|  |
| --- |
| Kurzeinführung BPMN Statische Validierung |
| Maven-Plugin CamundaStaticValidator |
| Autor: Christian Tophinke, Claus Alexander Usener  Stand: 2016-09-24 |

|  |  |
| --- | --- |
| VERTEILER |  |
|  |  |
| AUTOR | [Hier klicken und den Autor eingeben] |
| ABLAGE |  |

Versionsverzeichnis

|  |  |  |  |
| --- | --- | --- | --- |
| Datum | Version | MA | Kommentar |
|  | 1.0 |  | Ersterstellung |
|  |  |  |  |

Inhaltsverzeichnis

1 Beispiel Überschrift 1. Ebene 3

1.1 Beispiel Überschrift 2. Ebene 3

2 Kurzanleitung Handbuch 3

2.1 Abbildungsverzeichnis 3

2.2 Index 3

A Anhang Beispiel 1 3

B Anhang Beispiel 2 3

Index 3

# Einführung

Dieses Nutzerhandbuch gibt eine kurze Einführung in das Statische Analyse-Werkzeug, mit dem typische Fehler in prozessgesteuerten Anwendungen auf Basis von Camunda BPM identifiziert werden können. Es werden über ein Maven-Plugin potenzielle Fehler im Zusammenspiel von Prozessmodell und Quellcode aufgedeckt.

Zunächst wird die Installation und Nutzung des Werkzeugs an einem einfachen Beispiel erläutert. Danach werden übergreifende Einstellungen beschrieben. Der letzte Teil erklärt die einzelnen Prüfungen im Detail.

# Grundszenario

In diesem Kapitel wird die Installation und Nutzung des Werkzeugs anhand eines einfachen Beispielprojektes erläutert. Im Beispiel geht es um die Schadensabwicklung für einen Kfz-Glasbruch innerhalb einer Versicherung.

## Voraussetzungen

Folgende Voraussetzungen gegeben sein, damit eine Analyse durchgeführt werden kann:

* Java 1.7 (JDK)
* Maven 2

## Installation

Das Werkzeug zur statischen Analyse wird als Build-Schritt in die Maven-Konfiguration des zur prüfenden Projektes eingebunden. Hierzu muss die Maven-Konfiguration des Projekts (pom.xml) um folgende Plugin-Konfiguration ergänzt werden:

<project ...>

...

<build>

<plugins>

...

<plugin>

<groupId>de.viadee.bpm</groupId>

<artifactId>CamundaStaticValidator</artifactId>

<version>2.0.0-SNAPSHOT</version>

<executions>

<execution>

<phase>prepare-package</phase>

</execution>

</executions>

</plugin>

...

</plugins>

</build>

</project>

Die Konfiguration des Plugins geschieht über die Datei **ruleSet.xml** im Pfad "src/main/resources". Als Grundeinstellung kann folgender Inhalt übernommen werden:

<ruleSet>

<rule>

<name>JavaDelegateChecker</name>

<state>true</state>

</rule>

<rule>

<name>EmbeddedGroovyScriptChecker</name>

<state>true</state>

</rule>

<rule>

<name>VersioningChecker</name>

<state>true</state>

<settings>

<setting name=*"versioningSchemaClass*">([^\_]\*)\_{1}([0-9][\_][0-9]{1})\.(java|groovy)

