Solibri API-Regelentwicklung – Programmierung von Prüfschablonen für Solibri

TU Wien – Digitaler Bauprozess

Simon Fischer, Daniel Pfeiffer,
Markus Kallinger, Patrick Loibl,
Harald Urban, Christian Schranz

10. Juli 2024

Allgemein

Das Softwareunternehmen Solibri bietet eine Programmierschnittstelle (API – *Application Programming Interface*), die es ermöglicht, die Software Solibri basierend auf der Programmiersprache Java um eigene Funktionen zu erweitern. Dieses Schulungsbuch beschreibt die *Rule-API* von Solibri, die zur Erstellung eigener Regelschablonen dient. Das Schulungsbuch wurde für die API-Version 9.13.0 erstellt. Neuere Versionen können in ihrer Funktionalität leicht abweichen. Die Beschreibung zielt auf Einsteiger in das Thema Regelprogrammierung mit Solibri und bietet neben den Grundlagen auch praxisrelevante Beispiele. Das Dokument ist jedoch kein Schulungsbuch für Solibri oder Programmieren im Allgemeinen. Hier werden Grundkenntnisse vorausgesetzt. Folgende Themengebiete sind auf den nachfolgenden Seiten behandelt:

- Grundlagen/Vorwissen: Auflistung und Beschreibung von benötigtem Vorwissen
- Setup: Installation und Einstieg in die erforderlichen Programme
- Java-Project: Inhalte eines Java-Projects zur Regelprogrammierung
- Prüfregelaufbau: Aufbau und Ablauf einer Prüfregel
- UI-Elemente: Möglichkeiten zur Erstellung einer Benutzeroberfläche
- Kochrezept Prüfregel: Methodische Herangehensweise zur Prüfregelentwicklung
- 2 Beispielregeln
- Weiterführende Themen: Wissenswertes zur fortgeschrittenen Prüfregelprogrammierung

Inhaltsverzeichnis

1 Grundlagen/Vorwissen

Dieses Kapitel gibt einen Überblick über benötigte Grundlagen bzw. benötigtes Vorwissen, das nicht Teil dieses Schulungsbuchs ist, jedoch zum Verständnis des Inhalts vorausgesetzt wird. Die Autoren empfehlen deshalb, sich mit folgenden Themengebieten über andere Kanäle vorab vertraut zu machen.

1.1 Java

Solibri setzt in der API zur Prüfregel-Programmierung auf die Programmiersprache *Java*, weshalb Grundlagenwissen auf diesem Gebiet als Voraussetzung notwendig ist. Die API ist wie ein eigenes *Java-Package* zu verstehen, die eigene Klassen, Methoden, Interfaces, etc. definiert. Wichtige Kenntnisse sind der Umgang mit verschiedenen Datentypen (primäre und abstrakte Datentypen) und die objektorientierte Programmierung (Klassen, Objekte, Interfaces, etc.). Darüber hinaus ist die Bearbeitung von Gruppen von Elementen für die Erstellung von Prüfregeln von großer Bedeutung. Java bietet dafür mit dem *Collection-Framework* verschiedene Datenstrukturen (z. B. *List*, *Set*, *Map*), die in der Solibri-API häufig verwendet werden. Die notwendigen Installationen zur Java-Programmierung sind in Kapitel 2 enthalten.

1.2 IDE – Integrated Development Environment

Die IDE bzw. integrierte Entwicklungsumgebung ermöglicht als graphische Oberfläche die Erstellung bzw. Programmierung von Prüfregeln. Eine IDE unterstützt und erleichtert durch ihren Aufbau und ihre Funktionen das Programmieren. Solibri empfiehlt *Eclipse* als Entwicklungsumgebung. Im Kapitel 2 ist eine Erklärung zur Installation von *Eclipse* enthalten.

1.3 Solibri

Um die eigens programmierten Prüfschablonen in Solibri anwenden und testen zu können, sind zwei Bereiche wesentlich: *Ruleset Manager* und *Checking View*. Der *Ruleset Manager* dient zur Verwaltung von Regelsätzen. Dazu gehört die Befüllung und Kombination von Prüfschablonen. Im *Ruleset Manager* werden alle von Solibri bereitgestellten sowie eigens programmierten Prüfschablonen angezeigt. Um eine Prüfung durchführen zu können, muss zunächst eine Regelsatz geladen order neu erstellt werden, welchem man dann die jeweiligen Prüfregeln zuordnen kann. Anschließend können die erforderlichen Eingaben für die Prüfregel getätigt werden. Die *Checking View* dient zur Ausführung der Prüfregeln. Dort werden Prüfregeln gestartet sowie ihre Ergebnisse angezeigt. Zur genaueren Betrachtung der Ergebnisse und Analyse der programmierten Prüfregel, sollte weiters ein gewisses Grundverständnis über die Navigation in Solibri bekannt sein.

2 Setup

2.1 Solibri

Um mit der Programmierung eigener Regeln beginnen zu können, muss *Solibri Office* installiert sein. Für den Zugriff auf das Office-Produkt benötigt man eine Lizenz. Sobald eine Lizenz vorliegt, ist über den nachfolgende Link ein Login im *Solibri Solution Center* möglich (siehe Abb. 2.1):

`https://solution.solibri.com/`

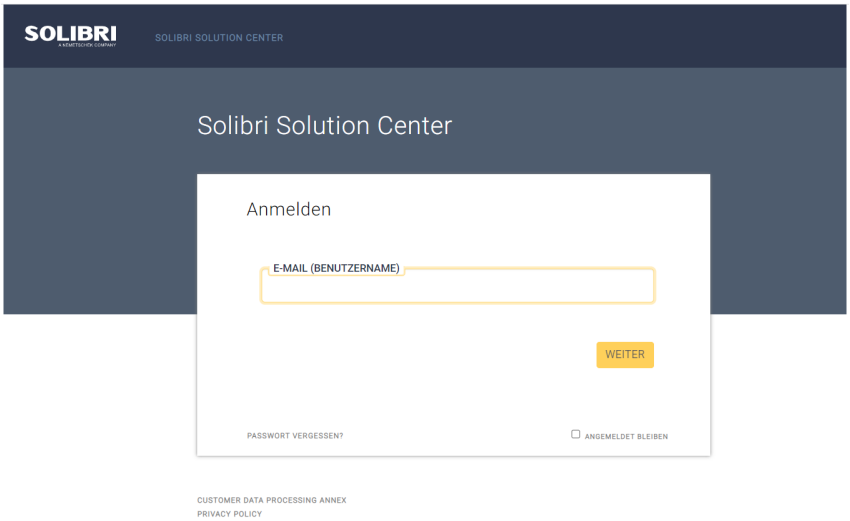

Abb. 2.1: Login - Solibri Solution Center

Anschließend kann die gewünschte Software (Solibri-Office) für das vorhandene Betriebssystem ausgewählt und heruntergeladen werden (siehe Abb. 2.2).

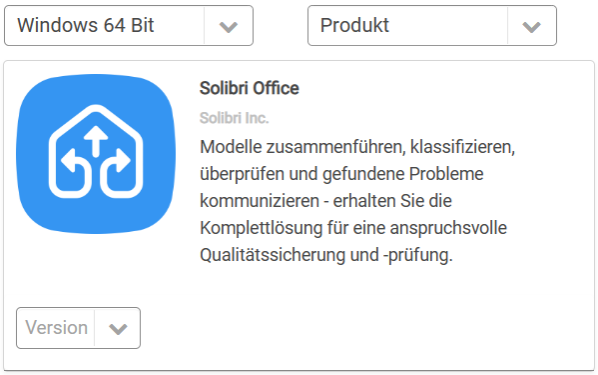

Abb. 2.2: Download Solibri Office

Um fortfahren zu können, muss Solibri installiert werden. Es empfiehlt sich, um später keine Schwierigkeiten mit den Speicherorten zu haben, Solibri an dem vorgeschlagenen Ort zu installieren (`C:/Program Files/Solibri/SOLIBRI`). Die Anmeldung erfolgt mit der gleichen E-Mail-Adresse

wie zu vor beim *Solibri Solution Center*. Wichtig dabei ist, dass unter den drei verschiedenen Produkten *Solibri Office* ausgewählt wird (siehe Abb. 2.3), da ansonsten die Prüfregeln nicht zur Verfügung stehen.

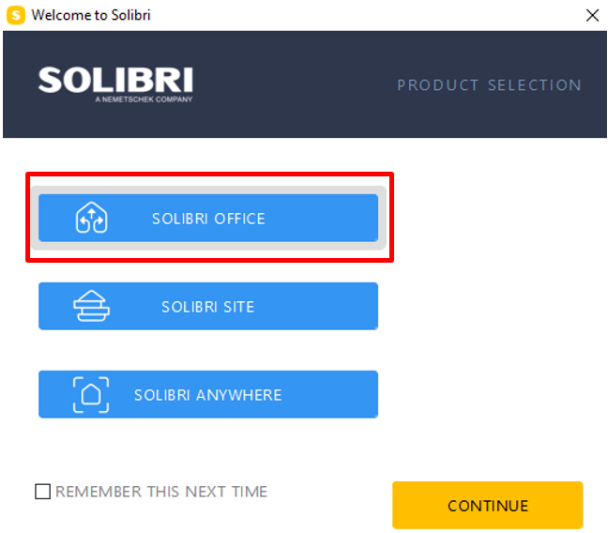

Abb. 2.3: Start von Solibri Office

2.2 zusätzliche Software

Da Solibri in Java programmiert ist und die API somit ebenfalls auf Java basiert, müssen das sogenannte *Java SE Development Kit* (SDK) und eine Entwicklungsumgebung heruntergeladen und installiert werden. Um die richtige Version zu finden empfiehlt es sich, den aktuellen Anweisungen von Solibri auf der *Solibri Developer Plattform* zu folgen. Dabei ist zu beachten, dass es für jede Solibri API-Version eine eigene Version der *Solibri Developer Plattform* gibt. Um stets die aktuellste Version abzurufen, ist folgendem Link zu folgen:

`https://solibri.github.io/Developer-Platform/index.html`

Anschließend kann einerseits die gewünschte Version ausgewählt werden, oder andererseits durch direkte Navigation mithilfe der oberen Reiter automatisch die *Solibri Developer Plattform* der aktuellsten API Version aufgerufen werden (siehe Abb. 2.4).

