitel 3, existiert nun eine Möglichkeit alle Modellierungssprachen hinsichtlich der Instanziierung in Mechanismen zu unterteilen. Die *JodaEngine* stellt eine technische Grundlage bereit, mit welcher es möglich ist, die unterschiedlichen Mechanismen zur Prozessinstanziierung zu realisieren und zu evaluieren. Im Folgenden soll gezeigt werden, welche Erweiterungspunkte die *JodaEngine* bereitstellt, um das CASU-Framework und die einzelnen Mechanismen zu implementieren.

Zunächst muss jedoch untersucht werden, ob die *JodaEngine* alle Voraussetzungen erfüllt, die vom CASU-Framework an die Infrastruktur gestellt werden. In Abbildung 2 in Kapitel 3 wurde die erforderliche Infrastruktur visualisiert, dessen Elemente im folgenden Text unterstrichen werden. Die Subscription-Engine, die vom Event-Pool mit Ereignissen versorgt wird, entspricht dem Eventmanager der *JodaEngine* mit der Schnittstelle EventSubscriptionManagement. Die als Process-Instance-Factory bezeichnete Komponente ähnelt der zentralen Schnittstelle NavigatorService und letztendlich der Implementierung, welche die Prozessinstanzen in der *JodaEngine* erzeugt, der Prozessdefinition aus Kapitel 4.1. Die *JodaEngine* besitzt auch das Konzept der Process-Instances, welches jedoch in den Bachelorarbeiten [14] und in [7] näher beschrieben wird.

Nachdem gezeigt wurde, dass die *JodaEngine* eine grundlegende Infrastruktur für das CASU-Framework bereitstellt, kann nun auf die konkreten Erweiterungspunkte für die einzelnen Mechanismen eingegangen werden. Wie in Kapitel 3 beschrieben, ist das CASU-Framework in vier Teile gegliedert, die jeweils unterschiedliche Mechanismen (Patterns) bereitstellen: Creation, Activation, Subscription and Unsubscription.

- Die Creation-Patterns beschreiben, wann eine Prozessinstanz erstellt werden soll. In den meisten Fällen muss man sich auf Startereignisse registrieren, die eine Instanziierung auslösen können. Daher spielen die einzelnen Creation-Patterns nur bei Aktivierung der Prozessdefinition eine Rolle und werden somit

als `ProcessDeActivationPatterns` implementiert. Die Creation-Patterns C-2 bis C-5 können im Rahmen der Aktivierung das *Event-Subscription Framework* verwenden, zur Registrierung auf Startereignisse verwendet werden.

- Activation-Patterns definieren, bei welchen Eintrittspunkten des Prozesse eine Prozessinstanz gestartet wird. Sie sind also direkt an der Prozessinstanziierung beteiligt. Hierbei bietet die *JodaEngine* als Erweiterungspunkt die Möglichkeit an, die `ProcessInstantiationPatterns` zu verwenden, um einen bestimmten Mechanismus zur Prozessinstanziierung zu implementieren. Derzeit werden die Activation-Patterns A-1 und A-2 direkt unterstützt und A3 bis A5 mit Hilfe des *Event-Subscription Frameworks*.
- Die Subscription-Patterns legen fest, auf welche der noch nicht benötigten Startereignisse sich registriert werden muss. Aus zeitlicher Sicht werden die `Subscription-Patterns` sofort nach den Activation-Patterns ausgeführt. Da eine Prozessdefinition in der *JodaEngine* die Möglichkeit anbietet, mehrere `ProcessInstantiationPatterns` hintereinander zu schalten, können die Subscription-Patterns als ein weiteres `ProcessInstantiationPattern` implementiert werden. Weil sich die *JodaEngine* zunächst nur auf die Unterstützung von BPMN-Prozessen konzentriert hat und da in der BPMN im Rahmen einer Prozessinstanz keine anderen Startereignisse registriert werden [3, Kapitel 3.5], wurden derzeit auch keine Subscription-Patterns implementiert.
- Die Unsubscription-Patterns definieren, wie lange die Registrierung auf ein Startereignis im Rahmen einer Prozessinstanz erhalten bleibt. Nach gleicher Begründung wie im vorherigen Abschnitt unterstützt die BPMN bei genauer Auslegung des CASU-Frameworks kein Unsubscription-Pattern. Schließt man bei diesen Patterns jedoch auch Zwischenereignisse mit ein, so unterstützt die *JodaEngine* einige Unsubscription-Patterns. Dabei wird die Deregistrierung ausschließlich im *Event-Subscription Framework* gekapselt. Im vorherigen Kapitel wurde gezeigt, wie mit Hilfe der `TriggeringBehavior` verschiedene Eintrittslogiken für Ereignisse implementiert werden können, die auch bestimmte Unsubscription-Patterns unterstützen. Derzeit sind die Unsubscription-Pattern U-3 und U-4 im *Event-Subscription Framework* unterstützt.