</setting>

</settings>

</rule>

<rule>

<name>DmnTaskChecker</name>

<state>false</state>

</rule>

<rule>

<name>ProcessVariablesModelChecker</name>

<state>false</state>

</rule>

<rule>

<name>ProcessVariablesNameConventionChecker</name>

<state>true</state>

<elementConventions>

<elementConvention>

<name>internal</name>

<!-- field types for process variables -->

<elementFieldTypes excluded=*"true"*>

<elementFieldType>Class</elementFieldType>

<elementFieldType>ExternalScript</elementFieldType>

<elementFieldType>DMN</elementFieldType>

<elementFieldType>DelegateExpression</elementFieldType>

</elementFieldTypes>

<pattern>int\_[a-zA-Z]+</pattern>

</elementConvention>

<elementConvention>

<name>external</name>

<!-- field types for process variables -->

<elementFieldTypes>

<elementFieldType>Class</elementFieldType>

<elementFieldType>ExternalScript</elementFieldType>

<elementFieldType>DMN</elementFieldType>

<elementFieldType>DelegateExpression</elementFieldType>

</elementFieldTypes>

<pattern>ext\_[a-zA-Z]+</pattern>

</elementConvention>

</elementConventions>

</rule>

<rule>

<name>TaskNamingConventionChecker</name>

<state>false</state>

<elementConventions>

<elementConvention>

<name>convention</name>

<pattern>[A-ZÄÖÜ][a-zäöü\\\-\\\s]+</pattern>

</elementConvention>

</elementConventions>

</rule>

</ruleSet>

Danach wird das Plugin-Projekt mit folgendem Maven-Goal "clean install" installiert.

Damit Spring Beans und Prozessvariablen zur Prozessinstanziierung im Beispiel ebenfalls mit analysiert werden sind zwei zusätzliche Schritte durchzuführen (vgl. Kapitel 3.1 und 3.2).

## Nutzung

Das Plugin ist für das Beispielprojekt mit folgendem Maven-Goal auszuführen:

test de.viadee.bpm:CamundaStaticValidator:2.0.0-SNAPSHOT:check

Die detaillierten Prüfungsergebnisse können anschließend in der Datei bpmn\_validation.xml bzw. bpmn\_validation.json in dem Verzeichnis „target“ innerhalb des zu prüfenden Projekts eingesehen werden.

# Übergreifende Einstellungen

In diesem Kapitel werden Einstellungen des Werkzeugs erläutert, die sämtliche Konsistenzprüfungen betreffen.

## Konfiguration zur Erkennung von Spring Beans

Damit die im Quellcode referenzierten SpringBeans gefunden werden können, ist das zu analysierender Projekt zusätzlich um folgende Abhängigkeit in der Maven-Konfiguration (pom.xml) zu ergänzen:

<dependency>

<groupId>de.viadee.bpm</groupId>

<artifactId>CamundaStaticValidatorUtils</artifactId>

<version>2.0.0-SNAPSHOT</version>

</dependency> >

Zusätzlich muss ein Unit Test erstellt werden, der ein Bean-Mapping im „target“-Verzeichnis erzeugt. Diese Datei enthält eine Zuordnung der Bean-Namen zu den Java-Klassen. Als Testklasse reicht der folgende Quellcode:

**import** org.junit.Test;

**import** org.junit.runner.RunWith;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.test.context.ContextConfiguration;

**import** org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

**import** de.viadee.bpm.camundaStaticValidator.CamundaStaticValidatorTestHelper;

@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration(classes = { SpringTestConfig.**class** })

**public** **class** GenerateBeanMappingTest {

@Autowired

**private** ApplicationContext ctx;

@Test

**public** **void** generateMapping() {

CamundaStaticValidatorTestHelper.*generateBeanMappingFile*(ctx);

}

}

Ziel des Testfalls ist es, den Spring-Context zu analysieren und die im Context zur Verfügung stehenden Beans zu identifizieren. Die bei der Ausführung des Testfalls identifizierten Beans werden in der Datei beanMapping.xml im Verzeichnis „target“ gespeichert. Das Plugin liest diese Datei aus, um Spring Beans innerhalb der Analyse erkennen zu können. Sofern die bestehende Spring-Konfiguration **noch** **nicht** funktioniert, kann für den Test **vorübergehend** auch eine Test-Spring-Konfiguration „gemockt“ werden (vgl. nachfolgendes Beispiel):

**import** org.mockito.InjectMocks;

**import** org.mockito.MockitoAnnotations;

**import** org.springframework.context.annotation.Configuration;

**import** de.viasurance.kfzglasbruch.delegate.KontoErmittelnDelegate\_1\_0;

@Configuration

**public** **class** SpringTestConfig {

**public** SpringTestConfig() {

MockitoAnnotations.*initMocks*(**this**);

}

@InjectMocks

**private** KontoErmittelnDelegate\_1\_0 kontoErmittelnDelegate;

@Bean

**public** KontoErmittelnDelegate\_1\_0 kontoErmittelnDelegate() {

**return** kontoErmittelnDelegate;

}

...