Über den Reiter *Getting Started* kommt man zu einer Erklärung von Solibri, wie und welche Softwareanwendungen heruntergeladen werden müssen. Im Falle der Java SDK wird auf *AdoptOpenJDK* verwiesen. Der angegebene Link führt zu einem Downloadmenü, indem Windows als Betriebssystem voreingestellt ist. Andere Betriebssysteme sind unter *Other platforms* zu finden (siehe Abb. 2.5). Als zweiter Schritt ist die Entwicklungsumgebung *Eclipse IDE* nach der gegebenen Anleitung herunterzuladen. Nach dem Öffnen der EXE-Datei wird der Eclipse-Installer gestartet. Im ersten Fenster ist *Eclipse IDE for Java Developers* auszuwählen. Im zweiten Fenster belässt man die Standardeinstellungen und startet die Installation.

Die Seite *Getting Started* bietet im Anschluss eine Erklärung, wie man mit der Regelentwicklung startet. Diese Vorgehensweise wird in diesem Dokument im nachfolgenden Abschnitt erklärt.

2.3 Erstellung einer Regel

Um die Erstellung der Regel so einfach wie möglich zu gestalten, hat Solibri eine Regelvorlage (Rule-Template) erstellt, welche dem/der NutzerIn den Einstieg vereinfacht. Diese wird entweder

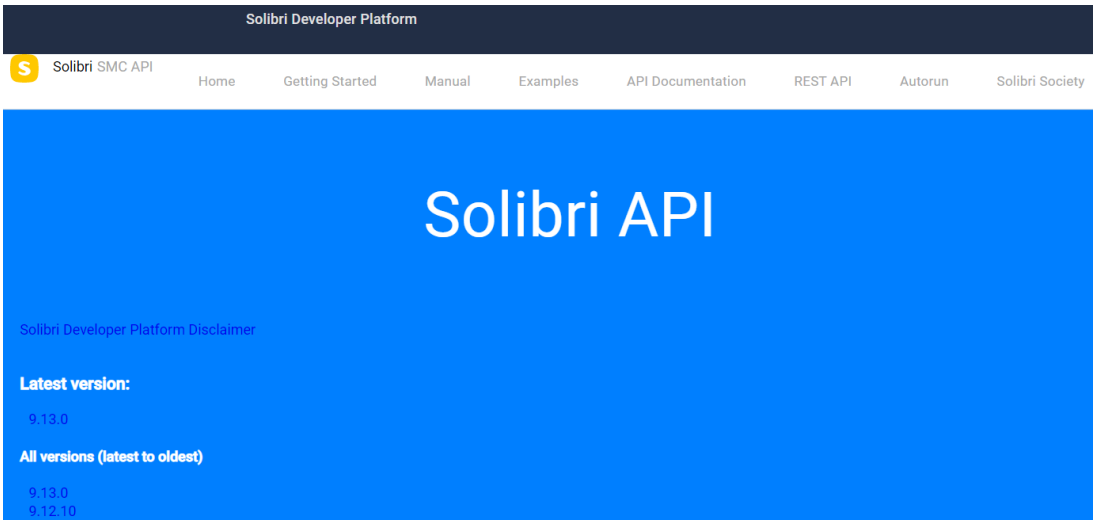

Abb. 2.4: Versionsunabhängige Startseite der Solibri Developer Plattform

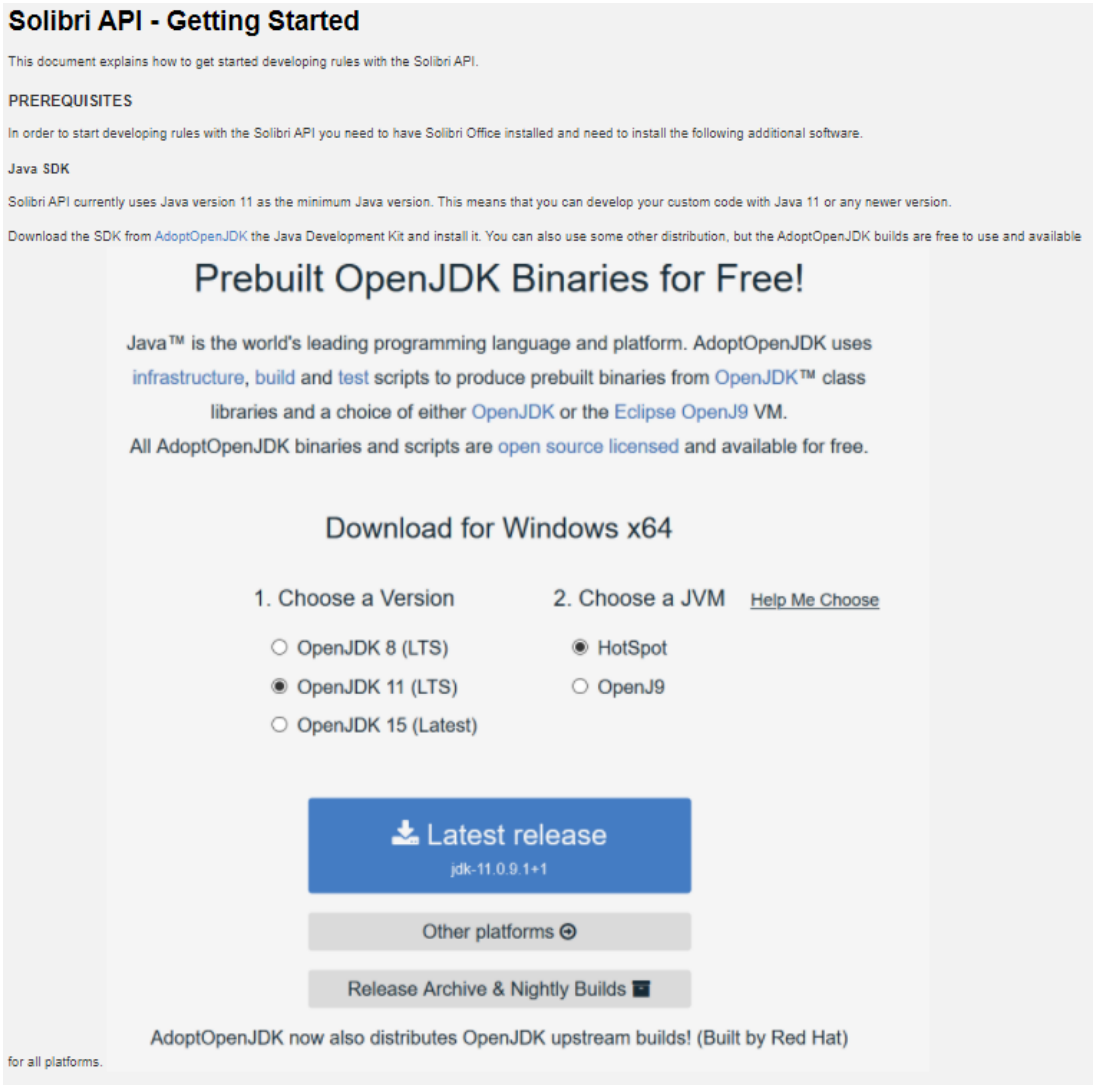

Abb. 2.5: Solibri Developer Plattform – Getting Started

unter dem Reiter *Examples* beim Unterpunkt *Rule Examples* oder unter dem Reiter *Getting Started* beim Unterpunkt *Environment Setup* gefunden und kann dort für die jeweils aktuelle Version heruntergeladen werden.

2.3.1 Die Vorlage in Eclipse öffnen

Zuerst muss die Vorlagedatei an einem beliebigen Ort entpackt werden. Der Inhalt des Vorlageordners ist in Abb. 2.6 dargestellt und wird im Kapitel 3 näher beschrieben. Soll der Name des Projektes oder der Firmenname geändert werden, muss an dieser Stelle das Pom-File im Texteditor geöffnet und an den entsprechenden Stellen die Bezeichnungen angepasst werden (siehe Abb. 2.7). Die *<groupId>* verändert den Namen des Projekts in *Eclipse*. Eine Änderung muss jedoch vor Import erfolgen. Nachträglich kann der Projektname nur mehr in der IDE selbst geändert werden. Mithilfe von *<artifactId>* kann der Name der beim Export generierten *.jar*-Datei definiert werden. Die Elemente *<name>* und *<version>* stellen sichtbare Angaben in Solibri dar. Ersteres definiert den Namen des Ordners in Solibri, welcher die einzelnen Regeln enthält. Das zweite Element stellt eine einfache Art der Versionierung dar.

Sollte Solibri nicht wie unter Abschnitt 2.1 empfohlen, im vorgeschlagenen Verzeichnis installiert worden sein, so muss im gleichen Pom-File das entsprechende Verzeichnis im Element `<smc-dir>` eingegeben werden (siehe Abb. 2.8).

