**Entwicklerhandbuch**

Dieses Handbuch soll die Funktionen der Event Processing Platform (EPP) und deren Implementierung erklären. Dazu gibt es für die Funktionen der Plattform jeweils einen Abschnitt, in dem erklärt ist, wie diese Funktionen implementiert sind.

Inhalt

[**1.** **Paketstruktur** 2](#_Toc361593323)

[**2.** **Kernklassen** 2](#_Toc361593324)

[**3.** **Import** 2](#_Toc361593325)

[**3.1** **Events** 2](#_Toc361593326)

[**3.2** **Deutscher Wetterdienst und TomTomImporter** 5](#_Toc361593327)

[**3.3** **BPMN** 6](#_Toc361593328)

[**4.** **Processing** 6](#_Toc361593329)

[**4.1** **Event Transformation** 6](#_Toc361593330)

[**4.2** **Event Correlation** 6](#_Toc361593331)

[**5.** **Event Repository** 6](#_Toc361593332)

[**6.** **Queries** 6](#_Toc361593333)

[**6.1** **On-Demand** 6](#_Toc361593334)

[**6.2** **Live** 6](#_Toc361593335)

[**6.3** **BPMN** 6](#_Toc361593336)

[**7.** **Monitoring** 7](#_Toc361593337)

[**7.1** **BPMN** 7](#_Toc361593338)

[**7.2** **Monitoring ohne BPMN** 8](#_Toc361593339)

[**7.3** **Attribute Charts** 10](#_Toc361593340)

[**7.4** **Event Views** 11](#_Toc361593341)

[**7.5** **Notifications** 12](#_Toc361593342)

[**8.** **Export** 15](#_Toc361593343)

[**9.** **Event Producing** 15](#_Toc361593344)

[**9.1** **Simulator** 15](#_Toc361593345)

[**10.** **Sign in** 15](#_Toc361593346)

1. **Paketstruktur**

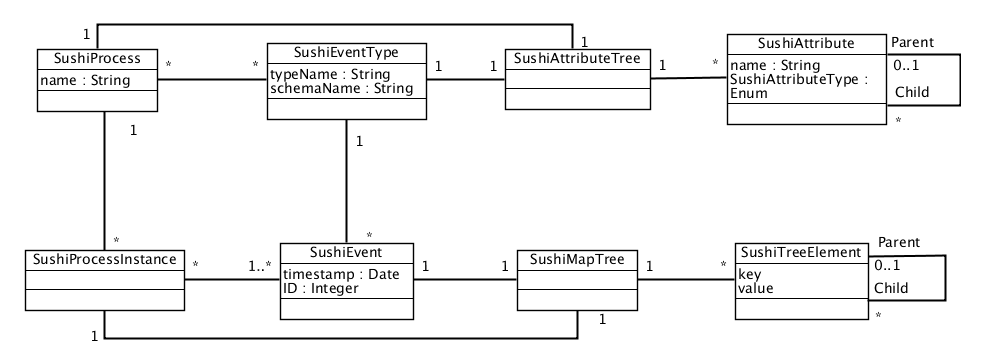
Die Zentrale Komponente in unserer Paketstruktur bildet *SushiCommon*. Hier liegen die Kernklassen (siehe 2.) und die Anbindung an die Datenbank. Um Objekte in unserer relationalen MySQL-Datenbank (<http://www.mysql.com/>) zu speichern nutzen wir zum object-relational mapping eclipseLink (<http://www.eclipse.org/eclipselink/>). *SushiEsper* bildet die Schnittstelle zur Steamingdatenbank Esper. Hier liegt auch der *Broker*, der die Events im System verteilt (siehe 3). Events gelangen über *SushiImport* per Dateiupload, *SushiWebservice* oder per Simulation in *SushiSimulation* ins System. Dabei nutzen sämtliche Pakete die in *SushiCommon* gelegenen Kernklassen. *SushiWicket* bildet das User Interface unserer Anwendung. Hierzu nutzten wir das Webframework Wicket (<http://wicket.apache.org/>). *SushiWicket* greift auf alle anderen Pakete zu, um den Zugriff auf sämtliche Funktionen über die Weboberfläche zu ermöglichen.



1. **Kernklassen**

In unserer Implementierung sind *SushiEvent*, *SushiEventType*, *SushiProcess* und *SushiProcessInstance* von zentraler Bedeutung. Das nachfolgende Klassendiagramm zeigt den Zusammenhang zwischen den einzelnen Klassen. Jedes *SushiEvent* ist mit einem *SushiEventType* verknüpft, genauso wie jede *SushiProcessInstance* mit einem *SushiProcess* verknüpft ist. Die SushiEvents speichern ihre Attributwerte nicht als eigene Attribute, da Events verschiedener Typen verschiedene Attribute haben und Attribute hierarchisch aufgebaut sein können. Zu diesem Zweck referenziert jedes *SushiEvent* einen *SushiMaptree*, der mehrere Wurzelelemente unterstützt. Die Elemente in diesem Baum gehören der Klasse *SushiTreeElement* an und speichern neben Eltern- und Kindelementen einen Schlüssel (Attributname) und einen Wert Attributwert). Die gleiche Beziehung gilt für die Klassen SushiEventType, SushiAttributeTree und SushiAttribute. Der Unterschied liegt darin, dass in *SushiAttribute* statt Attributwerten Werte der Enumeration *SushiAttributeType* gespeichert werden. Die Enumeration enthält die Werte *Integer*, *String* und *Date*.

*SushiProcess* bzw. *SushiProcessInstance* referenzieren ebenfalls einen *SushiAttributeTree* bzw. *SushiMapTree*. Hier sind die Attribute bzw. Attribut-Wert-Paare für die Korrelation gespeichert (siehe 4.2).



1. **Import**
   1. **Events**

Adapter transformieren Raw Events der Eventquellen (Eventlogs im XML, XLS, CSV und EDIFACT (.edi/.txt) Dateiformat) zu den Normalized Events der Plattform (*SushiEvent*) und stellen somit eine Schnittstelle zwischen den Eventquellen und der Plattform dar. Die Adapter haben alle eine ähnliche Funktionalität. Sie erzeugen aus den Daten der Eventquellen Events. Wie der Import und die Verarbeitung von Events mit Adaptern in der EPP umgesetzt ist zeigt nachfolgende Abbildung:



Um Events zu erzeugen, werden die Adapter von außen angestoßen (Schritt 1). Das kann durch einen Benutzer oder automatisiert geschehen. Zum Beispiel kann ein Nutzer auf der Web-Oberfläche der EPP Eventlogs (EventQuelle) hochladen. Die Daten werden anhand des Dateiformat an den jeweils für dieses Dateiformat zuständigen Adapter weitergeleitet. Bei automatisierten Adaptern, wie zum Beispiel dem TomTom-Adapter, ist der Adapter fest mit der Eventquelle verbunden.

Die Adapter lesen die Raw Events aus der Eventquelle aus (Schritt 1.1). Dazu wird eine Datei oder ein Webservices genutzt – je nachdem um was für eine Eventquelle es sich handelt. Die Realisierung unterscheidet die Adapter voneinander. Anhand der Eventquelle oder der Raw Events erkennt der Adapter den Eventtypen (*SushiEventtyp*) der Events (Schritt 2). Zur Erkennung wird das Datenformat und Informationen in den Raw Events genutzt. Falls in der EPP noch kein Eventtyp für die Eventquelle vorliegt, muss zuerst ein Eventtyp angelegt werden. Dies kann automatisch durch den Adapter geschehen, wenn die Events eines Adapters immer den gleichen Eventtypen haben. Ist das nicht der Fall, muss der Benutzer einen Eventtypen anlegen, bevor die Events in die Plattform geladen werden können. In der Abbildung wurde davon ausgegangen, dass der Eventtyp nicht angelegt werden muss.

Wenn der Eventtyp bestimmt ist, kann der Adapter die Raw Events aufschlüsseln (Schritt 3). Jedes Raw Event wird in ein Normalized Event umgewandelt (Schritt 4). Die Quelle für den Zeitstempel – entweder ein Attribut des Events oder die aktuelle Systemzeit – wird im Eventtypen festgehalten. Der Adapter fragt diese Information dann beim Eventtypen ab. Somit ist die Normalisierung von den Raw Events abgeschlossen.

Sobald alle Raw Events in der normalisierten Form vorliegen, können sie zum *Broker* gesendet werden (Schritt 5). Der Broker dient als Event-Verteiler und sendet die Events zur Korrelation zum Korrelator, zur Persistierung an die Datenbank und für die Erstellung von Benachrichtungen an den *NotificationObservable*. Korrelator, Datenbank und *NotificationObservable* sind in der Abbildung aus Gründen der Übersichtlichkeit weggelassen worden. Zuletzt wird das Event zum *Stream Processing Adapter* gesendet (Schritt 5), welcher Esper kapselt. Der *Stream Processing Adapter* wandelt die Events, in XML-Dokumente um (Schritt 5.1.1), um sie anschließend zu Esper zu senden (Schritt 5.1.2).