}

## Erkennung von Prozessvariablen im Quellcode (experimentell)

Diese Funktion ist als Proof-Of-Concept implementiert und daher als „**experimentell**“ zu betrachten.

Damit Datenfluss-Anomalien erkannt werden können, muss auch der vom Prozessmodell referenzierte Quellcode auf Prozessvariablen untersucht werden.

Die Quellcode-Dateien der in der Spring-Konfiguration referenzierten Klassen werden dafür mit einer Textanalyse mittels regulären Ausdrücken untersucht. Es wird nach den Methoden setVariable(...), getVariable(...) und removeVariable(...) gesucht, da diese im Camunda API[[1]](#footnote-1) für den Zugriff auf Prozessvariablen vorgesehen sind. Das erste Argument muss ein String sein und entspricht dabei dem Variablen-Namen (Bsp: .setVariable("ext\_vertrag", ...)).

Eine Annahme dieses Verfahrens ist, dass Variablen immer direkt in der referenzierten Implementierung verwendet werden. Camunda BPM verlangt bei Service Tasks die Implementierung der Schnittstelle *JavaDelegate*. Diese schreibt eine Methode *execute* vor, welche bei Ausführung zuerst aufgerufen wird.

## Analyse von Prozessvariablen bei Prozessinstanziierung

Diese Funktion ist als Proof-Of-Concept implementiert und daher als „**experimentell**“ zu betrachten.

Damit Prozessvariablen, die bei Prozessinstanziierung übergeben werden, innerhalb der Analyse zugeordnet werden können, muss der übergebene Variablen-Prozesskontext als Klasse, die von der Klasse InitialProcessVariablesBase erbt, definiert werden.

Es muss eine Klasse *InitialProcessVariables* angelegt werden, die sämtliche Variablen als Attribute enthält, die beim Prozessstart übergeben werden sollen. Später werden diese Attribute mit Werten gefüllt und über die Methode *createVariableMap* als Map an die Camunda-Methode für die Instanziierung übergeben.

Unterstützt werden folgende Camunda-Methoden zum Start einer Prozessinstanz:

* startProcessInstanceByKey
* startProcessInstanceByMessage

Die Basisklasse *InitialProcessVariablesBase* befindet sich im Maven-Projekt CamundaStaticValidatorUtils. Dieses Projekt muss ebenfalls als Dependency an der Stelle der Nutzung eingebunden werden (vgl. Kapitel 3.1).

Ein Beispiel für diese Konvention zeigt der folgende Quellcode zum Starten eines Prozesses, bei dem eine Versicherungsnummer und eine Schadenshöhe übergeben wird:

**public** **class** KfzGlasbruchRestController {

**class** InitialProcessVariables **extends** InitialProcessVariablesBase {

String ext\_vsnr;

Double ext\_schadenshoehe;

}

@Autowired

**private** RuntimeService runtimeService;

**public** String startProzess(Schadensmeldung meldung)**throws** Exception {

**final** InitialProcessVariables processVars = **new** InitialProcessVariables();

processVars.ext\_vsnr = meldung.getVersicherungsscheinnummer();

processVars.ext\_schadenshoehe = Double.*valueOf*(meldung. getSchadenshoehe());

ProcessInstance instance = runtimeService.startProcessInstanceByMessage(  
 "schadensmeldungKfzGlasbruch", processVars.createVariableMap());

**return** instance.getId();

}

}

## Ausschluss von False Positives

Zum Ausschluss von Scheinfehlermeldungen kann eine Ignore-Datei angelegt werden. Sie muss ".ignoredIssues" lauten und unter „src/main/resources“ im zu analysierenden Projekt abgelegt sein.

Dort können dann die IDs der Fehler eingetragen werden, die in Zukunft nicht mehr berücksichtigt werden sollen. Dies muss zeilenweise erfolgen. Zeilenkommentare werden mit „#“ eingeleitet.