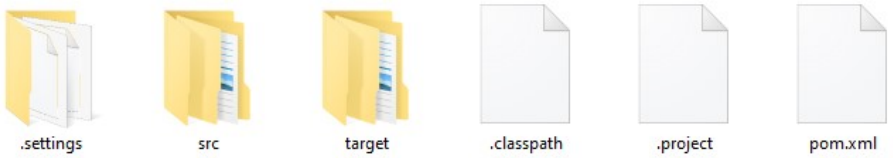

Abb. 2.6: Inhalt des Vorlageordners

Abb. 2.7: Vorlageregel – Bezeichnungen ändern

Wurden all diese Änderungen vorgenommen, kann wieder in *Eclipse* gewechselt werden und unter *File -> Import -> Maven -> Existing Maven Projects* das Fenster in Abb. 2.9 geöffnet werden. Unter *Browse...* kann nun der Speicherort der Vorlageregel gesucht und diese anschließend importiert werden. Mit dem Klick auf *Finish* wird das Projekt in *Eclipse* im *Package Explorer* hinzugefügt. Für alle weiteren Projekte, die zur Gliederung von Regeln erstellt werden sollen, gilt die gleiche Vorgangsweise.

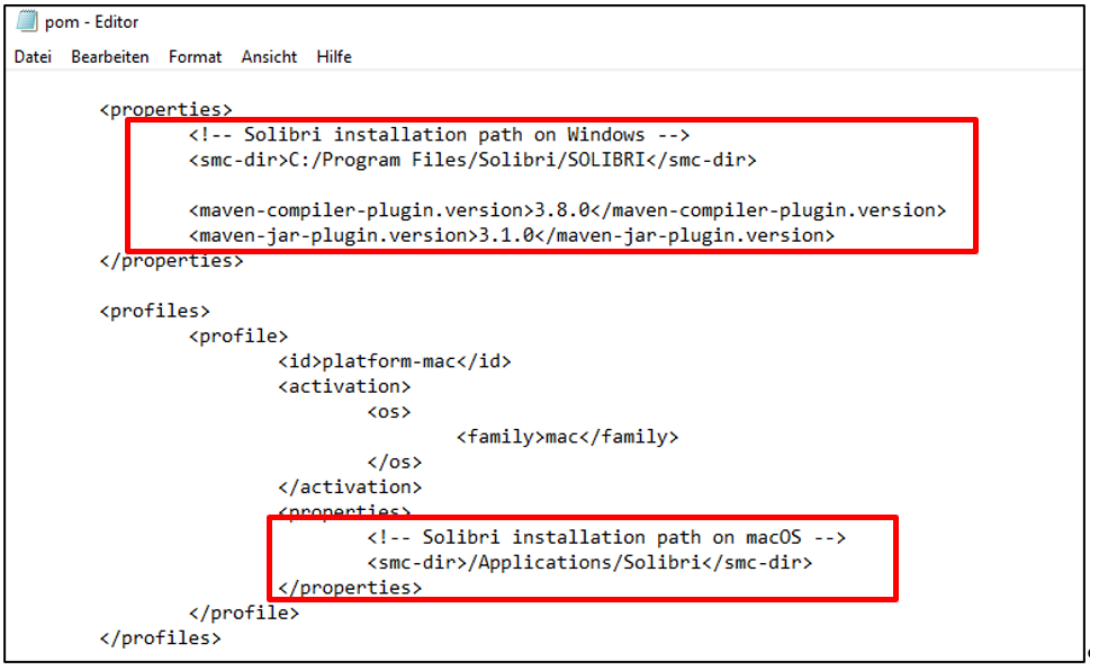

Abb. 2.8: Vorlageregel – anderes Installationsverzeichnis

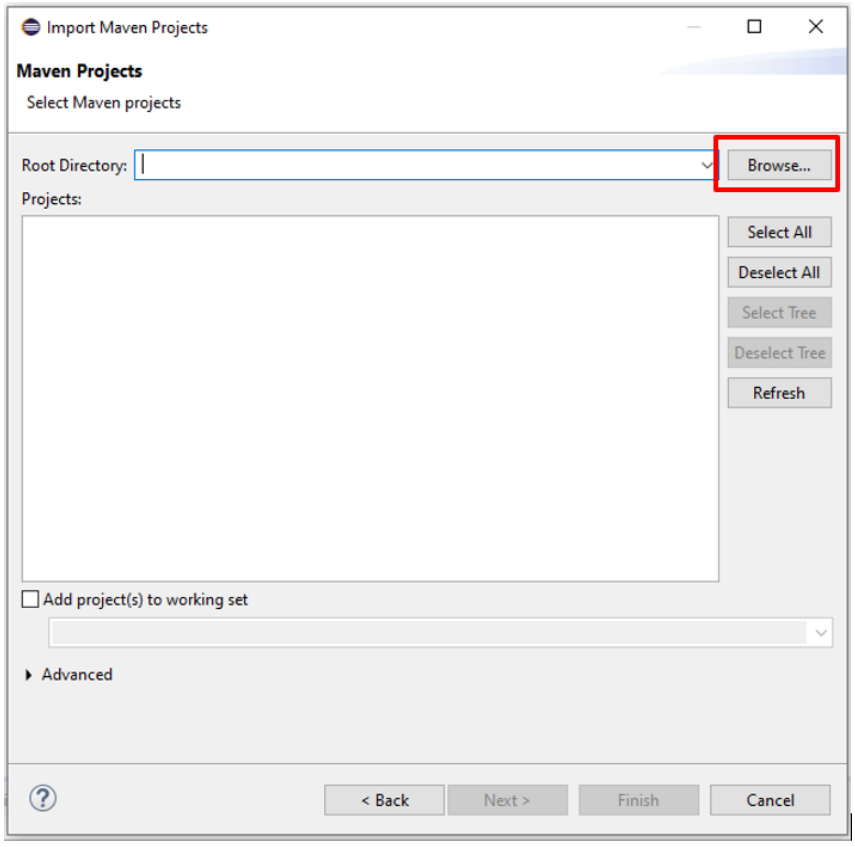

Abb. 2.9: Vorlageregel - Import Maven Projects

2.3.2 Regeln in Solibri einfügen

Nachdem die Regel in *Eclipse* entsprechend programmiert wurde, muss diese als *Maven install* in Solibri eingefügt werden. Das Build-Tool *Maven* der *Apache Software Foundation* dient dazu, das Java-Project in eine in Solibri ausführbare Struktur zu bringen. An diesem Punkt wird noch einmal darauf hingewiesen, dass das Installationsverzeichnis von Solibri im Pom-File richtiggestellt werden muss (siehe Abb. 2.8). Danach wird durch einen Rechtsklick auf den *Projektordner -> Run As -> 6 Maven install* die Prüfregel in Solibri hinzugefügt (siehe Abb. 2.10). Dabei wird im Solibri-Hauptverzeichnis eine *.jar*-Datei im Ordner *lib* (*Library*) erstellt. Sollte Solibri, wie in Abschnitt 2.1 empfohlen, im Standardverzeichnis installiert worden sein, befindet sich der Speicherort unter `C:\Program Files\Solibri\SOLIBRI\lib`. Wurde ein anderer Speicherort gewählt, muss trotzdem der Ordner `*\Solibri\SOLIBRI\lib` aufgerufen werden. Möchte man einen Projektordner wieder aus Solibri entfernen, muss die *.jar*-Datei gelöscht werden. Generell ist jedoch innerhalb des Bibliotheksverzeichnisses Vorsicht geboten. Wird eine falsche Datei gelöscht, kann das die Funktionalität von Solibri stören und die Software muss neu installiert werden.

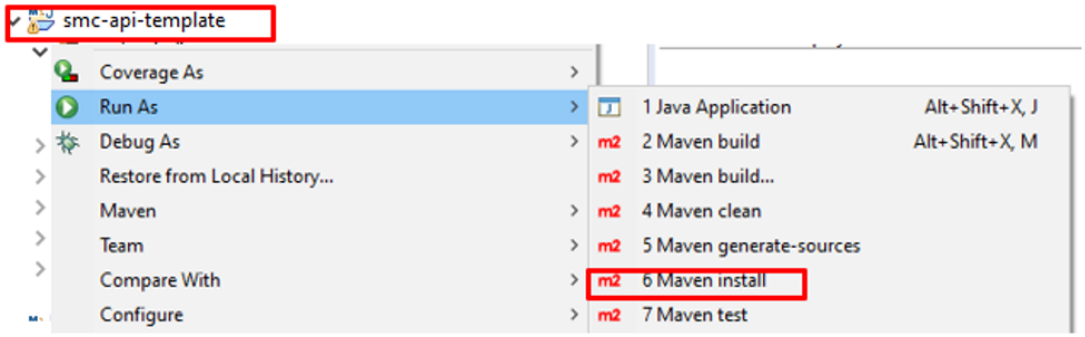

Abb. 2.10: Prüfregeln in Solibri einfügen

2.4 Übergabe von Regeln an Dritte

Prüfregeln können an Dritte über die beim *Maven install* erstellte *.jar*-Datei weitergegeben werden. Die Datei befindet sich im Bibliotheksverzeichnis im Solibri-Hauptverzeichnis (siehe Abschnitt 2.3.2). Generell ist zu beachten, dass es keine einzelne Regel weitergegeben wird, sondern alle Regeln die in *Eclipse* in diesem Projektordner erstellt wurden.