**XML-Adapter**

Für den XML-import ist der XMLParser zuständig. Er nimmt die XML-Datei entgegen und wandelt sie in ein *SushiEvent* um (*generateEventFromXML*(String filePath)). Damit er das tun kann, ist es erforderlich, dass zuvor ein Eventtyp angelegt wurde, der den Namen trägt, der in dem xsd Namespace der XML-Datei steht (z.B. xsi:noNamespaceSchemaLocation="EventTaxonomy.xsd“). Dafür kann der *XSDParser* genutzt werden. Dieser wandelt eine XSD-Datei in ein Eventtypen um (*generateEventTypeFromXSD*(String filePath, String eventTypeName)). In dem Eventtypen kann in *timestampName* der Pfad zum Timestamp in der XML Datei angegeben werden. Beim Umwandeln der XML-Datei in ein Event wird dieser Pfad genommen und der Timestamp aus diesem XML-Knoten genommen.

Beispiel XML-Event:

<Event xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="TruckEventType.xsd">

<id>1234567</id>

<timestamp>24.12.2013 20:25</timestamp>

<sender>DHL</sender>

<location>

<area>Sachsen</area>

</location>

</Event>

Beispiel XSD-Eventtyp zum XML-Event:

<?xml version="1.0" encoding="UTF-8"?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:element name="TruckEventType">

<xs:complexType>

<xs:sequence>

<xs:element name="id" type="xs:int"/>

<xs:element name="timestamp" type="xs:date"/>

<xs:element name="sender" type="xs:string"/>

<xs:element name="location" type="xs:string">

<xs:complexType>

<xs:element name="area" type="xs:string"/>

</xs:complexType>

</xs:element>

</xs:sequence>

</xs:complexType>

</xs:element>

</xs:schema>

**XLS-Adapter**

Für den XLS-Import ist der *ExcelImporter* zuständig. Er nimmt die XLS-Datei entgegen und wandelt die Datei in (mehrere) *SushiEvents* um. Dafür wandelt er jede Zeile der Excel-Datei in ein *SushiEvent* um. Hauptmethode hierfür ist *importEventsFromFileimportEventsFromFile*(String filePath, List<SushiAttribute>selectedAttributes, String timestamp). Wobei filePath den Pfad zur XLS-Datei ist. SelectedAttributes gibt ab, welche Spalten überhaupt genutzt werden sollen und welchen Typ die Daten bekommen sollen (SelectedAttributes entspricht dem Baum, der im Eventtypen abgelegt ist). Timestamp gibt den Spaltennamen an, aus dem der Timestamp genommen werden soll.

**CSV-Adapter**

Für den CSV-Import ist der *CSVImporter* zuständig. Er nimmt die CSV-Datei entgegen und wandelt die Datei ähnlich zu dem XLS-Importer in *SushiEvents* um. Auch hier wird die Methode *importEventsFromFile*(String filePath, List<SushiAttribute>selectedAttributes, String timestamp) genutzt.

**Edifact-Adapter**

EDIFACT Dateien werden vom *EdifactImporter* entgegen genommen. Die Hauptmethode ist hier *generateEventFromEdifact*(String filePath). Der Edifact-Adapter wandelt die Edifact-Datei in ein XML-Dokument um und sendet sie anschließend zum *XMLParser*. Dieser wandelt das XML-Dokument wiederum in ein *SushiEvent* um.

* 1. **Deutscher Wetterdienst und TomTomImporter**

Die EPP Unterstützt den Import von Wetterwarnungen vom Deutschen Wetterdienst sowie Staumeldungen von TomTom. Damit die beiden Importer genutzt werden können, brauchen sie Zugangsdaten. Für den Deutschen Wetterdienst kann man sich hier registrieren: [Deutscher Wetterdienst FTP](http://www.dwd.de/bvbw/appmanager/bvbw/dwdwwwDesktop?_nfpb=true&_pageLabel=dwdwww_spezielle_nutzer&_state=maximized&_windowLabel=T174800248261285831499722&T174800248261285831499722gsbDocumentPath=Navigation%252FOeffentlichkeit%252FHomepage%252FWetter__Ihre__Website%252Fftp-Zugriff__node.html%253F__nnn%253Dtrue) und für die TomTom hier:  [TomTom API](http://developer.tomtom.com/). Die Zugangsdaten müssen dann in die jeweiligen Klassen *DWDImporter*.java und *TomTomTrafficImporter*.java eingepflegt werden. Es müssen hierbei nicht beide Quellen genutzt werden.

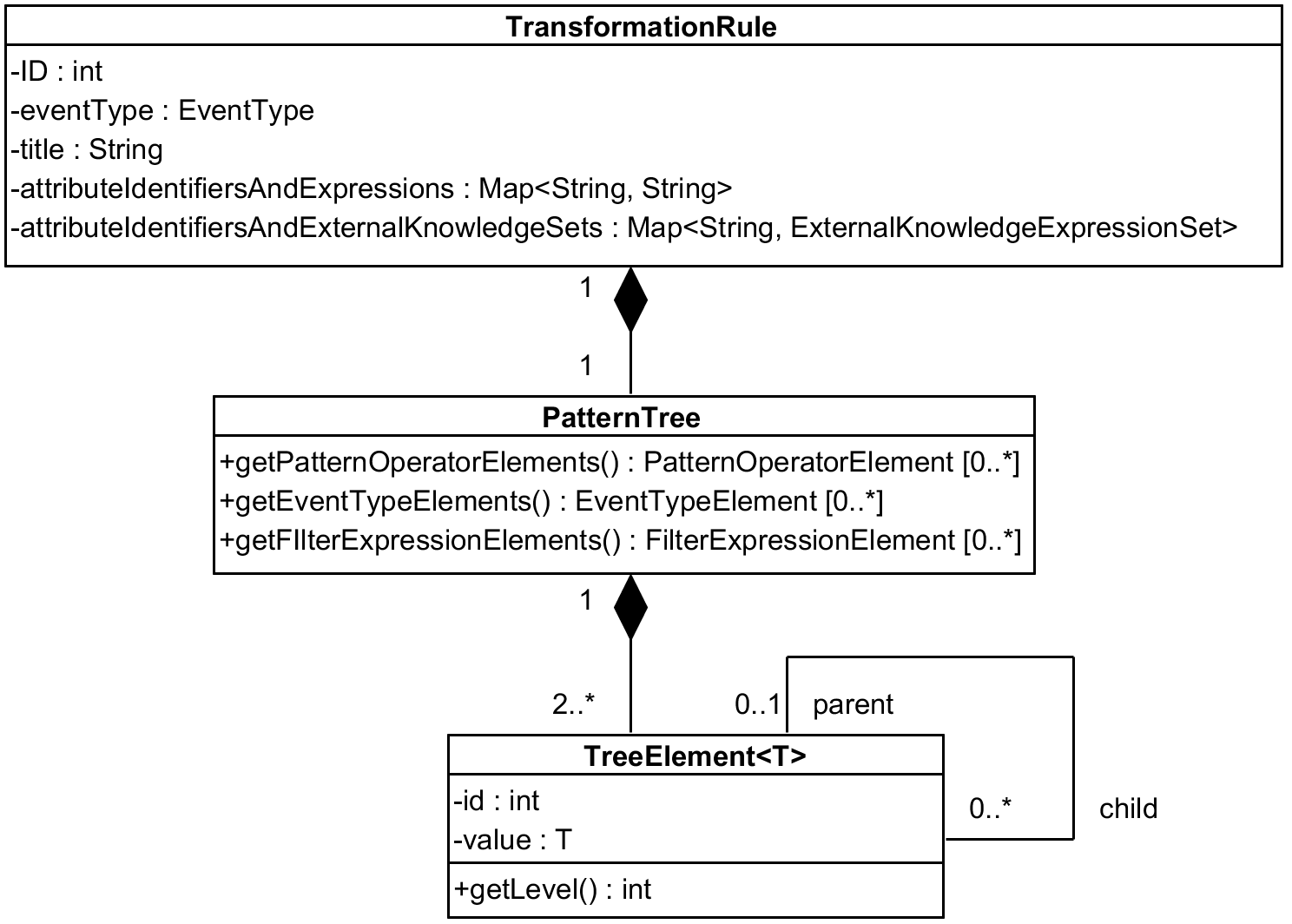
Die EPP nutzt den FTP Server des Deutschen Wetterdienstes um Wetterwarnungen auszulesen und in Events umzuwandeln. Die relevanten Klassen hierzu sind *DWDImporter* und *SushiWeatherAdapter*. Der *SushiWeatherAdapter* ist dabei allerdings lediglich zur Automatisierung des Imports zuständig. Der Importer fragt für die Gebiete, die in Coordinates\_DWD\_Regions.xml beschrieben sind, auf dem Server nach Wetterwarnungen und lädt diese herunter sofern sie noch unbekannt sind. Sollen für andere Gebiete Warnungen heruntergeladen werden muss bei einer Anpassung dieser Datei auch die Methode *syncAllWeatherWarningsExampleUseCase*() angepasst werden. Die Wetterwarnungen sind in XML-Dateiformat und werden an den XML-Parser weitergeleitet.