Da nun gezeigt wurde, auf welche Art und Weise die verschiedenen Patterns des CASU-Frameworks implementiert werden können, wird nachfolgend die Herange-

hensweise für die Implementierung des Referenzprozesses aus Kapitel 2.1.3 gezeigt. Um den Referenzprozess (siehe Abbildung 1) hinsichtlich der Prozessinstanziierung implementieren zu können, muss zunächst untersucht werden, welche Patterns des CASU-Frameworks für diesen Referenzprozess angewendet werden können. In diesem Fall wurde das Creation-Pattern C-5 ausgewählt, da mehrere Ereignisse den Prozess starten könnten. A-4 wurde als Activation-Pattern ausgewählt, weil nur ein eintretendes Ereignis tatsächlich die Prozessinstanziierung anstoßen kann. Wie bereits im vorherigen Abschnitt beschrieben, unterstützen BPMN-Prozesse kein Unsubscription-Pattern und als Subscription-Pattern nur S-2, sodass beide keinen Einfluss auf die Implementierung haben.

Für das Creation-Pattern C-5 existiert das `ProcessDeActivationPattern` mit dem Namen `RegisterAllStartEventPattern`, welches alle Startereignisse über das `EventSubscriptionManagement` registriert. Das `EventBasedInstanceCreationPattern` ist ein `ProcessInstantiationPattern`, welches das geworfene Ereignis untersucht und den passenden Startknoten ermittelt. Daraufhin wird die Prozessinstanz auf diesem Knoten gestartet.

Abschließend wurde in diesem Kapitel gezeigt, dass die *JodaEngine* hinsichtlich der Prozessinstanziierung eine erweiterbare Grundlage bereitstellt, um andere Mechanismen zur Instanziierung zu realisieren. Es wurden Teile des CASU-Frameworks implementiert und für die fehlenden Patterns Einstiegspunkte in der Implementierung dargestellt. Dadurch ist es theoretisch möglich, das CASU-Framework zu großen Teilen in die *JodaEngine* zu integrieren, wodurch gleichzeitig die Prozessinstanziierung in anderen Prozesssprachen realisiert würde.

4.4.2 Zwischenereignisse bei Prozessausführung

Wie bereits in Kapitel 2.1.2 erwähnt, sind Zwischenereignisse ein wichtiger Bestandteil im Rahmen der Prozessausführung. Durch das ereignisbasierte Gateway ist die Prozessausführung sogar in der Lage, den Kontrollfluss durch Ereignisse zu beeinflussen.