2.5 Importieren von API-Regeln ohne Eclipse

Wurde die *.jar*-Datei wie in Abschnitt 2.4 kopiert und weitergegeben, so sollte diese von Dritten im Verzeichnis `C:\Users\Public\Solibri\SOLIBRI\API` abgelegt werden. Beim nächsten Start von Solibri steht der Regelsatz dann zur Verfügung. Grundsätzlich könnte die Datei auch in das Bibliotheksverzeichnis `C:\Program Files\Solibri\SOLIBRI\lib` bzw. `*\Solibri\SOLIBRI\lib` abgelegt werden, wie es im Falle des *Maven install* direkt aus *Eclipse* der Fall ist. Das hat jedoch einen entscheidenden Nachteil: Dieses Verzeichnis wird bei Updates überschrieben, wodurch die eigens erstellten Dateien erneut eingefügt werden müssten.

2.6 Solibri Developer Platform

Die *Solibri Developer Platform* bietet neben der Erklärung zum Installation der erforderlichen Software und einer Einstiegsanleitung weitere wichtige Informationen und Hilfestellungen. Die

wichtigste Information ist die Verlinkung zur aktuellen Dokumentation der API, die über den Reiter *API Documentation* abrufbar ist. Die Dokumentation enthält alle Interfaces und Klassen der API und beschreibt deren Funktionalität allgemein sowie alle vorhandenen Methoden.

Weiters enthält die *Solibri Developer Platform* einen Link zur *Solibri Society*. Dabei handelt es sich um ein Forum der Solibri-Community. Hier können Fragen zu aktuellen Problemen gestellt werden. Neben anderen Nutzern helfen hier auch Angestellte von Solibri bei Problemen. Um auf das Forum zugreifen zu können, muss ein kostenloser Account erstellt werden.

Abschließend ist der Reiter *Examples* besonders hilfreich. Als Einstieg für die Prüfregelprogrammierung empfiehlt es sich, die Beispielregeln (*Examples*) von Solibri Schritt für Schritt durchzugehen, um so die Struktur der Prüfregeln zu verstehen. Außerdem kann so ein erster Einblick in die Anwendungsmöglichkeiten der API-Regeln gewonnen werden. Darüber hinaus ist auch die Regelvorlage unter den Beispielen zu finden.

3 Java-Project – Ordnerstruktur

In diesem Kapitel wird die Ordnerstruktur des Projekts anhand der Regelvorlage (*rule-template*) von Solibri erklärt. Die Ordnerstruktur wird sowohl in *Eclipse* als auch im Dateimanager erklärt.

3.1 Ordnerstruktur in Eclipse

Wurde der Projektordner *rule-template*, wie in Kapitel 2.3 beschrieben, richtig in *Eclipse* importiert, erscheint ein Projekt mit dem Namen *smc-api-template* (siehe Abb. 3.1). Das Projekt enthält sechs Unterordner und das entsprechende Pom-File. Die ersten beiden Unterordner, *src/main/java* und *src/main/resources* sind für die Programmierung von Prüfregeln am bedeutendsten, da in ihnen einerseits der Programmcode, der den Aufbau und die Funktion der Regel beschreibt, sowie andererseits die Ressourcen in Form von Bildern und Text enthalten sind. Die nächsten zwei Unterordner enthalten Java-Bibliotheken. Das sind einerseits alle Bibliotheken, die in der installierten Java Version enthalten sind, und andererseits zusätzliche Pakete, die mittels *Maven Dependencies* im Pom-File dem Projekt hinzugefügt wurden. Diese beiden Ordner werden automatisch aktualisiert und sollen nicht manuell bearbeitet werden. Sie sind daher vorerst nicht von Bedeutung. Gleiches gilt für die nächsten beiden Unterordner: *src* und *target*. Sie werden beim Ausführen von *Maven install* automatisch aktualisiert. Eine wesentliche Rolle für die Prüfregel Programmierung spielt wiederum das Pom-File. Eine genauere Erklärung dazu folgt in Abschnitt 8.6.

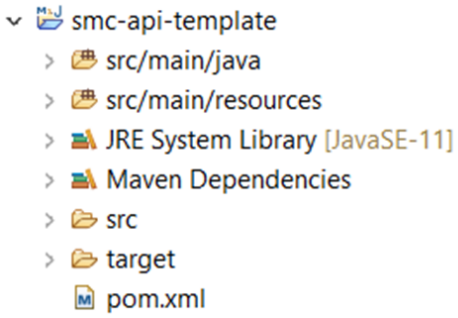

Abb. 3.1: Inhalt des Projekts

Der Unterordner *src/main/java* beinhaltet das Package *com.solibir.rule* in welchem sich die Java-Datei mit dem Code der Prüfregel befindet (siehe Abb. 3.2).

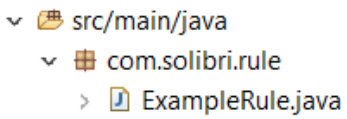

Abb. 3.2: Unterordner *java*

Im Unterordner *src/main/resources* sind die verschachtelten Ordner *com*, *solibri* und *rule* zu finden. Im innersten Ordner *rule* sind die Properties-Dateien für die verschiedenen Sprachen enthalten (siehe Abb. 3.3). Die Properties-Dateien sind regel-spezifisch und dienen zur Definition der angezeigten Inhalte in Solibri. Dazu gehören Metadaten wie der Regelname und die Version,

alle Texte und Bilder der Benutzeroberfläche der Regel sowie alle Texte der Ergebnisse der Regel. Die Auslagerung dieser Informationen aus der Java-Datei in eine Properties-Datei ermöglicht Mehrsprachigkeit. Dafür ist für jede gewünschte Sprache (nur jene Sprachen die Solibri unterstützt) eine eigene Properties-Datei anzulegen.

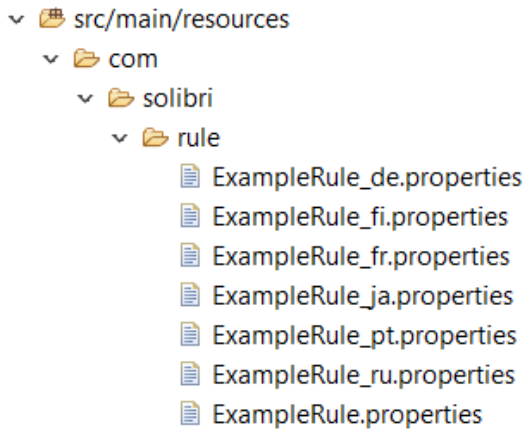

Abb. 3.3: Unterordner *resources*

3.2 Verwendung von Bilddateien

In der Benutzeroberfläche verwendete Bilder müssen ebenso im Projekt abgelegt werde, damit von den Properties-Dateien darauf verwiesen werden kann. Es ist zweckmäßig dazu einen neuen Ordner *images* im Verzeichnis der Properties-Dateien *src/main/resources/com/solibri/rule* anzulegen (siehe Abb. 3.4). Um in einer Properties-Datei auf ein Bild verweisen zu können, muss neben dem Dateinamen und Dateityp des Bildes auch der direkte Verweis auf den Ordner *images* vorhanden sein.

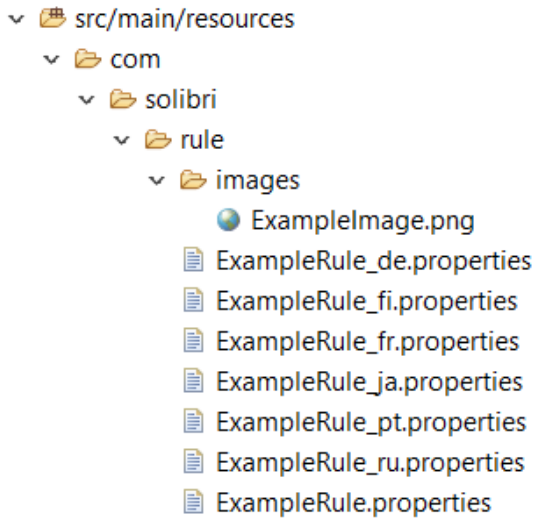

Abb. 3.4: Unterordner *resources* mit erstelltem Ordner *images*

3.3 Mehrere Regeln

Neben dem in den Abb. 3.2 und 3.3 gezeigten Aufbau ist es auch möglich, dass mehrere Regeln in einem Projekt gespeichert werden. Dazu können einfach mehrere Java-, Properties- und Bilddateien in den jeweiligen Verzeichnissen gespeichert werden (siehe Abb. 3.5 und 3.6). Dadurch sind die Regeln auch im *Ruleset Manager* in Solibri unter *Libraries* in einem Ordner gebündelt.

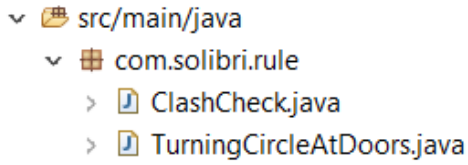

Abb. 3.5: Unterordner *java* mit den beiden Beispielregeln

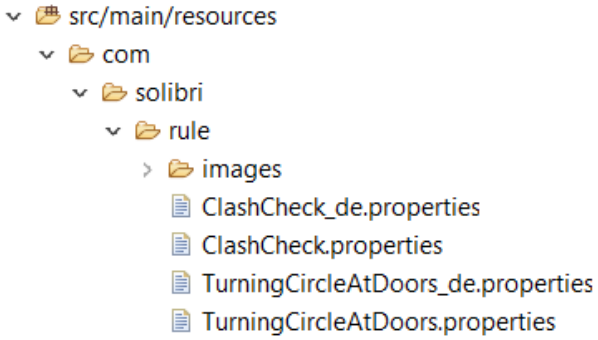

Abb. 3.6: Unterordner *resources* mit den Ressourcen der beiden Beispielregeln

3.4 Ordnerstruktur im Dateimanager

Neben der Entwicklungsumgebung kann die Ordnerstruktur auch im Dateimanager geöffnet und bearbeitet werden. Ausgehend vom Projektordner *rule-template* findet man darin die Ordner *.settings*, *src* und *target* sowie die Dateien *.classpath*, *.project* und *pom.xml* (siehe Abb. 3.7). Davon sind für die manuelle Bearbeitung nur der Ordner *src* und die Datei *pom.xml* relevant (gleich wie in *Eclipse*). In den anderen Ordnern und Dateien soll nichts verändert werden.