**Beispiel .ignoredIssues**:

# Sonderfall 1

8d04f2e77a7d282c521098ab947ac060

# Sonderfall 2

5bafe978d328a216bd0bf4935ca88a8a

# Prüfungen

In diesem Kapitel sind die einzelnen Prüfungsfunktionen im Detail beschrieben.

## Java Delegate Checker

Mit dem *Java Delegate Checker* lässt sich in Prozessmodellen überprüfen, ob folgende Bedingungen für Service Tasks, Send Tasks, Receive Tasks, Script Tasks oder Business Rule Tasks zutreffen:

* Keine Implementierung angegeben
* Klasse als Implementierung angegeben, aber nicht gefunden
* Klasse implementiert nicht die Schnittstelle *JavaDelegate*

Die Regel ist wie folgt zu konfigurieren:

<rule>

<name>JavaDelegateChecker</name>

<state>true</state>

</rule>

Über *state = true*, kann die Prüfung aktiviert und über *state = false* deaktiviert werden.

Das Plugin kann folgende Nachrichten generieren, welche mit Herkunft und Kritikalität versehen sind:

* task '%taskName%' with no code reference yet

Es ist keine Referenz auf Quellcode hinterlegt worden. Damit die Warnung nicht mehr auftaucht, ist eine Implementierung zu hinterlegen.

* class for task '%taskName' not found

Es ist eine Java-Klasse angegeben worden, die nicht existiert. Es ist zu prüfen, ob sich der Package-Pfad z.B. im Rahmen eines Refactorings geändert hat oder die Datei versehentlich gelöscht wurde.

* class for task %taskName% does not implement interface JavaDelegate

Es ist eine Java-Klasse angegeben worden, die nicht die Schnittstelle JavaDelegate implementiert. Die Java-Klasse ist insofern zu überarbeiten, als dass diese die Schnittstelle implementiert.

## Embedded Groovy Script Checker

Der *Embedded Groovy Script Checker* überprüft eingebettete Skripte in Listenern und Script Tasks auf Validität. Dafür prüft er folgende Bedingungen ab:

* Es ist kein Skriptformat angegeben
* Es ist kein Skriptinhalt angegeben
* Nur für Groovy: Der Skriptinhalt passt nicht zum Skriptformat (Syntax-Prüfung)

Die Regel ist wie folgt zu konfigurieren:

<rule>

<name>EmbeddedGroovyScriptChecker</name>

<state>true</state>

</rule>

Das Plugin kann folgende Nachrichten generieren, welche zusätzlich mit Herkunft und Kritikalität versehen sind:

* there is no script format for given script

Es ist kein Skriptformat für ein eingebettetes Skript hinterlegt. Es muss für das Skript ein Skriptformat hinterlegt werden.

* there is no script content for given script format

Es existiert ein Skriptformat, aber kein eingebettetes Skript. Es muss ein Skript hinterlegt werden.

* there is an empty script reference

Es ist kein Skript hinterlegt worden. Es muss ein Skript angegeben werden, damit der Fehler verschwindet.

* [syntax checker message]

Die Syntax des Skriptes ist nicht valide.

## Versioning Checker

Der *Versioning Checker* überprüft Referenzen in Service Tasks, Script Tasks, Business Rule Tasks, Send Tasks, Listenern und Message Events auf die Angabe einer versionierten Java-Implementierung. Wenn eine versionierte Implementierung referenziert wird, wird überprüft, ob diese aktuell ist.

Die Regel ist wie folgt zu konfigurieren:

<rule>

<name>VersioningChecker</name>

<state>true</state>

<settings>

<setting name="versioningSchemaClass">([^\_]\*)\_{1}([0-9][\_][0-9]{1})\.(java|groovy)</setting>

</settings>

</rule>

Die Einstellung „versioningSchemaClass“ beinhaltet ein Datei-Namensschema (regulärer Ausdruck) für versionierte Klassen. Dieses wird für die Erkennung von solchen Klassen benötigt, hier z.B. *<Name>\_<Majorversion>\_<Minorversion>.java*.