Die Implementierung des ereignisbasierten Gateways beginnt bei der Klasse `BpmnEventBasedXorGateway`, die als konkrete `Activity` implementiert ist und daher Bestandteil des Prozessgraphen ist. Diese `Activity` implementiert das Verhalten des

ereignisbasierten Gateways und wird von der Prozessausführung aufgerufen, sobald diese die Stelle im Prozess erreicht hat, siehe [14] und [7] für genauere Informationen zur Implementierung von BPMN-Konstrukten in der *JodaEngine*.

```
1 ExclusiveIntermediateProcessEventGroup eventXorGroup =
2     new ExclusiveIntermediateProcessEventGroup(token);
3
4 // Folgender Code wird fuer jedes verbundene Zwischenereignisse ausgefuehrt
5 AbstractIncomingIntermediateProcessEvent processEvent =
6     eventBasedGatewayEvent.createProcessIntermediateEvent(token);
7 processEvent.setFiringNode(node);
8
9 eventXorGroup.addIncomingProcessEventToGroup(processEvent);
10
11 eventManager.subscribeToIncomingIntermediateEvent(processEvent);
```

Codebeispiel 1: Implementierung der BpmnEventBasedXorGateway-Activity

Das Codebeispiel 1 zeigt die Verhaltenslogik des `BpmnEventBasedXorGateways`. Nachdem die `ExclusiveIntermediateProcessEventGroup` instanziiert wurde, können nun die `IncomingProcessEvents` aus allen Zwischenereignissen erzeugt werden, die mit dem ereignisbasierten Gateway verbunden sind. Anschließend werden die `IncomingProcessEvents` der Gruppe hinzugefügt (Zeile 9) und letztendlich über den `EventService` registriert (Zeile 11), siehe Kapitel 4.3.

```
1 // Entfernen des eingetretenen IncomingProcessEvents
2 getGroupedEvents().remove(triggeredIncomingProcessEvent);
3
4 for (IncomingIntermediateProcessEvent theGroupedEvent : getGroupedEvents()) {
5     eventManager.
6         unsubscribeFromIncomingIntermediateEvent(theGroupedEvent);
7 }
8
9 token.resume(triggeredIncomingProcessEvent);
```

Codebeispiel 2: `trigger()`-Methode in der `ExclusiveIntermediateProcessEventGroup`

Sobald eines der registrierten Ereignisse eintritt, wird die `trigger()`-Methode aufgerufen (siehe Codebeispiel 2). Da die registrierten `IncomingProcessEvents` zur `ExclusiveIntermediateProcessEventGroup` gehören, wird die `trigger()`-Methode dieser Gruppe stattdessen aufgerufen, die folgendermaßen implementiert ist: Die `ProcessEventGroup` enthält eine Liste der ihr zugehörigen `IncomingProcessEvents`, aus wel-

cher im ersten Schritt das eingetretene `ProcessEvent` entnommen wird. Die übrigen Prozessereignisse werden über den `EventService` deregistriert, da diese nach BPMN-Semantik verworfen werden müssen. Abschließend wird die Ausführung der Prozessinstanz fortgesetzt. Aus dem übergebenen `IncomingProcessEvent` (Zeile 9) ermittelt das `BpmnEventBasedXorGateway` den neuen Pfad für die Prozessinstanz.

Race-Conditions in ProcessEventsGroups. Allerdings können bei dieser Implementierung Probleme hinsichtlich Race-Conditions auftreten, wenn die gruppierten `IncomingProcessEvents` gleichzeitig eintreten. Kritisch ist die Tatsache, dass jedes eingetretene `IncomingProcessEvent` in einem eigenen Thread ausgeführt wird und daher keine Kenntnisse voneinander haben. Treten nun die Prozessereignisse gleichzeitig ein, so könnte es zu einem kritischen Wettlauf zwischen den verschiedenen Threads kommen, welcher in einem inkonsistenten Zustand enden würde.