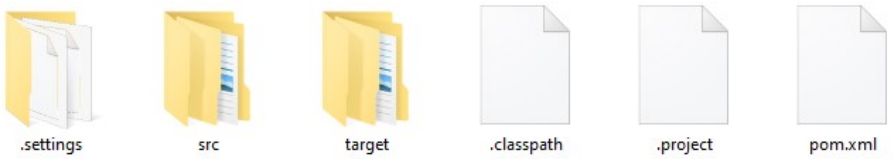

Abb. 3.7: Inhalt des Projektordners

Der Ordner *src* enthält die in *Eclipse* erklärte Ordnerstruktur für den Programmcode und die Ressourcen. Die Java-Datei ist somit ausgehend vom Projektordner in folgendem Verzeichnis zu finden:

```
rule-template\src\main\java\com\solibri\rule
```

Das Verzeichnis der Properties-Datei lautet wie folgt:

`rule-template\src\main\resources\com\solibri\rule`

Theoretisch können über den Dateimanager neue Ordner und Dateien angelegt, sowie Dateien bearbeitet werden. Dies kann jedoch auch direkt in *Eclipse* geschehen, was grundsätzlich zu bevorzugen ist. Eine Ausnahme bildet die Bearbeitung des Pom-File. Da das Pom-File auch für die Eindeutigkeit eines Java-Projekts verantwortlich ist, muss es in manchen Fällen über den Dateimanager bearbeitet werden, damit ein Projekt erst mittels *Maven install* in *Eclipse* importiert werden kann.

4 Prüfregel-Aufbau und - Ablauf

In diesem Kapitel wird der grobe Aufbau einer Prüfregel sowie deren Funktionsweise und der Ablauf der wesentlichen Methoden für den Check (*preCheck*, *check* und *postCheck*) erklärt. Ein genauerer Aufbau einer Prüfregel kann den jeweiligen Kapiteln über die beiden Beispielregeln entnommen werden.

Der Beginn jedes Regelcodes startet mit dem Aufruf des Packages `com.solibri.rule` und dem Importen der weiteren benötigten Packages. Diese können allgemein von Java ausgehen oder speziell für Solibri notwendig sein. Alle importieren Packages ergeben sich aus den im Programmcode verwendeten Befehlen von selbst. Wenn ein neuer Befehl verwendet wird und das entsprechende Package noch nicht importiert ist, kommt von *Eclipse* ein Hinweis darauf. Somit müssen jene nicht sofort zu Beginn der Programmiervorgangs importiert werden, sondern ergeben sich erst im Laufe der Zeit.

```
package com.solibri.rule;

import java.util.Collection;
import java.util.Collections;

import com.solibri.smc.api.checking.OneByOneRule;
import com.solibri.smc.api.checking.Result;
```

Als nächstes kommt der wesentliche Bestandteil der Prüfregel, nämlich die Klasse der Prüfregel, welche die Elternklasse *OneByOneRule* erweitert. Die Erweiterung einer Elternklasse entspricht dem bekannten Vorgehen der Vererbung bei Objektorientierung. Die Klasse *OneByOneRule* selbst implementiert das übergeordnete Interface *Rule*. Das Rule-Interface gibt grundsätzliche Funktionalitäten einer Prüfung in Solibri vor. Die Klasse *OneByOneRule* spezifiziert dieses für die Überprüfung von Komponenten, die einen Filter einzeln nacheinander passieren. Gemeinsam stellen sie einige grundlegende Methoden, wie *preCheck*, *check* und *postCheck* zur Verfügung. Im Grunde sollte gerade von Einsteigern für jede selbst programmierte Prüfregel die *OneByOneRule* als Elternklasse verwendet werden.

```
public final class ExampleRule extends OneByOneRule{
```

Die Klasse der Regel selbst lässt sich in drei Teile aufteilen. Im ersten Teil werden alle Klassenvariablen erstellt, im zweiten Teil erfolgt die eigentliche Prüfung und im dritten Teil wird der Aufbau des *User Interface* UI definiert. Solibri zeigt in ihren Beispielregeln, dass das UI und die Prüfung auf zwei Java-Klassen aufgeteilt werden kann. Es hat sich aber herausgestellt, dass das zu Problemen führen kann, weswegen empfohlen wird, alles in eine Klasse zu schreiben.

Zum ersten Teil der Klasse, zu den Klassenvariablen, gehört stets die Definition der *Ruleparameter*. *Ruleparameter* sind Elemente, die in der Benutzeroberfläche einer Prüfregel angezeigt und dort befüllt werden können. Im Code können die Werte der *Ruleparameter* (tatsächiche benutzerdefinierte Werte oder Verweise auf Modellinformationen) abgerufen und weiterverarbeitet werden. *Ruleparameter* sind also jener Bestandteil, der eine parametrische Programmierung von Prüfregeln ermöglicht. Der wichtigste *Ruleparameter* ist der *DefaultFilterParameter*. Er enthält alle Komponenten, für die Prüfung durchgeführt werden soll. Jede Prüfregel muss daher einen *DefaultFilterParameter* enthalten. Aufgrund seiner Wichtigkeit ist der Parameter in der *OneByOneRule* integriert und kann über die Methode `getDefaultFilterParameter()` von der

aktuellen Instanz der *OneByOneRule* (`this`) abgerufen werden.

```
private final FilterParameter defaultFilter = this.
    getDefaultFilterParameter();
```

Alle anderen *Ruleparameter* benötigen zur Erstellung den sogenannten *RuleParameterHandler*. Dieser ist jeweils eine Instanz des Interfaces *RuleParameters*. Der *RuleParameterHandler* wird für die aktuelle Instanz der Klasse (`.of(this)`), also die aktuelle Regel, erstellt und meistens unter dem Namen `params` abgespeichert. Die genaue Erstellung wird zu Beginn der ersten Beispielregel näher erläutert.

```
private final RuleParameters params = RuleParameters.of(this);
```

Ein weiterer wichtiger Teil der Klassenvariablen ist die Verknüpfung zu den Ressourcen, um Inhalte aus den Properties-Dateien oder Bilder einbinden zu können. Darüber können weitere Konstanten oder Listen definiert werden, die für alle Methoden der Klasse relevant sind.

```
private final RuleResources resources = RuleResources.of(this);
```

Der zweite Teil der Klasse, der den gesamten Check enthält, setzt sich wiederum aus drei verschiedenen Methoden zusammen. Einerseits gibt es die preCheck- und check-Methode aufbauend auf der Elternklasse *OneByOneRule*, andererseits gibt es auch die postCheck-Methode aus dem übergeordneten Interface *Rule*.

Die preCheck-Methode dient zur Feststellung, ob eine gültige Durchführung der check-Methode möglich ist oder nicht. Zum Beispiel können falsche Eingaben aus dem UI hier kontrolliert werden, bevor die check-Methode für die einzelnen Komponenten überhaupt beginnt. Bei Verwendung der preCheck-Methode muss diese aus der Elternklasse überschrieben werden.

```
@Override
public PreCheckResult preCheck() {
```

Die preCheck-Methode kann eines von zwei möglichen Ergebnissen zurückgeben. Entweder ein Ergebnis, das die check-Methode für irrelevant erklärt und somit die Prüfung abbricht und ein Ergebnis in Solibri ausgibt (`createIrrelevant()`), oder ein Ergebnis, welches die Fortführung des Checks zulässt (`createRelevant()`). Beide Ergebnisarten können auch einen erklärenden String besitzen, wobei dies eher bei der ersten Art Anwendung findet, da der Text in diesem Fall als Ergebnis in Solibri angezeigt wird.

```
return PreCheckResult.createIrrelevant("Fehler in der Eingabe im UI!");

// oder

return PreCheckResult.createRelevant();
```

Ist keine preCheck-Methode vorhanden oder liefert diese ein passendes Ergebnis, folgt im nächsten Schritt die check-Methode, welche als main-Methode das Herzstück der Regel bildet. Die Methode muss wiederum überschrieben werden.

```
@Override
public Collection<Result> check(Component component, ResultFactory
    resultFactory) {
```

Da die Klasse der Regel die Elternklasse *OneByOneRule* erweitert, hat die check-Methode den

bereits erwähnten speziellen Ablauf: die Komponenten des *DefaultFilters* werden einzeln der Reihe nach untersucht. So ist die check-Methode als eine Schleife zu verstehen, die nach jedem Durchlauf mit einer neuen Komponente `component` des *DefaultFilters* aufgerufen wird. Der spezielle Stellenwert dieses Filters ist in den folgenden Punkten nochmals hervorgehoben:

- Der *DefaultFilter* muss in jeder Regel enthalten sein.
- Die check-Methode läuft wie eine Schleife über alle Komponenten des *DefaultFilters*.
- Es können nur für die Komponenten aus dem *DefaultFilter* Ergebnisse in der check-Methode erstellt werden.
- Bei Anwendung mehrerer Regeln hintereinander, können nur diese Komponenten übergeben werden.