Die EPP nutzt die API von TomTom (developer.tomtom.com) um Stauwarnungen abzufragen. Hierzu sind die relevanten Klassen *TomTomTrafficImporter* und *SushiTrafficAdapter*. Die zuletzt genannte Klasse dient hierbei lediglich zur Automatisierung. Der *TomTomTrafficImporter* fragt bei TomTom nach Stauevents auf der Strecke zwischen Hamburg und Berlin (dabei werden die Gebiete aus Coordinates\_DWD\_Regions.xml genommen).

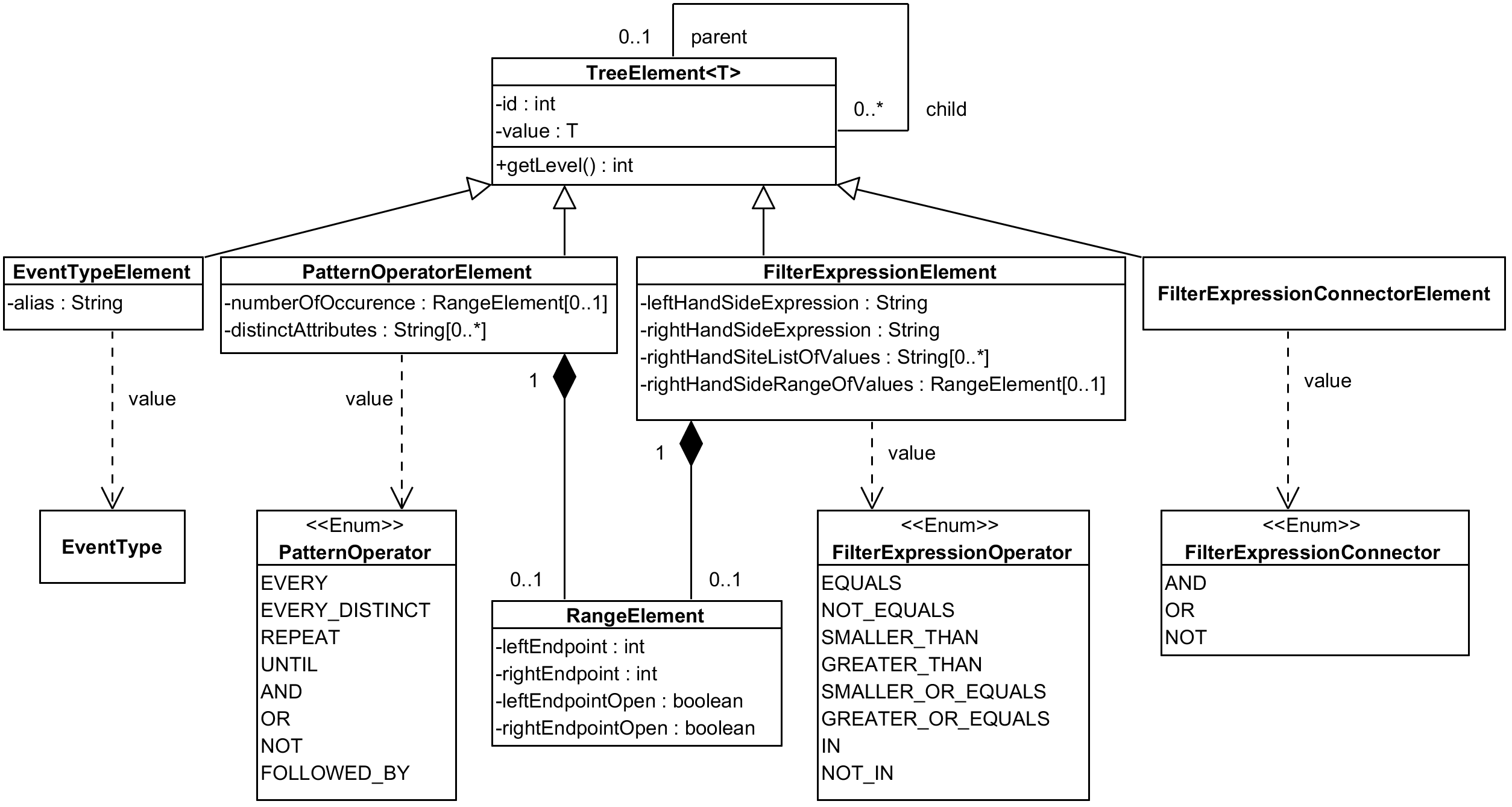
* 1. **BPMN**

Beim Import von BPMN-Modellen wird eine BPMN-XML-Datei (http://www.omg.org/spec/BPMN/2.0/) unterstützt. Die zentrale Klasse hier ist der *BPMNParser*. Dieser nimmt ein XML-Dokument entgegen. Wir wandeln eine Pfadangabe dabei zu erst in ein Document um und lesen daraus die BPMN-Elemente aus. Wir erstellen für jedes BPMN-Element eine logische Repräsentation erstellt. Alle BPMN-Elemente erben dabei von der abstrakten Oberklasse *AbstractBPMNElement***.** Die Elemente werden alle zu einem *BPMNProcess* hinzugefügt. Diese Klasse dient auch als Rückgabe für das Einlesen. Die sichtbare Methode des *BPMNParser* zur Erzeugung des *BPMNProcess* ist **public static** *BPMNProcessgenerateProcessFromXML*(String filePath).