Für diesen Fall muss die Implementierung den kritischen Wettlauf auflösen und sicherstellen, dass sich die Ausführungen der `IncomingProcessEvents` im Rahmen der `trigger()`-Methode nicht überholen. Die passende Stelle für die Synchronisation ist die Ereignisgruppe, da alle gruppierten `IncomingProcessEvents` die `TriggeringBehavior` der Gruppe verwenden müssen. Daher wurde das *Event-Subscription Framework* um die Klasse `AbstractThreadSafeIntermediateProcessEventGroup` erweitert (siehe Abbildung 9), welchen einen einmaligen Aufruf der `trigger()`-Methode der abgeleiteten `ProcessEventGroup`-Klasse sicherstellt.

```
1 public synchronized void trigger(IncomingProcessEvent processEvent) {
2
3     // Die Instanzvariable called ist initial mit false belegt
4     if (!called) {
5         this.called = true;
6         return;
7     }
8
9     triggerIntern((IncomingIntermediateProcessEvent) processEvent);
10 }
```

Codebeispiel 3: `trigger()`-Methode in der `AbstractThreadSafeIntermediateProcessEventGroup`

Der Code-Abschnitt 3 verdeutlicht die Implementierung der `trigger()`-Methode in der `AbstractThreadSafeIntermediateProcessEventGroup`. Durch das Schlüsselwort

`synchronized` wird von der Java-Umgebung sichergestellt, dass diese Methode nicht gleichzeitig von zwei Threads auf diesem Objekt aufgerufen werden kann. Während ein Thread die Methode ausführt, muss der andere Thread außerhalb der Methode warten. Dem ersten Thread in dieser Methode ist dabei noch erlaubt, die abstrakte interne `trigger()`-Methode aufzurufen, welche von den abgeleiteten Klassen implementiert wird (siehe Codebeispiel 2). Bei diesem ersten Aufruf wird die Instanzvariable `called` verändert, sodass nachfolgenden Threads nicht erlaubt ist, die interne Methode aufzurufen. Da nach dem Aufruf der `trigger()`-Methode die `IncomingIntermediateProcessEvents` automatisch gelöscht werden (vergleiche Abschnitt 4.3.1), entstehen keine Seiteneffekte und der Systemzustand bleibt konsistent. Schließlich wurde dadurch verhindert, dass die `trigger()`-Methode der `ExclusiveIntermediateProcessEventGroup` mehrfach aufgerufen wird und dadurch letztendlich ein inkonsistenter Zustand entsteht.

Um diese Implementierung zu testen, wurden zwei `IncomingProcessEvents` definiert, welche im Abstand von 20 Millisekunden ausgelöst wurden. Währenddessen wurden mit Hilfe von Test-Mock-Ups sichergestellt, dass die `trigger()`-Methode einmal aufgerufen und ein konsistenter Zustand erhalten wurde. Bei kleineren Abständen zwischen den ausgelösten Ereignissen konnte von der Test-Umgebung (Eclipse) die korrekte Reihenfolge nicht sichergestellt werden. In kleineren Benchmarks (600 Durchläufe pro zeitlichen Abstand) wurde ermittelt, dass bei einer Differenz von fünf Millisekunden mit einer Wahrscheinlichkeit von 4,5% die Ereignisse in der falschen Reihenfolge ausgeführt wurden, was für Testfälle nicht praktikabel gewesen wäre. Bei einem zeitlichen Abstand von 20 Millisekunden liegt die Fehlerwahrscheinlichkeit bei 0,5%.

Rückblickend lässt sich zusammenfassen, dass durch die Einführung und Verwendung von `IncomingProcessEvents` und `ProcessEventGroups` komplexe Logik außerhalb der Prozessausführung gekapselt werden konnte, wie es am Beispiel von ereignisbasierten Gateways gezeigt wurde. Durch das *Event-Subscription Framework* wird der Umgang mit Ereignissen bei der Implementierung deutlich vereinfacht, wodurch weitere Konstrukte aus der BPMN und anderen Sprachen realisiert werden könnten.

5 Ausblick

In dieser Arbeit wurde mit dem Eventmanager und *Event-Subscription Framework* eine Grundlage gelegt, welche nun in zukünftigen Arbeiten erweitert werden kann. Neben offensichtlichen Erweiterungsmöglichkeiten, wie die Implementierung von weiteren CASU-Patterns oder neuen `ProcessEventGroups`, können auch neue Szenarien realisiert werden, wie das angeheftete Zwischenereignis [4, Kapitel 13.4.3]. In der Bachelorarbeit von Tobias Pfeiffer [9] werden noch andere Erweiterungsmöglichkeiten für den Eventmanager beschrieben.