Nach der check-Methode ist es noch möglich, eine postCheck-Methode einzufügen. Diese wird ähnlich zur preCheck-Methode nur einmal durchgeführt und ist unabhängig von den Komponenten im *DefaultFilter*. Die Methode wird zum Beispiel verwendet, um Ergebnisse der Regel in eine externe Datei zu speichern.

```
@Override
public PreCheckResult postCheck() {
```

Der Aufbau und der gesamte Ablauf einer Prüfregel sowie das Zusammenspiel der verschiedenen Checks ist in Abb. 4.1 dargestellt. Nach dem Start der Regel wird, wenn vorhanden, der *preCheck* ausgeführt. In diesem wird die Entscheidung getroffen, ob der eigentliche Check gestartet werden soll, oder der Check irrelevant ist. Sollte der Check irrelevant sein, folgt der Abbruch der Prüfung mit einer Ergebnisausgabe in Solibri. Tritt kein irrelevantes Ergebnis auf, wird der Check für jede Komponente aus dem *DefaultFilter* durchgeführt. Dann erfolgt, ebenso nur bei Vorhandensein, der postCheck. Nach diesem werden dann die Ergebnisse zurückgegeben.

Der dritte Teil der Klasse besteht aus der Definition der Benutzeroberfläche (UI – *User Interface*). Das Rule-Interface definiert bereits einen Standardaufbau für das UI, in dem alle erstellten Parameter untereinander angeordnet sind. Um einen eigenen Aufbau festzulegen, muss die Methode `getParametersUIDefnition` überschrieben werden.

```
@Override
public UIContainer getParametersUIDefinition() {
```

In dieser Methode müssen nun alle Änderungen des UI angeführt werden. Das UI der Prüfregel kann individuell angepasst werden, beziehungsweise so individuell, wie es die API zu lässt. Alle verschieden, zur Verfügung stehenden UI-Elemente werden in folgendem Kapitel angeführt.

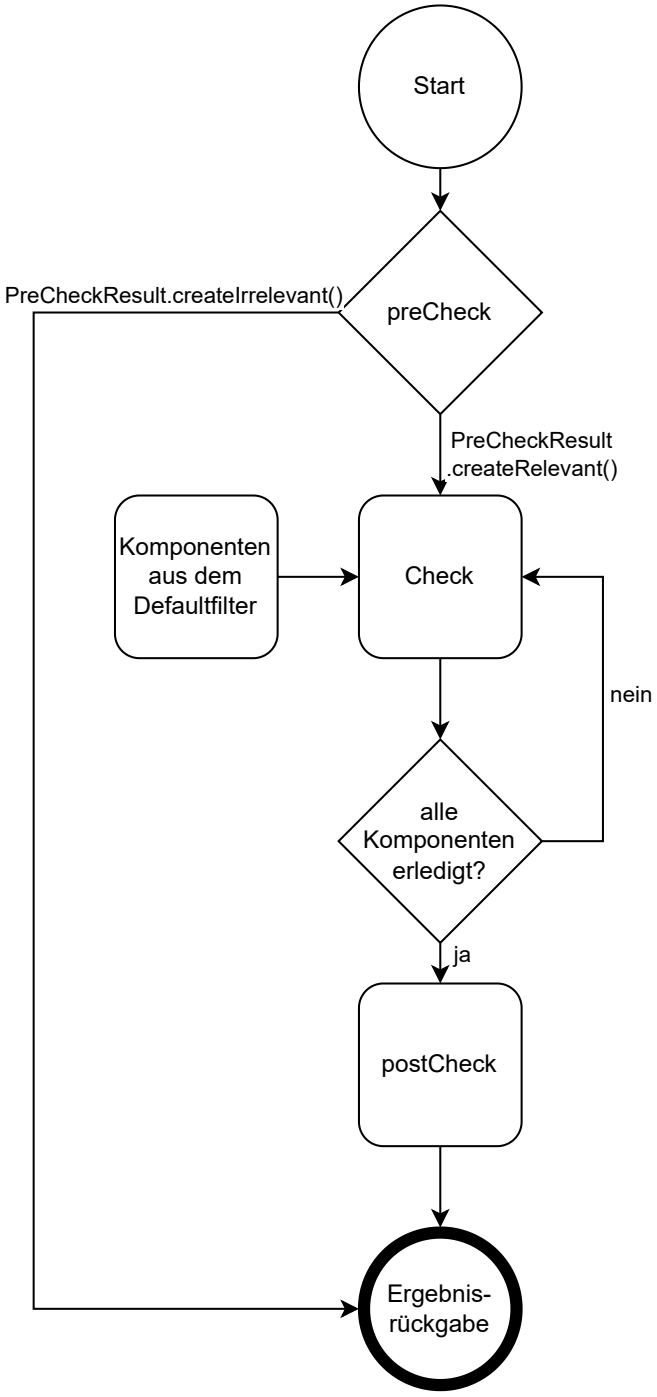

Abb. 4.1: Ablauf einer Prüfregel

5 UI-Elemente

Die Schnittstelle zwischen dem Programmcode und Anwenderinnen und Anwendern ist das *User Interface* (UI). Die Definition der Bestandteile des UI erfolgt am Ende des Codes der Prüfregel. Beschreibungen und Texte werden in den Properties-Dateien definiert. Dieses Kapitel erklärt die wichtigsten *Ruleparameter* sowie die wichtigsten Interfaces und Klassen, welche im Paket `com.solibri.smc.api.ui` enthalten sind. Die Ruleparameter dienen zur Interaktion bzw. Eingabe von Informationen, wohingegen die anderen Interfaces und Klassen zur Strukturierung und Erklärung des UI dienen.

5.1 Grundlagen und einfache UI-Elemente

Um das UI definieren zu können, muss zuerst die Voreinstellung von Solibri *getParametersUI-Definiton* überschrieben werden. Ohne diesen Schritt würden die Elemente in der Reihenfolge untereinander angeordnet werden, in der sie im Code erzeugt werden. Somit wäre keine individuelle Strukturierung der UI-Elemente möglich.

```
@Override
public UIContainer getParametersUIDefinition() {

   //Start UI Elements
```

5.1.1 Container

Die Grundstruktur des UI bilden die sogenannten `UIContainer`. Diesen können über `.addComponent` weitere Container und UI-Elemente hinzugefügt werden.

- `UIContainerHorizontal`: erzeugt einen horizontalen Container, in welchem hinzugefügte Komponenten horizontal ausgerichtet werden.
- `UIContainerVertical`: erzeugt einen vertikalen Container, in welchem hinzugefügte Komponenten vertikal ausgerichtet werden.

Zu Beginn des UI wird ein Hauptcontainer (`mainContainer`) erstellt, der alle Elemente des UI enthält. Das nachfolgende Beispiel zeigt die Erstellung eines Hauptcontainers, welchem direkt ein horizontaler Container mit drei vertikalen Untercontainern hinzugefügt wird. Die Ausgabe des Beispiels in Solibri ist in Abb. 5.1 dargestellt. Der String in der Klammer `("mainContainer")` stellt dabei den Titel des Containers dar und wird auch im UI ausgegeben.

```
@Override
public UIContainer getParametersUIDefinition() {

   //Start UI Elements

   //Begin main Container
   UIContainer mainContainer = UIContainerVertical.create("mainContainer");

   UIContainer uiContainerHorizontal = UIContainerHorizontal.create("
      horizontalContainer");
```

```
    UIContainer uiContainerVertical1 = UIContainerVertical.create("vertical1
        ");
    UIContainer uiContainerVertical2 = UIContainerVertical.create("vertical2
        ");
    UIContainer uiContainerVertical3 = UIContainerVertical.create("vertical3
        ");

    uiContainerHorizontal.addComponent(uiContainerVertical1);
    uiContainerHorizontal.addComponent(uiContainerVertical2);
    uiContainerHorizontal.addComponent(uiContainerVertical3);

    mainContainer.addComponent(uiContainerHorizontal);

    return mainContainer; //return main Container
}
```

Abb. 5.1: Hauptcontainer mit verschachtelten Untercontainern

Weitere `UIContainer` können zur übersichtlicheren Abwicklung auch als eigene Methode erstellt und dann dem `mainContainer` hinzugefügt werden (siehe Abb. 5.2).

```
    mainContainer.addComponent(ContainerMethod())

    return mainContainer; //return main Container
}

//Begin container as a method
private UIContainer ContainerMethod() {

    UIContainer uiContainerHorizontal = UIContainerHorizontal.create("
        ContainerMethod");

    return uiContainerHorizontal;
}
```

Abb. 5.2: Container als Methode

5.1.2 Verwendung von Ressourcen

Zu den Ressourcen zählen alle Texte und Bilder des UI und der Ergebnisbeschreibungen. Statt diese direkt als *String* (Text oder Dateipfad für ein Bild) in den Programmcode einzufügen, sollte eine Referenz zu den Ressourcen angelegt werden. Dies gelingt durch den Code `resources.getString("...")` in der Prüfregel. Dadurch können die Texte und Bilder in den

Ressourcen mehrsprachig angelegt werden. Das erfolgt in den Properties-Dateien einer Regel. Der String innerhalb der Methode `resources.getString("...")` verweist auf eine Zeile in der Properties-Datei, in welcher der eigentliche Inhalt (Text oder Dateipfad) angegeben ist. Als Beispiel wird der Titel der `containerMethod` über die Properties-Datei definiert. Abb. 5.3 zeigt die Darstellung in Solibri.

```
private UIContainer ContainerMethod() {

   UIContainer uiContainerHorizontal = UIContainerHorizontal.create(
      resources.getString("UI.ContainerMethod"));

   return uiContainerHorizontal;
}
```

Programmcode in der Properties-Datei:

```
UI.ContainerMethod = Container title via resources
```

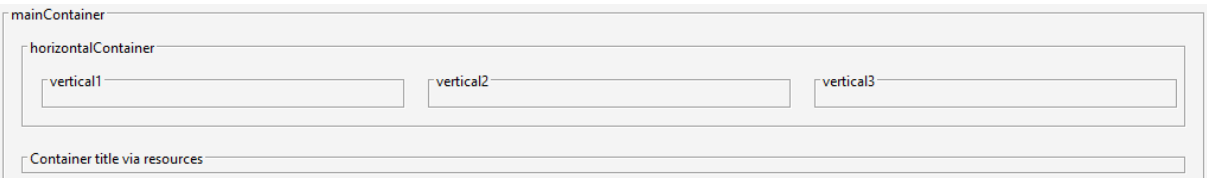

Abb. 5.3: Containertitel über Ressourcen Datei

Bei *Ruleparameter* erfolgt die Verlinkung zur Properties-Datei automatisch über die ID des Parameters. Die Methode `resources.getString("...")` wird dabei nicht benötigt. Genaueres zur Verwendung der Ressourcen für *Ruleparameter* folgt in Abschnitt 5.2.