1. **Processing**
   1. **Event Transformation**

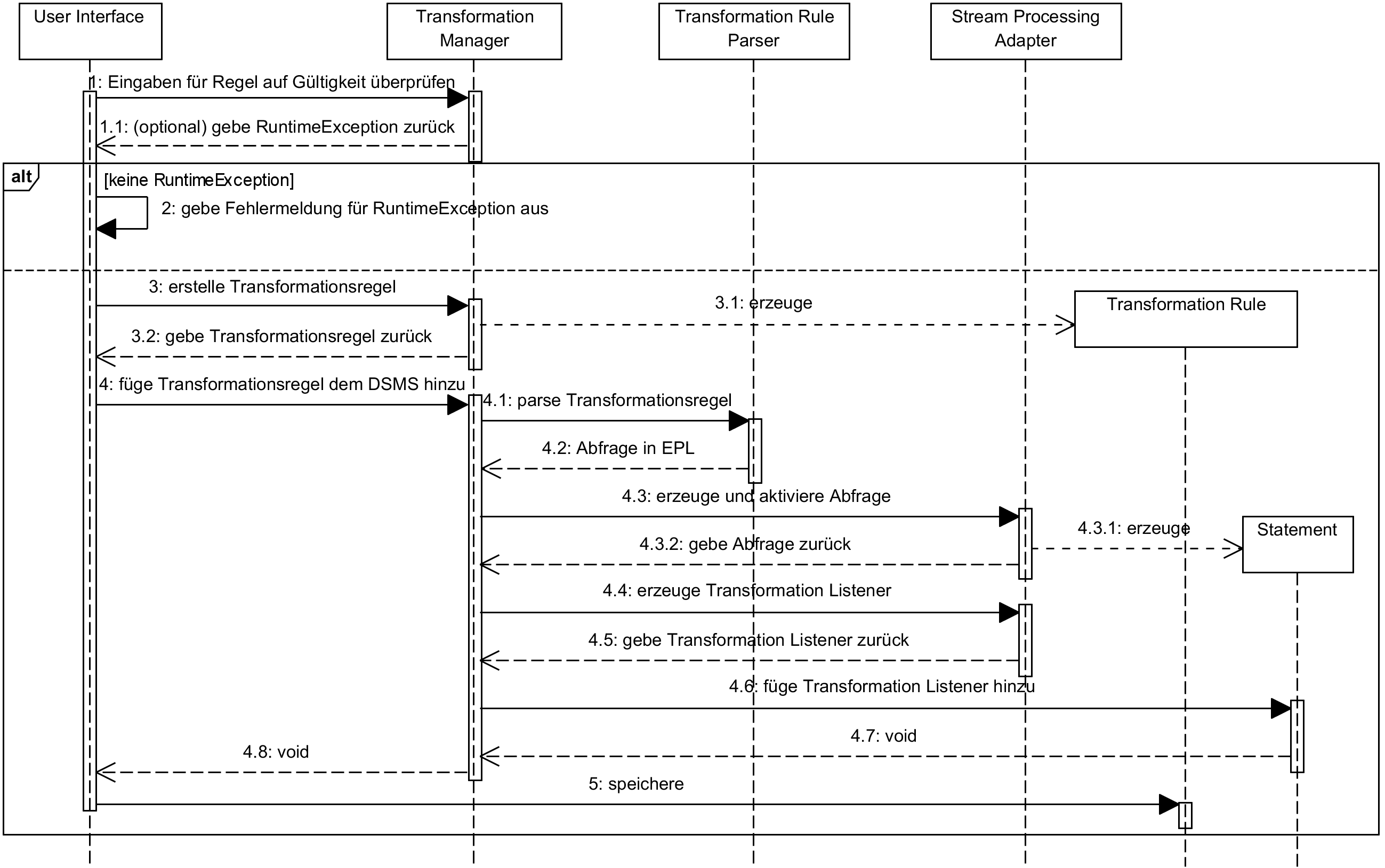


Für die Transformation von Events werden *TransformationRules* als Behälter verwendet. Er hält u.a. einen *SushiPatternTree*, das eine Menge an *SushiTreeElement<Serializable>* enthält, die aufeinander referenzieren wie in einer Baumstruktur. *SushiTreeElement<Serializable>* dient als Oberklasse für die unterstützten Patternelemente (ihre Verwendung wird im Nutzerhandbuch beschrieben), die jeweils einen Wert (value) besitzen, in der Regel den Operator aus einem Enum:



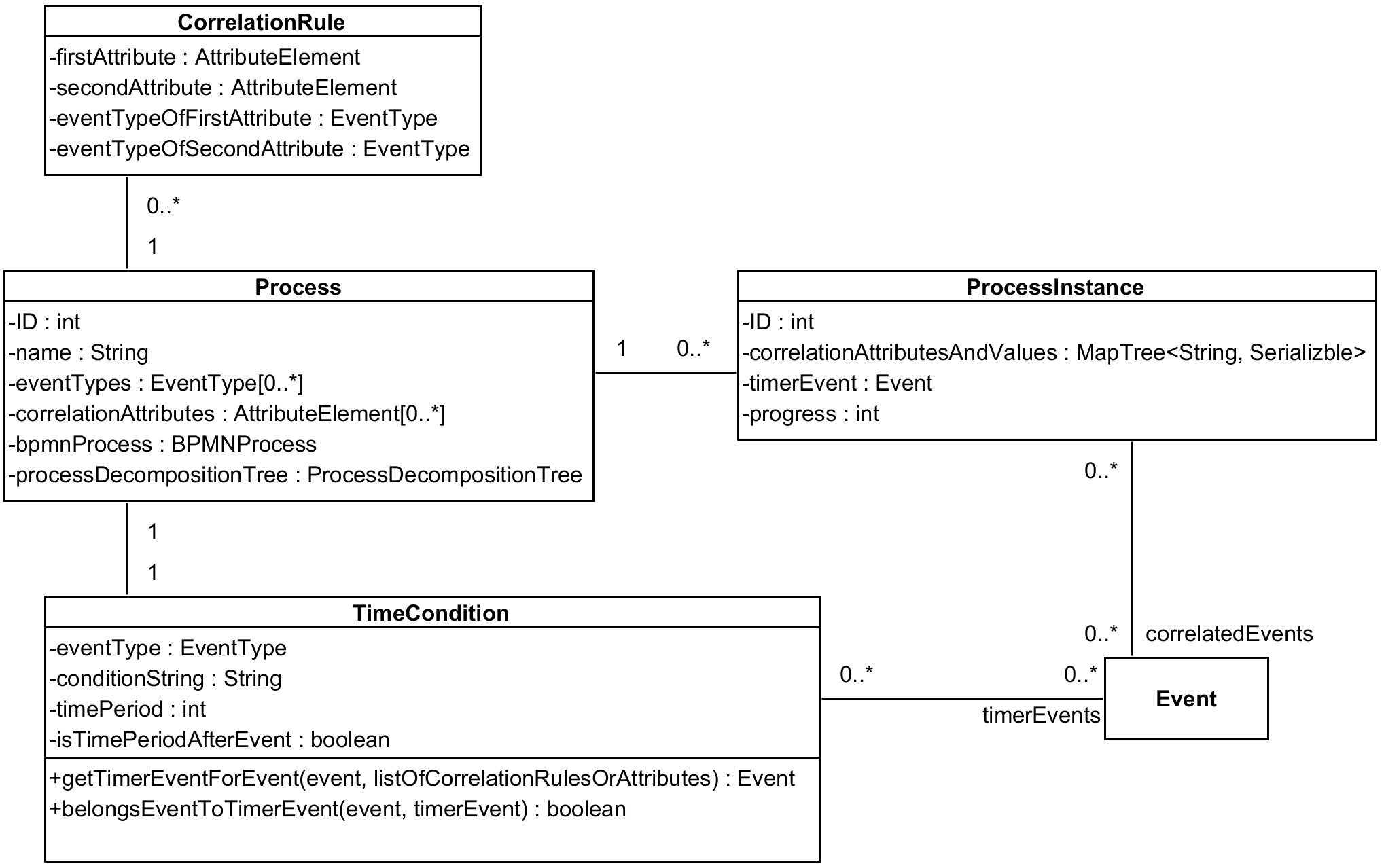
|  |  |
| --- | --- |
|  | Eine *TransformationRule* hält außerdem *ExternalKnowledgeExpressionSets*, die ihrerseits jeweils eine Menge an *ExternalKnowledgeExpressions* enthalten. In diesen wird bestimmt, wie und woher das zu einer Transformation optional hinzuzuziehende externe Wissen bestimmt wird. |

Die Erzeugung von *TransformationRules* (*createTransformationRule()*) und deren Registrierung (*register()*) und Abmeldung (*remove*()) vom *SushiStreamProcessingAdapter* erfolgt im *TransformationManager*. Dort können vor der Erzeugung einer *TransformtionRule* die dafür benötigten Parameter überprüft werden (*checkForValidity*). Der *TransformationManager* hat außerdem Zugriff auf eine Realisierung des *TransformationRuleParser*. Diese ist eine abstrakte Klasse, die Methoden zur Implementierung der Umwandlung der Elemente aus einer *TransformationRule* in die für eine Event Processing Language erforderliche Query bietet. Für die Esper EPL wird der *EsperTransformationRuleParser* genutzt.