Nachfolgend soll nun das Problem der „Verpuffung“ von Ereignissen (siehe Kapitel 2.1.2) näher dargestellt werden. Gleichzeitig werden mögliche Lösungsansätze kurz diskutiert, die in zukünftigen Arbeiten wissenschaftlich evaluiert werden könnten.

Problem der „Verpuffungssemantik“. In Kapitel 2.1.2 wurde bereits die „strenge Verpuffungssemantik“ eingeführt, die von der BPMN-Spezifikation [1][4] vorgeschrieben wird und besagt, dass sich auf Ereignisse erst registriert werden kann, wenn die Prozessausführung die Stelle erreicht hat, die auf ein Ereignis wartet. Würde das Ereignis eintreten, bevor sich die Prozessausführung darauf registrieren kann, so würde es „verpuffen“ und ignoriert werden. Dies ist jedoch keine zufriedenstellende Lösung, weil dadurch die Prozessausführung in bestimmten Fällen in einen Deadlock laufen könnte, wie Abbildung 10 verdeutlicht.

Bei diesem fiktiven Szenario wird eine Anfrage an den asynchronen Webservice aus Kapitel 2.1.3 gesendet. Anschließend wird zunächst noch eine manuelle Aufgabe ausgeführt bis letztendlich auf die Antwort des ProKK-Webservices gewartet wird. Wenn die manuelle Aufgabe zu viel Zeit beansprucht, dann ist die Antwort des Webservices bereits eingetroffen und wird nach BPMN-Semantik ignoriert. Nun würde ein Deadlock entstehen, da die Prozessinstanz für immer auf die Antwort des Webservices warten würde, welche bereits eingetroffen ist.

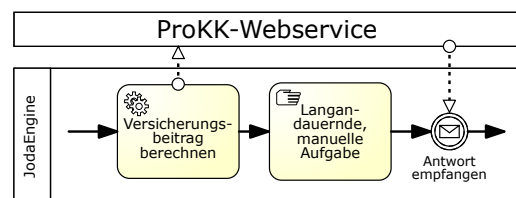


Abbildung 10: Referenzprozess für zukünftige Arbeiten

Daher benötigt man einen Mechanismus, der bereits eingetretene Ereignisse (`AdapterEvents`) abspeichert und diese für die Ausführung von Prozessen vorhält, sodass die Korrelation auch zu einem späteren Zeitpunkt möglich ist. Es ist sehr wahrscheinlich, dass ein solcher Ereignisspeicher, während der Prozessausführung immer mehr `AdapterEvents` vorhalten muss, was dazu führt, dass die Anzahl der zu speichernden `AdapterEvents` schnell ausufern könnte.

Temporärer Ereignisspeicher. Eine mögliche Lösung wäre, dass der Ereignisspeicher nach einer vordefinierten Zeitspanne alle bisher noch nicht korrelierten `AdapterEvents` verwirft. Dabei steht die Zeitspanne in direkter Verbindung mit den Laufzeiten der Prozessinstanzen. Ein `AdapterEvent` braucht nur für die Dauer vorgehalten werden, die die Prozessausführung benötigt, um vom Startpunkt des Prozesses bis zur letzten Ereignisaktivität im Prozess zu navigieren, da nur in dieser Zeitspanne eine Registrierung für dieses `AdapterEvent` erfolgen kann. Beispielsweise konnte nach einigen Durchläufen des Szenarios aus Abbildung 10 ermittelt werden, dass eine Prozessinstanz maximal 60 Minuten benötigt, um nach der Instanziierung die Stelle im Prozess zu erreichen, die auf die Antwort des Webservice wartet. Ein `AdapterEvent`, welches dieses Antwortereignis repräsentiert, könnte nach 60 Minuten verworfen werden, da eine Registrierung in der Mehrheit der Fälle innerhalb der 60 Minuten erfolgt wäre.