5.1.3 UILabel

Mit `UILabel` können Textelemente in das UI eingefügt werden. Dies gelingt indem der Code `addComponent(UILabel.create("..."))` dem entsprechenden Container an der gewünschten Stelle angehängt wird (siehe Abb. 5.4).

```
//Begin main Container
UIContainer mainContainer = UIContainerVertical.create("mainContainer");

mainContainer.addComponent(UILabel.create(resources.getString("UI.
    mainContainer.Description")));
```

Programmcode in der Properties-Datei:

```
UI.mainContainer.Description = Description of the Rule
```

Falls längere Texte im UI vorkommen, setzt Solibri nicht automatisch Zeilenumbrüche. Stattdessen werden die Container einfach auf die notwendige Breite vergrößert. Zur besseren Übersicht sollten daher Umbrüche über `<br>` in der Properties-Datei gesetzt werden. Ebenfalls können Texte über `<font color=\COLOR\>` farbig dargestellt werden. Um die farbige Darstellung wieder zurückzusetzen muss `</font>` nach dem jeweiligen Text gesetzt werden. Sowohl für Zeilenumbrüche als auch für farbige Texte muss in der Textdefinition in der Properties-Datei vor dem Text `<html>` angeführt werden, da es sich bei den Befehlen um html-Befehle handelt. In der

Abb. 5.4: Texte über UILabel

Properties-Datei muss jedoch der gesamte Text in eine Zeile geschrieben werden. Hier lässt die Syntax keine Zeilenumbrüche zu. Das nachfolgende Beispiel fügt in den ersten beiden vertikalen Containern einen Text hinzu (siehe Abb. 5.5).

```
uiContainerVertical1.addComponent(UILabel.create(resources.getString("UI
    .Text1")));
uiContainerVertical2.addComponent(UILabel.create(resources.getString("UI
    .Text2")));
```

Programmcode in der Properties-Datei:

```
UI.Text1 = <html> This text is an example. If you <font color=\"red\">
    don't set a linebreak</font> the container gets wider.
UI.Text2 = <html> This text is an example. <br> Therefore <font color=\"
    green\">setting a linebreak</font> can help <br> keeping your UI
    clearly arranged.
```

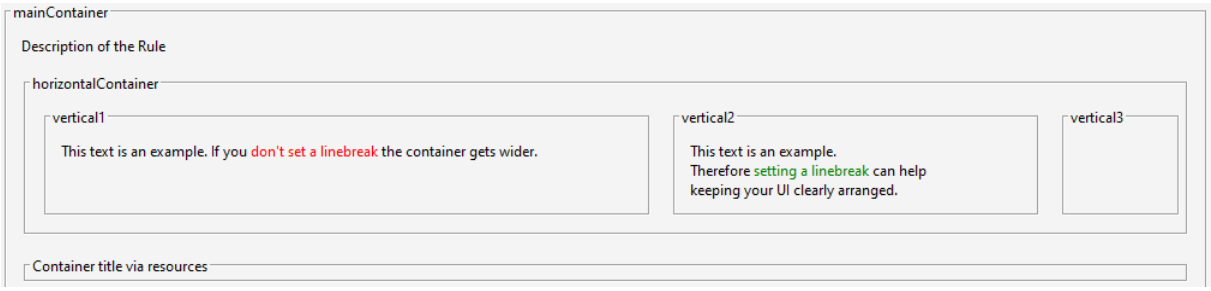

Abb. 5.5: Farbige Texte und Zeilenumbrüche – UILabel

5.1.4 UIImage

`UIImage` ermöglicht es, Bilder in das UI einzufügen. Dafür ist der Speicherpfad der Datei anzugeben. In dieser muss das Bild mit dem genauen Speicherpfad und der Dateiendung (z.B.: .png) angeführt werden. Im Sinne der Mehrsprachigkeit sollte das über die Properties-Datei erfolgen. In der Definition des UI wird auf die Properties-Datei referenziert, in der der Speicherpfad des Bildes enthalten ist (`resources.getImageURL(resources.getString("..."))`). Dieses muss, wie in Kapitel 3.2 beschrieben wird, im *images* Ordner gespeichert werden. In diesem Beispiel wird das Bild *UI_Image.png* aus dem *images* Ordner in den dritten vertikalen Container eingefügt (siehe Abb. 5.6).

```
UIImage image = UIImage.create(resources.getImageUrl(resources.getString
    ("UI.Image")));

uiContainerVertical3.addComponent(image);
```

Angabe des Dateispeicherpfades in der Properties-Datei:

```
UI.Image = images/UI_Image.png
```

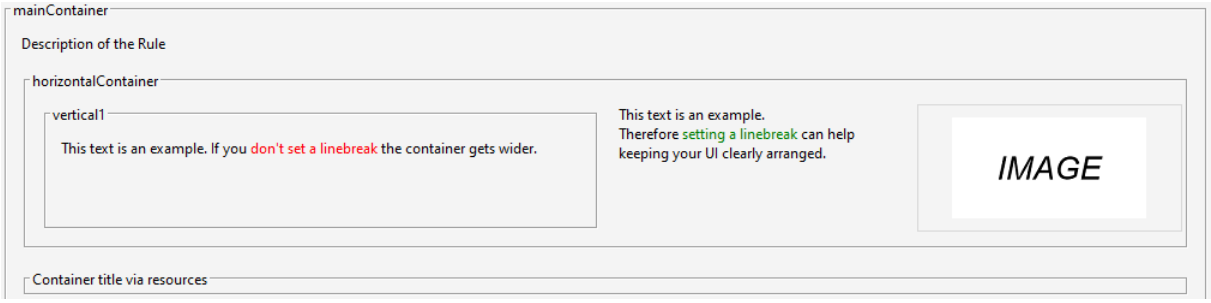

Abb. 5.6: UIImage

5.1.5 BorderType

Mit der Klasse `BorderType` (enum) können bei Containern Umrandungen angezeigt oder entfernt werden. Allgemein sind bei Containern mit Titeln automatisch immer Umrandungen sichtbar. Wird allerdings kein Titel (String in der Definition des Containers) angegeben, wird der Container ohne Rahmen erstellt. Die Umrandung kann entweder direkt bei der Erstellung des Containers gesetzt oder nachträglich über `.setBorderType(BorderType.LINE)` oder `.setBorderType(BorderType.NONE)` eingeschaltet oder ausgeschaltet werden. In Abb. 5.6 sind der zweite Container ohne Titel und ohne Umrandung und der dritte Container ohne Titel mit Umrandung ausgeführt.

5.2 RuleParameter

Ruleparameter ermöglichen die parametrische Programmierung von Prüfregeln. Über Sie werden Benutzereingaben in den Code eingebunden. Zur Erstellung von *Rulearameter* dient der *RuleParameterHandler* des Interface *Ruleparameters*. Bei Erstellung eines *Ruleparameter* ist ein String zu übergeben. Dieser String ist die ID des Parameters und stellt die Verbindung zu den Ressourcen her. Solibri empfiehlt, diese ID unter einem konstanten String zu speichern, damit sie nicht überschrieben werden kann. Eine Ausnahme ist der *DefaultFilterParameter*, denn bei dessen Erstellung gibt es keinen String zur Definition der ID. Stattdessen ist die ID in der *OneByOneRule* mit „rpComponentFilter“ vordefiniert. Die ID eines Parameters ist dessen Referenz zu den Ressourcen und kann in der Properties-Datei verwendet werden, um den Namen, eine Beschreibung und einen Standardwert für den Parameter festzulegen. Der Name eines Parameters ist in Solibri direkt neben oder ober dem Eingabefeld (je nach Parameter) sichtbar. Die Beschreibung kommt zum Vorschein, wenn der Mauszeiger im UI direkt auf dem Parameter platziert ist. Der Standardwert ist im UI stets vorausgefüllt. Dadurch können beispielsweise Grenzwerte von Normen voreingestellt werden. Beispiele und Codeausschnitte sind in der folgenden Erklärung der einzelnen Ruleparameter enthalten.