Das Plugin kann folgende Nachrichten generieren, welche zusätzlich mit Herkunft und Kritikalität versehen sind:

* class reference is deprecated or file with version doesn't exist

Eine referenzierte Java-Klasse ist entweder veraltet oder sie ist nicht versioniert. Sofern keine versionierte Klasse hinterlegt ist, ist diese an die Versionsnamenskonvention anzupassen. Ist die referenzierte Klasse veraltet, ist die Referenz auf die aktuelle Version zu ändern.

* script reference is deprecated or file with version doesn't exist

Ein referenziertes Skript ist entweder veraltet oder es ist nicht versioniert. Sofern kein versioniertes Skript hinterlegt ist, ist dieses an die Versionsnamenskonvention anzupassen. Ist das referenzierte Skript veraltet, ist die Referenz auf die aktuelle Version zu ändern.

* bean reference is deprecated or file with version doesn't exist

Ein referenzierte Spring-Bean ist entweder veraltet oder sie ist nicht versioniert. Sofern keine versionierte Spring-Bean hinterlegt ist, ist diese an die Versionsnamenskonvention anzupassen. Ist die referenzierte Bean veraltet, ist die Referenz auf die aktuelle Version zu ändern.

## DMN Task Checker

Diese Funktion ist als Proof-Of-Concept implementiert und daher als „**experimentell**“ zu betrachten.

Der *DMN Task Checker* überprüft, ob für „Business Rule Tasks“ eine Geschäftsregel referenziert wird.

Die Regel ist wie folgt zu konfigurieren:

<rule>

<name>DmnTaskChecker</name>

<state>true</state>

</rule>

Das Plugin kann folgende Nachricht generieren, welche zusätzlich mit Herkunft und Kritikalität versehen ist:

* business rule task with dmn implementation without a decision ref

Es ist zu einem Business Rule Task keine DMN-Entscheidung referenziert worden. Damit die Warnung verschwindet, ist eine gültige Referenz zu hinterlegen.

## Process Variables Model Checker

Diese Funktion ist als Proof-Of-Concept implementiert und funktioniert nur sofern, auch die Prozessvariablen im Quellcode korrekt erkannt werden.

Der *Process Variables Model Checker* überprüft ein Modell auf Datenflussanomalien. Darunter befinden sich Anomalien nach den Mustern DD („Überschrieben“), DU („Definiert-Gelöscht“) und UR („Undefiniert-Gelesen“).

Die Regel ist wie folgt zu konfigurieren:

<rule>

<name>ProcessVariablesModelChecker</name>

<state>true</state>

</rule>

Das Plugin kann folgende Nachricht generieren, welche zusätzlich mit Herkunft, Kritikalität, betroffener Prozessvariable (*variable*), Anomalie-Typ (*anomaly*) und Anomalie-Pfaden (*paths*) versehen ist:

* process variable creates an anomaly (compare %Chapter%,%ElementFieldType%)

Die Meldung bedeutet, dass eine Anomalie für eine konkrete Prozessvariable gefunden wurde. Damit diese behoben werden kann, sind die Pfade je nach Anomalie-Typ zu analysieren. %Chapter% und %ElementFieldType% enthalten Angaben, wo im Camunda Modeler die Prozessvariable zum konkreten BPMN-Element zu finden ist.

Bei einer UR-Anomalie wird eine Variable gelesen, die zuvor nirgendwo definiert worden ist. Es ist möglich, dass der Entwickler übersehen hat die Variable vorher zu initialisieren. Eine mögliche Lösung ist es, die Variablendefinition zu ergänzen.

Bei einer DU-Anomalie wird eine Variable zwar definiert, aber ungenutzt gelöscht. Es ist möglich, dass der Entwickler versehentlich die falsche Variable gelöscht hat. Dies ist weiter zu überprüfen.

Bei einer DD-Anomalie wird eine Variable überschrieben ohne zuvor genutzt worden zu sein. Es ist möglich, dass der Entwickler sich über diesen Zusammenhang nicht bewusst war. Das Überschreiben der Variable ist zu entfernen.

## Process Variables Name Convention Checker

Der *Process Variables Name Convention Checker* überprüft, ob definierte Prozessvariablen interne und externe Namenskonventionen erfüllen.