Um die erforderliche Zeitspanne für das Verwerfen von `AdapterEvents` zu ermitteln, benötigt man jedoch die Kenntnisse über die Durchlaufzeit einzelner Prozessinstanzen. Auf Grundlage dieser Prozesslaufzeiten könnte die Zeitspanne abgeschätzt oder durch Machine-Learning fortlaufend angepasst werden. Diese Schätzung wäre allerdings eine potentielle Fehlerquelle bei der Prozessausführung, da in Ausnahmefällen die Ausführungsdauer einer Prozessinstanz die Zeitspanne überschreitet und sich die Prozessinstanz somit erneut in einem Deadlock-Zustand befindet.

Zusammenfassend lässt sich sagen, dass der Ansatz des temporären Ereignisspeichers eine Möglichkeit darstellt, das Problem des überfüllten Ereignisspeichers zu begegnen. Allerdings kann dieser Ansatz nur in Szenarien angewendet werden, in welchen ein Fehler- oder Deadlock-Zustand einzelner Prozessinstanzen akzeptiert werden kann. Dieser Ansatz könnte in zukünftigen Arbeiten implementiert und evaluiert werden.

Event-Subscription ohne Ereignisspeicher. `AdapterEvents` müssen aus dem Grund im Ereignisspeicher vorgehalten werden, weil Registrierungen für Ereignisse (`ProcessEvents`) im Verlauf der Prozessausführung erfolgen könnten. Wenn jedoch alle im Prozess verwendeten Ereignisse sofort im Anschluss an die Instanziierung registriert werden, dann könnte auf einen Ereignisspeicher sogar ganz verzichtet werden, denn es ist sichergestellt, dass keine weiteren Registrierungen erfolgen werden.

Allerdings muss jetzt, sobald ein `AdapterEvent` eintritt, bei der Korrelation eine hohe Anzahl an `ProcessEvents` betrachtet werden. Wie in Kapitel 4.2 beschrieben, kann ein großer `ProcessEventStore` zu hohem Speicherverbrauch und Performance-Verlusten führen. Diesem Problem könnte durch die Einführung eines `AdapterEvent-Streams` begegnet werden [16]. Dabei legen die verschiedenen `EventAdapter` ihre erzeugten `AdapterEvents` auf einen Stream, welcher anschließend von mehreren Threads im Rahmen der Korrelation genutzt werden kann und die `AdapterEvents` automatisch verwirft.

Um die Performance bei diesem Ansatz weiter zu verbessern, könnten während der Prozessausführung Ereignisse bereits deregistriert werden, die im weiteren Verlauf der Prozessausführung nicht mehr erreicht werden würden. Hierbei könnten Behavioural-Profiles eingesetzt werden, die verschiedene Aktivitäten und Ereignisse in einem Geschäftsprozess in einem Profil gruppieren und dadurch in Beziehung setzen [17]. Aus bestimmten Behavioural-Profiles könnte man ableiten, dass ein Ereignis zu einem Zeitpunkt in der Prozessausführung nicht mehr erreicht werden kann und daher deregistriert wird.

Im Rahmen dieses Ansatzes muss allerdings noch gelöst werden, wie Ereignisse verarbeitet werden, die erst während der Prozessausführung alle erforderlichen Informationen besitzen. Beispielsweise hat das Ereignis in Abbildung 10 erst zur Laufzeit die Kenntnis darüber, welche bestimmte Antwort erwartet wird. Eine Registrierung zu Beginn der Prozessinstanz wäre nur eingeschränkt möglich. An dieser Stelle könnten Mechanismen für Late-Binding hilfreich sein.

Abschließend lässt sich zusammenfassen, dass mit diesem Ansatz für das Szenario in Abbildung 10 kein Ereignisspeicher benötigt werden würde. Allerdings eröffnen sich neue Fragestellungen hinsichtlich der Performance bei der Korrelation, zu welchen gleichzeitig mögliche Lösungen vorgestellt wurden. Im Rahmen einer zukünftigen Arbeiten könnte dieser Ansatz ebenfalls wissenschaftlich untersucht werden.