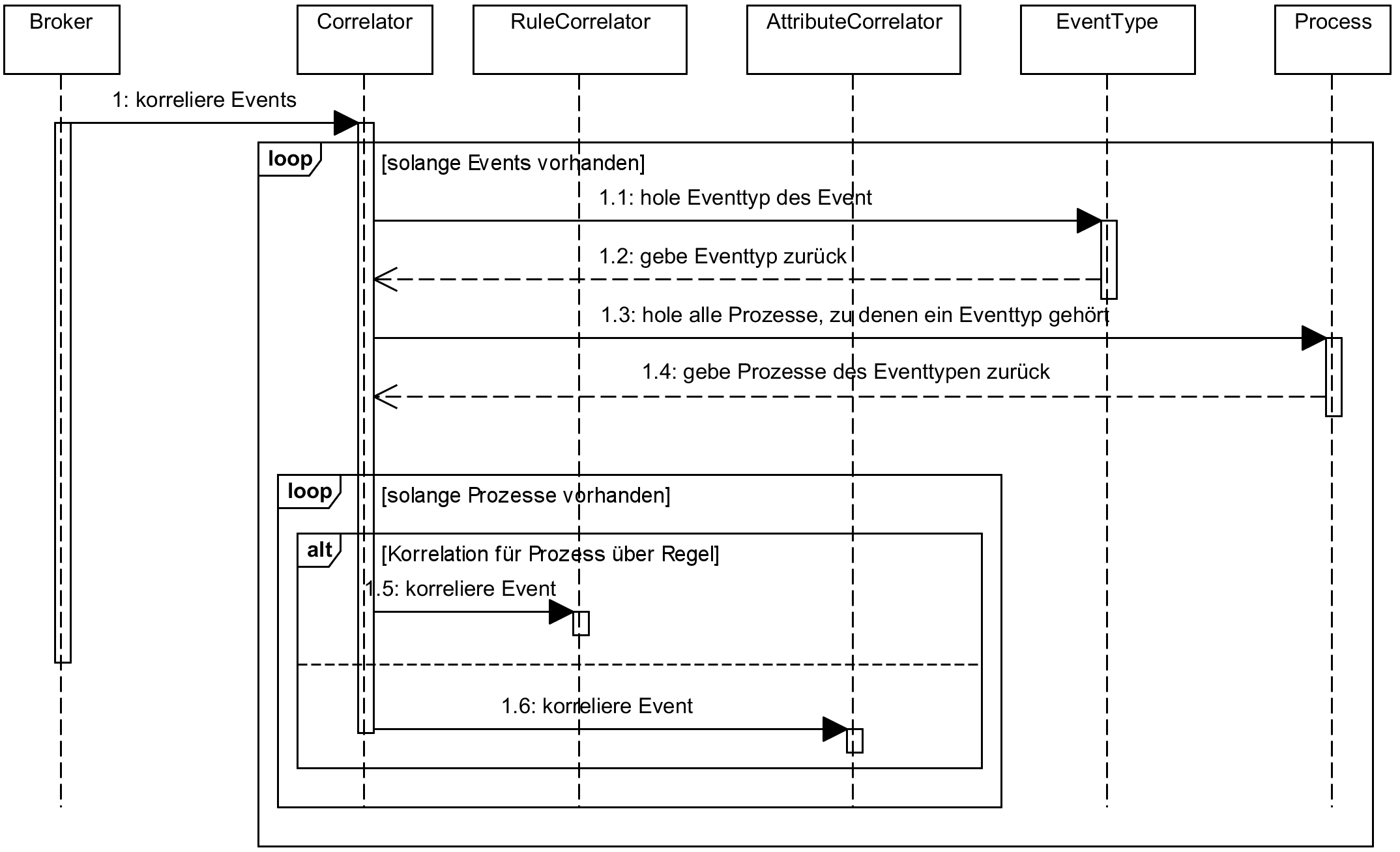


* 1. **Event Correlation**

Für die Korrelation von Events zu Prozessinstanzen werden *CorrelationRules* als Behälter verwendet. Jeder *SushiProcess* referenziert entweder auf eine Liste mit *CorrelationRules* (Korrelation über Paare verschiedennamiger Attribute) oder auf eine Liste mit *SushiAttributes* (Korrelation über gleichnamige Attribute). Ein *SushiProcess* referenziert auf *SushiProcessInstances*. *TimeConditions* werden genutzt für die erweiterte zeitliche Korrelation und sind genau einem *SushiProcess* zugeordnet.



Zentraler Ausgangspunkt für die Korrelation ist die Klasse *Correlator*. Dessen Methode *correlate()* wird u.a. vom *Broker* gerufen und leitet Events, die neu in die Plattform gelangt sind, je nach Art der Korrelation an den *RuleCorrelator* (Korrelation über Paare verschiedennamiger Attribute) oder an den *AttributeCorrelator* (Korrelation über gleichnamige Attribute) weiter. Diese beiden Klassen enthalten die für die Korrelation erforderliche Logik. Ihre correlate-Methoden werden verwendet für die Korrelation von Events, die sich bereits in der Plattform befinden.



1. **Event Repository**

Das Event Repository dient zur Visualisierung von Events, Eventtypen, Prozessen, Prozessinstanzen und BPMN-Modellen, die sich jeweils in der Datenbank befinden. Die Implementierung für das Event Repository befindet sich im *SushiWicket*-Projekt im Paket sushi.application.pages.eventrepository. Jedes Nav ist in etwa gleich aufgebaut. Es gibt eine Panel-Seite, welche die Darstellung der Daten und einen Filter enthält. Sie dient als Controller und in Kombination mit dem zugehörigen Mark-Up als View. Die Bereitstellung der Daten für jedes Nav erfolgt jeweils über einen Provider aus dem Paket sushi.application.pages.eventrepository.model, welcher das Model darstellt. In diesem Paket befinden sich auch die Filterklassen, die jeweils zur Auswahl der Ergebnisse in den Navs dienen.Es ist auch jeweils möglich auf den entsprechenden Navs Eventtypen (*EventTypEditor*) und Prozesse (*ProcessEditor*) anzulegen.

1. **Queries**
   1. **On-Demand**

Die On-Demand Queries werden über die Klasse *OnDemandQueryEditor* aus dem Paket sushi.application.pages.querying des Projektes SushiWicket erstellt. Diese erbt von der *QueryEditor*-Klasse und verwendet das *QueryEditorHelpModal*.

Beim Speichern einer neuen On-Demand-Query wird ein neues *SushiQuery*-Objekt erstellt (Projekt: SushiEsper, Paket: sushi.query). Dieses hat den Typ *SushiQueryTypeEnum* „ONDEMAND“. Das Objekt wird in der Datenbank über die Methode *save()* der *SushiQuery*-Klasse gespeichert. Vor dem Speichern wird die Abfrage über die Methode *validate*() der *SushiQuery*-Klasse überprüft. Dabei wird die Singleton-Instanz des *SushiStreamProcessingAdapters* verwendet.

Die SushiQuery kann über die Methode *execute()* ausgeführt werden.

* 1. **Live**

Die Live-Queries werden über die Klasse *LiveQueryEditor* aus dem Paket sushi.application.pages.querying des Projektes SushiWicket erstellt. Diese erbt von der *QueryEditor*-Klasse und verwendet das *QueryEditorHelpModal*.

Beim Speichern einer neuen Live-Query wird ein neues *SushiQuery*-Objekt erstellt (Projekt: SushiEsper, Paket: sushi.query). Dieses hat den Typ *SushiQueryTypeEnum* „LIVE“. Das Objekt wird in der Datenbank über die Methode *save()* der *SushiQuery*-Klasse gespeichert. Zudem wird die Abfrage zur Stream-Processing-Engine Esper hinzugefügt. Das geschieht über die Methode *addToEsper*() der *SushiQuery*-Klasse. Dabei wird die Singleton-Instanz des *SushiStreamProcessingAdapters* verwendet.

Beim Neuinitialisieren des *SushiStreamProcessingAdapters* werden die Live-Queries aus der Datenbank abgefragt und neu registriert. Das gleiche wird auch mit den Eventtypen aus der Datenbank gemacht.

* 1. **BPMN**



In diesem Abschnitt wird die Implementierung der Abfragenerstellung aus einemBPMN-Modell beschrieben. Nachdem das BPMN-Modell in unsere Plattform importiert wurde, wird das BPMN-Modell mit dem RPST-Algorithmus zerlegt. Diese Zerlegung passiert im *RPSTBuilder*. Wir haben für die Implementierung der Zerlegung diese RPST-Implementierung verwendet (http://code.google.com/p/jbpt/).Der resultierende RPST wird dann in einen *SushiTree<AbstractBPMNElement>* umgewandelt, da der RPST eigentlich nur Kanten und keine Knoten betrachtet. In dem Treewerden dann für die einzelnen Komponenten sogenannte Pattern ermittelt, die die Struktur wiederspiegeln (zum Beispiel XOR, AND, ...). Der aus der Zerlegung im RPSTBuilder resultierende Tree wird dann zur Erzeugung der Abfragen verwendet. Die Abfragen werden in Subklassen von *AbstractPatternQueryFactory* erzeugt. Diese Klassen sind als Factory-Method implementiert. Die Erzeugung beginnt in *PatternQueryFactory*, diese Klasse entscheidet über die konkret zu nutzende Factory. Die Factories erzeugen dann jeweils pro Abfrage eine *SushiPatternQuery***.** Die Abfrage wird dann noch bei Esper registriert. Als letztes muss dann noch eine Zuordnung der beobachtbaren Events in der Plattform zu den Elementen des Prozessmodells erfolgen. Das passiert über die *MonitoringPoint*-Klasse. Ein *MonitoringPoint* beinhaltet eine Zuordnung zwischen einem BPMN-Element und einem Eventtyp in der Plattform und schafft somit die Verbindung zwischen Events und Elementen. Die Eventtypen in den *MonitoringPoints* werden bei der konkreten Erzeugung der Abfragen genutzt.

In unserem UI in *SushiWicket* ist der *BPMNQueryEditor* für die Erzeugung von Queries verantwortlich, hier können auch die *MonitoringPoints* bearbeitet werden und es werden vor der Erzeugung der Abfragen die benötigen Vorbedingungen geprüft.