Die Regel ist wie folgt zu konfigurieren:

<rule>

<name>ProcessVariablesNameConventionChecker</name>

<state>true</state>

<elementConventions>

<elementConvention>

<name>internal</name>

<!-- field types for process variables -->

<elementFieldTypes excluded=*"true"*>

<elementFieldType>Class</elementFieldType>

<elementFieldType>ExternalScript</elementFieldType>

<elementFieldType>DMN</elementFieldType>

<elementFieldType>DelegateExpression</elementFieldType>

</elementFieldTypes>

<pattern>int\_[a-zA-Z]+</pattern>

</elementConvention>

<elementConvention>

<name>external</name>

<!-- field types for process variables -->

<elementFieldTypes>

<elementFieldType>Class</elementFieldType>

<elementFieldType>ExternalScript</elementFieldType>

<elementFieldType>DMN</elementFieldType>

<elementFieldType>DelegateExpression</elementFieldType>

</elementFieldTypes>

<pattern>ext\_[a-zA-Z]+</pattern>

</elementConvention>

</elementConventions>

</rule>

Für interne und externe Namenskonventionen sind sogenannte *element conventions* zu definieren. Eine ElementConvention besteht aus einem einem Namen, einem zu analysierenden Elementfeld-Typen und einem regulären Ausdruck für die Namenskonvention (das Pattern). Ein Elementfeld-Typ bezeichnet den Typ des zu prüfenden Inhalts. Es gibt dafür folgende Werte: *Class, FormField, Expression, DelegateExpression, ResultVariable, CalledElement, CaseRef, CollectionElement, ElementVariable, LoopCardinality, CompletionCondition, InlineScript, ExternalScript, Assignee, CandidateUsers, CandidateGroups, DueDate, FollowUpDate, DMN, CamundaIn* und *CamundaOut*. Das Attribut excluded=“true“ besagt, dass alle Elementfeld-Typen, außer die hinterlegten, überprüft werden sollen.

Das Plugin kann folgende Nachricht generieren, welche zusätzlich mit Herkunft, Kritikalität, und betroffener Prozessvariable (*variable*) versehen ist:

* process variable is against the naming convention '%conventionName%' (compare %Chapter%, %ElementFieldType%)

Die Meldung gibt an, dass eine Prozessvariable gegen eine bestimmte Namenskonvention verstößt. %conventionName% enthält den Namen der Konvention, welche innerhalb der Regel definiert worden ist. %Chapter% und %ElementFieldType% enthalten Angaben, wo im Camunda Modeler die Prozessvariable zum konkreten BPMN-Element zu finden ist. Als Lösung ist die Variable an die Konvention anzupassen.

## Task Naming Convention Checker

Der *Task Naming Convention Checker* überprüft, ob Tasks einer vormals definierten Namenskonvention folgen.

Die Regel ist wie folgt zu konfigurieren:

<rule>

<name>TaskNamingConventionChecker</name>

<state>false</state>

<elementConventions>

<elementConvention>

<name>convention</name>

<pattern>[A-ZÄÖÜ][a-zäöü\\\-\\\s]+</pattern>

</elementConvention>

</elementConventions>

</rule>

Über *state = true*, kann die Prüfung aktiviert und über *state = false* deaktiviert werden. Als *element convention* wird dafür ein regulärer Ausdruck definiert, auf welche dessen Übereinstimmung die Tasknamen überprüft werden.

Das Plugin kann folgende Nachrichten generieren, welche zusätzlich mit Herkunft und Kritikalität versehen sind:

* task name must be specified

Es ist kein Aufgabenname angegeben worden. Die Lösung ist es, den Aufgabennamen zu ergänzen.

* task name '%taskName%' is against the naming convention

Der Aufgabenname widerspricht der Namenskonvention in der Konfiguration. Die Lösung ist es, den Aufgabennamen an die vormals definierte Konvention anzupassen.

1. https://docs.camunda.org/javadoc/camunda-bpm-platform/7.3/org/camunda/bpm/engine/delegate/VariableScope.html [↑](#footnote-ref-1)