6 Zusammenfassung

Im Rahmen dieser Bachelorarbeit wurde die Behandlung von Ereignissen im Verlauf der Prozessausführung untersucht. Dabei wurden komplexe Anwendungsszenarien definiert, welche die Verarbeitung von eintretenden Ereignissen zum Zeitpunkt der Prozessinstanziierung und in der anschließenden Bearbeitung des Prozesses erfordern.

Als technische Lösung wurde das *Event-Subscription Framework* als Teil der *JodaEngine* vorgestellt, mit welchem sich auf Ereignisse während der Prozessausführung registriert und deregistriert werden kann. Das Framework stellt Konzepte und eine technische Grundlage bereit, wodurch es möglich ist, komplexe Anwendungsszenarien hinsichtlich der Registrierung von eintretenden Ereignissen zu unterstützen. Bei der Implementierung eines Referenzprozesses haben sich die vom Framework bereitgestellten Mittel als ausgereift und sehr flexibel herausgestellt.

Da Ereignisse im Rahmen der Prozessinstanziierung eine wichtige Rolle spielen, wurde ebenfalls gezeigt, wie das *Event-Subscription Framework* und die Mechanismen zur Prozessinstanziierung der *JodaEngine* zusammen verwendet werden können, um komplexe Instanziierungsszenarien zu realisieren. Hierbei wurde durch eine Teilimplementierung des konzeptionellen CASU-Frameworks dargestellt, dass es mit der *JodaEngine* theoretisch möglich ist, die Instanziierungssemantik verschiedener Sprachen zu implementieren.

In Zukunft kann die *JodaEngine* in vielen Teilbereichen weiterentwickelt werden. Im Hinblick auf die *Event-Subscription* könnten die im Ausblick vorgeschlagenen Ansätze zur Lösung des „Verpuffungsproblems“ im Rahmen einer Masterarbeit implementiert und wissenschaftlich evaluiert werden.

Letztendlich wurde mit der *JodaEngine* und dem *Event-Subscription Framework* ein weiterer Schritt gemacht, um den Funktionsumfang von Prozessengines zu erweitern und Geschäftsprozessmodelle ohne technische Anpassung sofort in einer Prozessengine auszuführen.

Glossar

Ausführbarer Geschäftsprozess: Ein *ausführbares Geschäftsprozessmodell* beschreibt ein Geschäftsprozessmodell mit einem sehr hohen Grad an Detailinformationen, welches in einer formalen Beschreibungssprache vorliegt. Es umfasst zumeist eine Vielzahl an automatisierbaren Aktivitäten und wird um etliche technische Informationen, wie etwa Bedingungen für einzelne Pfade des Ablaufs, ergänzt. Dadurch wird eine computergestützte Verarbeitung der beschriebenen Aktivitätsfolge ermöglicht.

Geschäftsprozess: Als *Geschäftsprozess* wird eine Folge von wertschöpfenden Aktivitäten bezeichnet, wobei eine beliebige Anzahl von Eingaben eine klar definierte kundenorientierte Ausgabe erzeugt. Nur sich wiederholende Abläufe werden als Geschäftsprozess bezeichnet.

Geschäftsprozessmodell: *Geschäftsprozessmodelle* sind graphische, formale oder mathematische Abbildungen eines realen Prozessablaufs und können in unterschiedlichen Abstraktionsebenen und variierendem Detailgrad vorliegen.

Prozessengine: Mit *Prozessengine* wird eine Software bezeichnet, die ausführbare Geschäftsprozessmodelle verarbeiten kann. Dafür ist sie in der Lage die formal definierten Aktivitäten in korrekter zeitlicher Abfolge und unter Einhaltung etwaiger Bedingungen abzuarbeiten. Pro Ausführung eines Modells wird ein neues Prozessexemplar erzeugt.