Benötigte Vorbedingungen:

* Eventtypen müssen für die benötigten Monitoring Points angelegt (Monitoring Points können in der zum Prozessmodell gehörenden XML-Datei in den ExtensionElements definiert werden oder bei der Erstellung der Abfragen hinzugefügt werden)
* es muss ein Prozess angelegt sein
* es muss ein BPMN-Prozess importiert werden und dieser BPMN-Prozess muss einem normalen Prozess zugeordnet sein
* es muss eine Korrelation angelegt werden, die die Eventtypen, die bei der BPMN-Prozess-Beobachtung/Abfrageerstellung benötigt werden, und den Prozess, der den BPMN-Prozess enthält, mit einander verbindet und die Events zu diesen Eventtypen direkt zu Prozessinstanzen korreliert (ohne Prozessinstanzen funktionieren die Abfragen und die Beobachtung nicht)

1. **Monitoring**
   1. **BPMN**



Der *BPMNQueryMonitor* kapselt das zum Monitoring benötigte Verhalten und dient als Schnittstelle für andere Module der Plattform. Er ist als Singleton implementiert. Zum einen sammelt er Informationen über die Status der einzelnen Prozessinstanzen. Zum anderen dient der *BPMNQueryMonitor* auch als Schnittstelle für die *SushiPatternQueryListener*. Diese Listener können auf die Aktualisierung einer Abfrage reagieren. In diesemFall informieren sie den *BPMNQueryMonitor* über die Aktualisierung der ihnen zugeordneten Abfragen für BPMN-Elemente. Der *BPMNQueryMonitor* reicht diese Aktualisierungen dann an die weiteren zur Beobachtung nötigen Monitor-Klassen weiter. Zuerst an den *ProcessMonitor*, dieser beobachtet einen ganzen Prozess und sammelt Informationen über den Prozess. Er enthält mehrere *ProcessInstanceMonitore*. Diese beobachten jeweils einzelne Prozessinstanzen. *ProcessMonitore* und *ProcessInstanceMonitore* werden jeweils dann erzeugt, wenn sie das erste Mal abgefragt werden oder wenn Status-Updates für die betreffende Monitore auftreten. Die *QueryMonitore*beobachten dann konkrete Abfragen und liegen im *ProcessInstanceMonitor*. Bei der Erzeugung eines *ProcessInstanceMonitors* werden jeweils auch alle benötigten *QueryMonitore* direkt erzeugt. Die *QueryMonitore* verwalten den Status einer *SushiPatternQuery* mithilfe der Enums *ViolationStatus* (für Ausführungsverletzungen bei der Ausführung) und *QueryStatus* (für den konkreten Status der Abfrage). Die Erkennung von Ausführungsverletzungen erfolgt mit dem *ViolationMonitor*. Dieser ist jeweils einem *ProcessInstanceMonitor* zugeordnet und prüft bei jedem Status-Update im *ProcessInstanceMonitor* auf Ausführungsverletzungen unter Nutzung der *QueryMonitore*.

* 1. **Monitoring ohne BPMN**

Ohne BPMN-Modelle kann der Prozessfortschritt über Abfragen überwacht werden. Dieses Verfahren ist nur im Backend der Plattform implementiert und noch nicht in der Oberfläche repräsentiert.



Zur Überwachung des Prozesses werden *QueryMonitoringPoints* erzeugt. Diese verbinden Abfragen mit Prozessen.

Die Abfragen werden als *Query*-Objekt persistiert. Für jedes *Query*-Objekt wird ein *QueryListener* erstellt. Dieser Listener übernimmt die Aufgabe, auf das Auslösen der Abfrage zu reagieren. Der *StreamProcessingAdapter* registriert die Abfragen und die dazugehörigen *Listener* bei der *EPRuntime*, die aus der verwendeten Event Processing Engine Esper stammt. Die Events werden dann ebenfalls durch den *StreamProcessingAdapter* an die *EPRuntime* geschickt, wodurch die registrierten Abfragen getriggert und die *QueryListener* mit der Methode *update()* ausgelöst werden.

Um die Vorgänge beim Aktualisieren der Fortschritte der einzelnen Prozessinstanzen zu verstehen, ist es wichtig zu wissen, dass die einzelnen Events über einen Broker in unsere Plattform gelangen. Der Broker sendet die Events an die verschiedenen Verarbeitungseinheiten der Plattform. Bevor diese Events jedoch weitergeleitet werden, werden sie zunächst korreliert. Unter einer Korrelation verstehen wir die Zuordnung der einzelnen Events zu Prozessinstanzen. In dem Moment, wenn die Events also über den *StreamProcessingAdapter* an unsere Streaming-Datenbank gesendet werden, sind die Events bereits mit Prozessinstanzen verknüpft.

Die korrelierten Events werden an die *EPRuntime* geschickt. Wenn nun eine Abfrage ausgelöst wird, wird die Methode *update()* des *QueryListeners* ausgeführt. Der *QueryListener* fragt das zu dem Listener gehörende *Query*-Objekt nach den zu dieser Abfrage registrierten *QueryMonitoringPoint*-Objekten. Anschließend überprüft er, ob die Prozessinstanzen der triggernden Events auch zu den in den *QueryMonitoringPoints* definierten Prozessen gehören. Ist diese Überprüfung positiv, wird der *QueryMonitoringPoint*getriggert. Die *ProcessInstance* wird dabei übergeben. Der *QueryMonitoringPoint* aktualisiert anschließend den Fortschritt der *ProcessInstance*, wobei entweder die Methode *addToProgress()* bei relativer Prozentangabe in dem *QueryMonitoringPoint* oder die Methode *setProgress()* bei absoluter Prozentangabe in dem *QueryMonitoringPoint* aufgerufen wird.



Die aktuellen Prozessausführungsfortschritte sind dadurch in der Prozessinstanz gespeichert und können abgerufen werden.

* 1. **Attribute Charts**

Die Implementierung der Attributdiagramme ist sehr ähnlich zu der Implementierung der Event Views. Die Erstellung eines Attributdiagramms erfolgt über die Web-Oberfläche der Plattform. Dafür haben wir ein Dialogfenster, repräsentiert durch die Klasse *ChartConfigurationPanel*, entwickelt, in dem der Nutzer alle Konfigurationen vornehmen kann. Um die Konfiguration für den Nutzer möglichst einfach zu halten und eine fehlerfreie Konfiguration sicherzustellen, müssen die einzelnen Formularfelder des Dialogs miteinander verknüpft sein und sich interaktiv an die gewählten Werte der anderen Felder anpassen. So müssen zum Beispiel die wählbaren Attribute mit den Attributen des gewählten Eventtyps übereinstimmen. Um das zu realisieren, haben wir den Feldern ein Ajax-Updating-Verhalten hinzugefügt. In dem Verhalten wird die Aktualisierung der abhängigen Felder ausgelöst, die wiederum kaskadierend weitere Änderungen auslösen kann. Beim Abschicken des Formulars wird ein Objekt der Klasse *ChartConfiguration* erzeugt und in der Datenbank gespeichert. Bei diesem Objekt handelt es sich um ein reines Datenobjekt, das die notwendigen Konfigurationen kapselt.



Wie bei den Event Views erfolgt die Anzeige ebenfalls über eine Klasse des UIs, in dem Fall die *ChartPage*-Klasse. Die gespeicherten *ChartConfiguration*-Objekte werden aus der Datenbank geladen. Für jedes Konfigurationsobjekt wird dann entweder ein SplatterChartOptions- oder ein ColumnChartOptions-Objekt erzeugt.

Diese Klassen erben von der durch Wicked Charts definierten Oberklasse *Options*. Dazu wird der Typ des Konfigurationsobjektes geprüft: Es handelt sich entweder um ein Attributwertdiagramm, umgesetzt durch einen sogenannten Splatter-Chart, oder um ein Attributwerthäufigkeitsdiagramm, umgesetzt durch einen Column-Chart. Der Typ des Konfigurationsobjektes wird durch das Attribut type abgebildet und ist als Enumeration implementiert. Die *AttributeChartType-Enumeration* erlaubt momentan ausschließlich diese beiden Werte. Durch die Verwendung dieses Konstrukts ist es jedoch leicht möglich, andere Arten von Attributdiagrammen hinzuzufügen, indem einfach der Wertebereich der Enumeration erweitert wird. Das *Options*-Objekt konfiguriert das Diagramm und bereitet die Daten vor. Zur Anzeige auf der Oberfläche wird aus dem *Options*-Objekt eine *Chart*-Komponente erzeugt, die auf der Oberfläche angezeigt werden kann.