Prozessexemplar: Ein *Prozessexemplar*, auch als Prozessinstanz bezeichnet, ist ein Geschäftsprozess während seiner Ausführung. Es umfasst einen, an ein Geschäftsprozessmodell gebundenen, Kontext, welcher Informationen über den aktuellen Abarbeitungsstand des Prozesses enthält.

Thread: Ein *Thread* bezeichnet in der Informatik einen Ausführungsstrang innerhalb einer Anwendung. Pro Anwendung ist mindestens ein Thread vorhanden, die Threads werden parallel abgearbeitet.

Quellenverzeichnis

- [1] FREUND, Jakob ; RÜCKER, Bernd ; HENNINGER, Thomas: *Praxishandbuch BPMN*. Bd. 1. Hanser, 2010
- [2] LUCKHAM, David: *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, 2001
- [3] DECKER, Gero ; MENDLING, Jan: Process Instantiation. In: *Data Knowl. Eng.* (2009)
- [4] OBJECT MANAGEMENT GROUP: Business Process Modeling and Notation v2.0 / Object Management Group (OMG). 2011. – Forschungsbericht
- [5] WESKE, Mathias: *Business Process Management: Concepts, Languages, Architectures*. Springer, 2007 <http://dx.doi.org/10.1007/978-3-540-73522-9>. – ISBN 978-3-540-73521-2
- [6] BARROS, Alistair ; DUMAS, Marlon ; HOFSTEDE, Arthur H.: *Service Interaction Patterns: Towards a Reference Framework for Service-based Business Process Interconnection*. Queensland University of Technology, Brisbane : QUT Technical report, FIT-TR-2005-012, 2005
- [7] STREEK, Jannik: Prozessengine als Plattform: Anforderungen und Architektur / Hasso-Plattner-Institut Potsdam. 2011. – Bachelorarbeit
- [8] ZANG, Chuanzhen ; FAN, Yushun ; LIU, Renjing: Architecture, implementation and application of complex event processing in enterprise information systems based on RFID. In: *Information Systems Frontiers* 10 (2008), Nr. 5, 543–553. <http://dx.doi.org/10.1007/s10796-008-9109-0>
- [9] PFEIFFER, Tobias: Ereignisadapter und Korrelation in Prozessengines / Hasso-Plattner-Institut Potsdam. 2011. – Bachelorarbeit
- [10] CAMUNDA SERVICES GMBH: *PSI-Dokumentation*. <http://fox.camunda.com/doc/#psi>. Version: Juni 2011
- [11] DECKER, Gero ; MENDLING, Jan: Instantiation Semantics for Process Models. In: *InProceedings of the 6th International Conference on Business Process Management (BPM), LNCS 2008*, 2008, S. 164–179

- [12] REHWALDT, Jan: Debugging von Geschäftsprozessen in Prozessengines / Hasso-Plattner-Institut Potsdam. 2011. – Bachelorarbeit
- [13] METZKE, Tobias: Ein flexibles Framework zur Ressourcenverteilung: Design und Implementierung / Hasso-Plattner-Institut Potsdam. 2011. – Bachelorarbeit
- [14] LINDHAUER, Thorben: BPMN-Ausführung in einer Prozessengine: Design und Implementierung / Hasso-Plattner-Institut Potsdam. 2011. – Bachelorarbeit
- [15] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1995. – ISBN 0–201–63361–2
- [16] WU, Eugene ; DIAO, Yanlei ; RIZVI, Shariq: High-performance complex event processing over streams. In: CHAUDHURI, Surajit (Hrsg.) ; HRISTIDIS, Vagelis (Hrsg.) ; POLYZOTIS, Neoklis (Hrsg.): *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, ACM, 2006. – ISBN 1–59593–256–9, 407–418
- [17] WEIDLICH, Matthias ; MENDLING, Jan ; ZIEKOW, Holger: Optimising Complex Event Queries over Business Processes using Behavioural Profiles. In: *Proceedings of the 4th International Workshop on Event-Driven Business Process Management (edBPM'10)*, 2010