* 1. **Event Views**



Die Erstellung eines Event Views erfolgt über die Web-Oberfläche der Plattform. In dem *ViewConfigurationPanel* kann der Nutzer die Konfigurationen fürseinen Event View vornehmen. Bestätigt der Nutzer seine Eingaben, wird ein*EventView*-Objekt erzeugt und in der Datenbank gespeichert. Damit diesmöglich ist, erbt die Klasse von der *Persistable*-Klasse, diedementsprechende Methoden bereitstellt. Bei demEventView-Objekt handelt es sich um ein reines Datenobjekt, das die notwendigenKonfigurationen kapselt.

Die Anzeige der Event Views wird ebenfalls über die Web-Oberfläche der Plattform ausgelöst. Die *EventViewPage* lädt die gespeicherten*EventView*-Objekte aus der Datenbank. Für jedes dieserKonfigurationsobjekte wird dann ein *EventViewOptions*-Objekt erzeugt. Diese Klasse erbt von der durch Wicked Charts definierten Oberklasse Options. Dieses Objekt bereitet die Informationen des *EventViewOptions*-Objektes in eine für das Visualisierungsframework verständliche Form auf, ruft die relevanten Daten aus der Datenbank ab und bereitet diese als Datenserien auf. Zur Anzeige auf der Oberfläche muss nun aus dem *EventViewOptions*-Objekt eine *Chart*-Komponente erzeugt werden. Diese Klasse entstammt ebenfalls dem Wicked Charts-Framework und erbt von der Klasse Component, die wiederum aus dem verwendeten Web-Framework Wicket stammt. Durch dieses Vererbungsverhalten kann das *Chart*-Objekt direkt auf der Oberfläche angezeigt werden. BeimInitialisieren dieses *Chart*-Objektes entsteht das eigentliche Diagramm.Angezeigt wird das Objekt auf der *EventViewPage*.

* 1. **Notifications**

Es gibt Event-Benachrichtigungen und Query-Benachrichtigungen. Für beide Arten muss zunächst eine Benachrichtungsregel erstellt werden, bevor Benachrichtigungen erzeugt werden können. Die Event- und Query-Benachrichtigungsregeln werden vom Nutzer der Plattform auf der Web-Oberfläche erstellt. Die Erstellung erfolgt über ein Dialogfenster.

Die zwei Arten von Benachrichtigungsregeln erben von der gemeinsamen, abstrakten Oberklasse *NotificationRule*. Unabhängig von der Art der Benachrichtigungsregel speichert sie einen *User*, also den Nutzer der Plattform, für den die Benachrichtigungen erstellt werden, und die *NotificationPriority*. Diese wurde als Enumeration implementiert und bestimmt die Form der Benachrichtigung. Z.B. wird eine Benachrichtung per E-Mail bei hoher Priorität oder ausschließlich eine Benachrichtung auf der Web-Oberfläche bei geringerer Priorität ausgelöst. Beide Arten von Benachrichtigungsregeln verfügen jeweils über eine *TriggeringEntity*, die einheitlich über die Methode *getTriggeringEntity()* abgefragt werden kann.



Die Event-Benachrichtigungsregeln werden durch die Klasse *EventNotificationRule* implementiert. Diese ist immer mit einem bestimmten *EventType* verbunden, der die *TriggeringEntity* repräsentiert, da ein Auftreten eines Events dieses Typs die Regel triggert. Das geschieht über die Methode *trigger()*.Zusätzlich kann auch eine *Condition* zu der Event-Benachrichtungsregel gespeichert werden, die die Benachrichtung auf Events beschränkt, die einen bestimmten Attributwert enthalten.

Die Klasse *QueryNotificationRule* repräsentiert die Query-Benachrichtigungsregeln. Im Gegensatz zu den Event-Benachrichtigungsregeln ist die *TriggeringEntity* hier eine *Query*, deren Auslösen die Benachrichtungsregel triggert.

Die Benachrichtigungsregeln werden zur dauerhaften Persistierung in der Datenbank abgelegt und erben daher wie auch die Klassen *User, Condition, Eventtyp* und *Query* von der *Persistable*-Klasse.



Event-Benachrichtigungen werden von den oben erläuterten *EventNotificationRules* erzeugt. Eine *EventNotification* erbt von der Oberklasse *Notification*, die die Gemeinsamkeiten der *EventNotifications* und *QueryNotifications* kapselt.

Diese *Notifications* repräsentieren eine konkrete Benachrichtigung, die für einen bestimmten Nutzer erstellt wurde, um auf eine konkrete Situation aufmerksam zu machen. Eine *Notification* enthält daher einen timestamp, der die Erstellungszeit angibt, und einen Boolean-Wert seen, der angibt, ob der Nutzer diese Benachrichtung bereits zur Kenntnisgenommen hat. Die *EventNotification* ist mit der *EventNotificationRule* verbunden, die sie erstellt hat, und mit dem *Event*, über das sie benachrichtigt. Die verschiedenen *EventNotifcationRules* werden von der *NotificationObservable*-Klasse gespeichert. Mit dieser Klasse setzen wir das Observer-Pattern um. Die Klasse*NotificationObservable* implementiert die Rolle des Observables und speichert die *NotificationRules*, die als Observer fungieren. Die Methode *trigger*() setzt in unserer Implementierung die Funktionalität der standardmäßig update() genannten Funktion des Observables um, die Observer über Änderungen informiert.



Nun soll der konkrete Ablauf bei der Erzeugung von *EventNotifications* erläutert werden. Die Events werden durch die verschiedenen Adapter an den *Broker* gesendet, der als zentraler Einstiegspunkt der Plattform gilt und die eintreffenden Events an Datenbank und Streaming Engine weitergibt. Der *Broker*sendet jedes eintreffende Event über die Methode *trigger*() auch an die*NotificationObservable*-Klasse. Diese leitet das Event an die *EventNotificationRules* weiter, die für den *Eventtyp* des eintreffenden*Events* registriert sind. Die *EventNotificationRule* schickt nun über die Methode *matches*() eine Abfrage an seine *Condition*, ob die Bedingung auf dieses Event zutrifft. Die *EventNotificationRule* nimmt die Antwort entgegen und erzeugt, sofern diese positiv ausgefallen ist, eine *EventNotification*, die mit dem entsprechenden Event verbunden ist.

Beim Erzeugen der Benachrichtigung wird diese in der Datenbank abgelegt und, je nach Priorität der Benachrichtigungsregel, weitere Aktivitäten ausgeführt.



Die *QueryNotifications* erben ebenfalls von der Oberklasse *Notification*. Diese werden durch die bereits vorgestellten *QueryNotificationRules* erzeugt. Die Events werden durch den *StreamProcessingAdapter*an die *EPRuntime* geschickt. Lösen nun die eintreffenden Events eine vorher registrierte *Query* aus, so wird der zugehörige *QueryListener* angestoßen und die Methode *update*() wird ausgeführt.



Der Erzeugung einer *QueryNotification* läuft wie folgt ab: Die *EPRuntime* löst, wie bereits erwähnt, die *update*()-Methode des *QueryListeners*aus. Der *QueryListener* fragt daraufhin die dazugehörige *Query* nachdem für diese Abfrage eingetragenen *QueryNotificationRules*. Der*QueryListener* triggert anschließend die *QueryNotificationRule* undübergibt dabei das Ergebnis der Query-Ausführung als String. Die *QueryNotificationRule* erzeugt dann eine neue *QueryNotification*. Bei dieser Erzeugung können äquivalent zur Erzeugung von Event-Benachrichtigungen noch weitere Aktivitäten ausgeführt werden.

Die Benachrichtigungen können unabhängig davon, ob es sich um eine Event-Benachrichtigung oder eine Query-Benachrichtigung handelt, auf der Weboberfläche der Plattform angezeigt werden. Dazu wird der aktuell eingeloggte Nutzer überprüft und für diesen aus der Datenbank die angelegten *Notifications* abgefragt.

Durch die Implementierung über die Oberklasse *Notification* können die Objekte der zwei Klassen auch in einer Tabelle angezeigt werden, da es sich bei dem jeweiligen Trigger zwar um unterschiedliche Attribute handelt, beide Klassen aber über die Methode *getTriggeringText*() verfügen und die Oberfläche damit für beide Klasseneinen Wert für diese Spalte abfragen kann. Durch Markieren der jeweiligen Zeile und Betätigen des Knopfes „seen“ können die *Notifications* bestätigt werden. Dadurch wird das seen-Attribut der *Notifications* auf true gesetzt und sie werden nichterneut angezeigt.

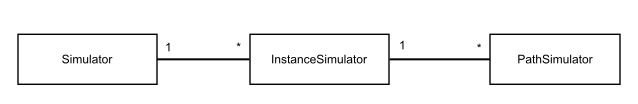
1. **Export**

Die EPP bietet die Möglichkeit Events ohne hierarchische Attribute als CSV-Datei zu exportieren. Die Funktionalität hierzu liegt in der Klasse *CSVExporter*. Dabei werden immer nur Events eines Eventtypen exportiert. Der *CSVExporter* erhält eine Liste von zu exportierenden Events und den Eventtypen. Die Attributnamen aus dem Eventtypen bestimmen den Inhalt der ersten Zeile (die Überschriften). Jedes Event wird in der CSV-Datei zu einer weiteren Zeile, in der die Attributwerte gespeichert werden.

1. **Event Producing**
   1. **Simulator**

Die Simulation ist in erster Linie dazu gedacht, die EPP zu testen. Die Simulation wird mit und ohne Prozessmodell unterstützt. Das Prozessmodell muss dabei angehängte Monitoring Points haben. Die Simulation ohne Modell ist mit einer vom Nutzer erstellten Eventfolge möglich. Diese ist in Form einer Baumstruktur dargestellt. Es ist dabei möglich, exklusive oder parallele Ausführungspfade zu erstellen. Um den Implementierungsaufwand gering zu halten, funktioniert die Simulation ohne Prozessmodell intern mit Hilfe eines generierten Prozessmodells. Die Klasse *SimulationTreeTableToModelConverter* ist für die Umwandlung der Baumstruktur in ein Prozessmodell zuständig. Sie liegt im Paket SushiWicket, da sie mit Komponenten von Wickte umgehen muss. Dazu wird zunächst die Methode *convertTreeToBPMNTree*() gerufen, die eine Umwandlung in BPMN-Komponenten vornimmt. Hierbei wird unter anderem für jeden Eventtypen der Folge ein Task erstellt, an dessen Monitoring Point am Zustandsübergang terminate der Eventtyp gehängt wird. Anschließend wird der Baum durch die Methode createSubBranch() rekursiv in ein Prozessmodell übersetzt. Die Methode bekommt initial die äußere Komponente des Baums, den Rekursionsanker bilden bei dieser Umwandlung die Tasks.

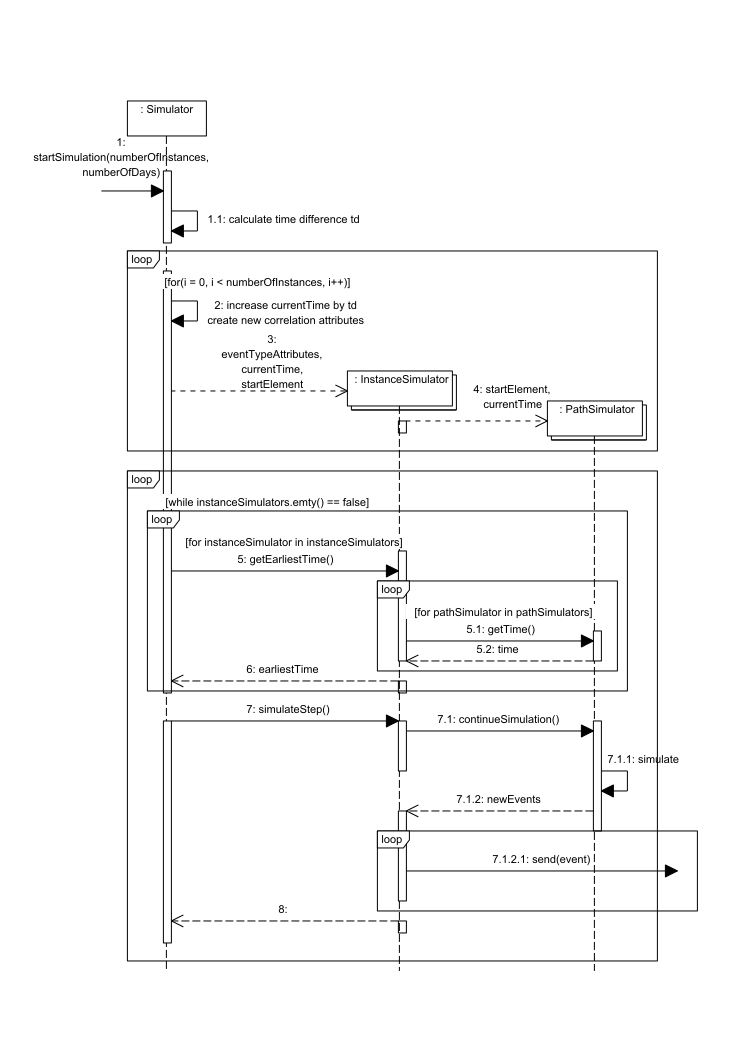
Die eigentliche Simulation wird von drei Klassen übernommen. Diese sind im nachfolgenden Klassendiagramm dargestellt.



Für jeden Simulationsaufruf wird eine Instanz der Klasse *Simulator* erzeugt. Diese referenziert pro zu erzeugender Instanz einen *InstanceSimulator*, der wiederum mehrere *PathSimulator*-Objekte halten kann (einen pro aktivem Ausführungspfad in der Prozessinstanz).

Der Ablauf der Simulation ist im nachfolgenden Sequenzdiagramm dargestellt. Nachdem der Simulator bei seiner Erzeugung bereits Eingaben wie Wertebereiche für Attribute, Wahrscheinlichkeiten für exklusive Verzweigungen erhalten hat, wird die Simulation mit dem Aufruf der Methode *startSimulation(numberOfInstances, numberOfDays)* gestartet. Danach ermittelt der *Simulator* zunächst die Zeitdifferenz, die für ein gleichverteiltes Starten der Instanzen über den angegebenen Zeitraum sorgt. Anschließend wird für jede Instanz die Startzeit angepasst und die Werte der Korrelationsattribute ermittelt.

Mit diesen Werten wird ein Objekt der Klasse *InstanceSimulator* konstruiert, die wiederum ein Objekt vom Typ *PathSimulator* erzeugt. Nachdem dies geschehen ist beginnt die eigentliche Simulation. Dies wird im *Simulator* mit der Methode *startSimulation()* ausgelöst, die so lange wie es *InstanceSimulator*-Objekte gibt das mit der frühesten Simulationszeit in einem zugehörigen *PathSimulator* anstößt (*simulateStep()*). Der *InstanceSimulator* ruft anschließend im frühesten *PathSimulator* die Methode *continueSimulation()* auf.



In dieser Methode wird ein Simulationsschritt durchgeführt, der je nach aktuellem Element und Zustand (Element bereits durchlaufen oder nicht) anders aussehen kann. Wenn das aktuelle Element noch nicht durchlaufen wurde, wird dies durchgeführt. Dabei werden Events erzeugt, falls am Zustandsübergang „enable“ des Elements ein Monitoring Point angehängt ist. Anschließend wird die Simulationszeit angepasst. Wenn das Element bereits durchlaufen wurde gilt dies für den Zustandsübergang „terminate“. Anschließend wird zum nächsten Element übergegangen.

Gibt es kein nachfolgendes Element, so kann der *PathSimulator* nicht mehr weiter laufen und meldet sich beim *InstanceSimulator* ab. Gleiches gilt, wenn es ein AND-Join als Nachfolger gibt, der jedoch noch nicht aktiviert werden kann (da noch nicht alle Vorgänger besucht wurden). Hat ein Element mehrere Nachfolger, so meldet sich der *PathSimulator* ebenfalls ab, erzeugt jedoch für jedes Nachfolgeelement einen neuen *PathSimulator*.

Handelt es sich bei dem aktuellen Element jedoch um einen XOR-Gateway, so wird kein *PathSimulator* erzeugt oder abgemeldet, sondern der existierende *PathSimulator* wählt einen Pfad. Nachdem dies geschehen ist, werden alle Elemente „geskippt“, die nicht mehr erreichbar sind. Während der Ausführung dieser Methode gibt es weitere Kommunikation mit dem Simulator (wegen Zeit, Pfadwahrscheinlichkeit) und dem *InstanceSimulator* (besuchte Vorgänger eines AND-Join). Die Methode gibt eine Liste von Events zurück an den *InstanceSimulator*. Hier werden die Events mit Attributwerten versehen und anschließend an den Broker gesendet. Wenn sich alle *PathSimulator* eines *InstanceSimulator* abgemeldet haben, meldet sich der *InstanceSimulator* ebenfalls beim *Simulator* ab.

1. **Sign in**

Die EPP unterstützt eine einfache Nutzerverwaltung. Die EPP bietet dafür die Möglichkeit sich mit einem Namen und Password anzumelden. Für Nutzerverwaltung sind die Klassen LoginPage, Registerpage und SushiUser zuständig. Wenn sich ein Nutzer erfolgreich in der LoginPage eingeloggt hat, wird seiner Session (SushiAuthenticatedSession) einem User zugewiesen.