Für den Zugriff auf bzw. die Vorgabe von Elementen aus dem ifc-Modell werden Filterparameter zur Erstellung von sogenannten Komponentenfiltern (`ComponentFilter`) benötigt. Sie filtern alle Elemente des ifc-Modells und geben nur jene Elemente an den Code weiter, die im UI definiert sind. Einen solchen Filterparameter haben wir bereits kennengelernt: den `DefaultFilterParameter`. Es besteht auch die Möglichkeit, falls notwendig, weitere `ComponentFilter` anzulegen. Folgendes Beispiel zeigt die Erstellung von zwei Filterparameter und die Definition von Name `.NAME` und

Beschreibung `.DESCRIPTION` in der Properties-Datei. Die Vorgabe von Standardinhalten ist für Filterparameter nicht möglich.

```
//Constant IDs of parameters
private static final String COMPONENT_FILTER_PARAMETER_ID1 = "
    rpComponentFilter1";
private static final String COMPONENT_FILTER_PARAMETER_ID2 = "
    rpComponentFilter2";

// FilterParameter
final FilterParameter rpComponentFilter1 = params.createFilter(
    COMPONENT_FILTER_PARAMETER_ID1);
final FilterParameter rpComponentFilter2 = params.createFilter(
    COMPONENT_FILTER_PARAMETER_ID2);
```

Programmcode in der Properties-Datei:

```
rpComponentFilter1.NAME = ComponentFilter 1
rpComponentFilter1.DESCRIPTION = This filter a set of components.
rpComponentFilter2.NAME = ComponentFilter 2
rpComponentFilter1.DESCRIPTION = This filter specifies another set of
    components.
```

Des Weiteren gibt es *Ruleparameter* zur Definition von (fast) beliebigen Werten:

- `DoubleParameter` für Zahlenwerte
- `StringParameter` für Zeichenfolgen
- `PropertyReferenceParameter` für Properties aus der ifc-Datei (Auswahl ist eingeschränkt durch die vorhandenen Properties im ifc-Modell)

Diese müssen zunächst wieder am Beginn der Prüfregel angelegt werden. Dem `DoubleParameter` können an dieser Stelle Formattypen (`PropertyType`) zugeordnet werden. Als Beispiel wird einmal eine Länge und einmal ein Prozentwert gefordert. `String`- und `PropertyParameter` benötigen zur Erstellung lediglich eine ID. Für `DoubleParameter` und `StringParameter` können in der Properties-Datei zusätzlich zu einem Namen und einer Beschreibung auch Standardwerte `.DEFAULT_VALUE` festgelegt werden.

```
//Constant IDs of parameters
private static final String DOUBLE_PARAMATER_LENGTH_ID =
"rpDoubleParamaterLength";
private static final String DOUBLE_PARAMETER_PERCENTAGE_ID =
"rpDoubleParamaterPercentage";

private static final String STRING_PARAMETER_ID = "rpStringParameter";

private static final String PROPERTY_PARAMETER_ID =
"rpPropertyParameter";

// Retrieve the parameters creator of OneByOneRule
private final RuleParameters params = RuleParameters.of(this);

// DoubleParameter
final DoubleParameter rpDoubleParamaterLength = params
.createDouble(DOUBLE_PARAMATER_LENGTH_ID, PropertyType.LENGTH);
final DoubleParameter rpDoubleParameterPercentage = params
.createDouble(DOUBLE_PARAMETER_PERCENTAGE_ID, PropertyType.PERCENTAGE);
```

```
// StringParameter
final StringParameter rpStringParameter = params.createString(
    STRING_PARAMETER_ID);

// PropertyReferenceParameter
final PropertyReferenceParameter rpPropertyParameter = params
.createPropertyReference(PROPERTY_PARAMETER_ID);
```

Programmcode in der Properties-Datei:

```
rpDoubleParamaterLength.NAME = Length
rpDoubleParamaterLength.DESCRIPTION = Enter the length
rpDoubleParamaterLength.DEFAULT_VALUE = 1.20 m

rpDoubleParameterPercentage.NAME = Percentage
rpDoubleParameterPercentage.DESCRIPTION = Enter the percentage
rpDoubleParameterPercentage.DEFAULT_VALUE = 0.40

rpStringParameter.NAME = String
rpStringParameter.DESCRIPTION = Enter the string
rpStringParameter.DEFAULT_VALUE = text

rpPropertyParameter.NAME = PropertyParameter
rpPropertyParameter.DESCRIPTION = Enter the PropertyParameter
```

Abschließend existieren *Ruleparameter*, die mögliche Werte zur Auswahl vorgeben:

- `BooleanParameter` für Logikwerte (Wahr oder Falsch)
- `EnumerationParameter` als *Combobox*
- `EnumerationParameter` als *RadioButtonPanel*

`BooleanParameter` ermöglichen durch setzen eines Häkchens die Auswahl von wahr oder falsch. Der `EnumerationParameter` dient zur Definition einer Auswahlliste, aus der im UI eine Option zu wählen ist. Dem `EnumerationParameter` sind direkt in Form einer Liste `Arrays.asList(...,...)` die Anzahl und Namen der Auswahlmöglichkeiten zuordenbar. Er kann in Form einer *Combobox* (Dropdown-Liste) oder eines *RadioButtonPanels* (Anordnung von klickbaren Knöpfen) verwendet werden. Für `BooleanParameter` können in der Properties-Datei zusätzlich zu einem Namen und einer Beschreibung auch Standardwerte `.DEFAULT_VALUE` festgelegt werden. Für `EnumerationParameter` sind es hingegen zusätzlich die Namen der einzelnen Optionen. Insofern Bilder als Ergänzung für ein *RadioButtonPanel* dienen, sind ihre Dateipfade auch in der Poperties-Datei anzugeben.

```
// Constant IDs of parameters
private static final String BOOLEAN_PARAMATER_ID =
"rpBooleanParameter";
private static final String ENUMERATION_COMBOBOX_ID =
"rpEnumerationParameterForComboBox";
private static final String ENUMERATION_COMBOBOX_OPTION1_ID =
"rpCBOption1";
private static final String ENUMERATION_COMBOBOX_OPTION2_ID =
"rpCBOption2";
private static final String ENUMERATION_RADIOBUTTON_ID =
"rpEnumerationParameterRadioButton";
private static final String EUNMERATION_RADIOBUTTON_OPTION1_ID =
"rpRBoption1";
private static final String EUNMERATION_RADIOBUTTON_OPTION2_ID =
```

```
"rpRBoption2";

// Retrieve the parameters creator of OneByOneRule
private final RuleParameters params = RuleParameters.of(this);

// BooleanParameter
final BooleanParameter rpBooleanParamater = params
.createBoolean(BOOLEAN_PARAMATER_ID);

// EnumerationParameter
final EnumerationParameter rpEnumerationParameterForComboBox = params.
    createEnumeration(ENUMERATION_COMBOBOX_ID, Arrays.asList(
    EUNMERATION_COMBOBOX_OPTION1_ID, EUNMERATION_COMBOBOX_OPTION2_ID));
final EnumerationParameter rpEnumerationParameterForRadioButtons = params.
    createEnumeration(ENUMERATION_RADIOBUTTON_ID,
Arrays.asList(EUNMERATION_RADIOBUTTON_OPTION1_ID,
    EUNMERATION_RADIOBUTTON_OPTION2_ID));
```

Programmcode in der Properties-Datei:

```
rpBooleanParamater.NAME = Boolean
rpBooleanParamater.DESCRIPTION = Enter the Boolean
rpBooleanParamater.DEFAULT_VALUE = true

rpEnumerationParameterForComboBox.NAME = ComboBox
rpEnumerationParameterForComboBox.DESCRIPTION = Select one value
rpCBOption1 = Option 1
rpCBOption2 = Option 2

rpEnumerationParameterForRadioButtons.NAME = RadioButtonPanel
rpEnumerationParameterForRadioButtons.DESCRIPTION = Select one value
rpRBoption1 = Option 1
rpRBoption2 = Option 2
UI.Image_RB_1 = images/UI_RB_1.png
UI.Image_RB_2 = images/UI_RB_2.png
```

Nach der Erstellung aller *Ruleparameter*, müssen diese noch in die UI-Struktur eingebettet werden. Dafür kommt das Interface `UIRuleParameter` zum Einsatz. Instanzen von `UIRuleParameter` können mittels der Methode `UIRuleParameter.create (...)` für fast alle *RuleParameter* erstellt werden. Die Ausnahme bildet das *RadioButtonPanel*, da es dafür ein *SubInterface* `UIRadioButtonPanel` und dafür zwei weitere Subinterfaces `UIRadioButtonPanelVertical` und `UIRadioButtonPanelHorizontal` gibt. Ein vertikales *RadioButtonPanel* wird beispielsweise über `UIRadioButtonPanelVertical.create(...)` erstellt. Diesem kann anschließend mit der Methode `.addOptionImages(...)` eine Liste von ergänzenden Bildern übergeben werden. Sowohl das `UIRadioButtonPanel` als auch die anderen `UIRuleParameter` sind schließlich über `uiContainer.addComponent(...);` in die UI-Struktur einzubetten. Das folgende Code-Beispiel zeigt die Einbettung eines `DoubleParameter` und eines *RadioButtonPanel*. In Abb. 5.7 ist ein UI mit allen beschriebenen *Ruleparameter* zu sehen. Um diese Form zu erreichen, müssen die *Ruleparameter* in Containern angeordnet werden. Der Programmcode zur Erstellung dieses UI kann dem Anhang 10 entnommen werden.

```
//Include Double Parameter
uiContainer.addComponent(UIRuleParameter.create(rpDoubleParamaterLength));

//Include RadioButtonPanel
UIRadioButtonPanel radioButtonPanelVertical = UIRadioButtonPanelVertical .
    createrpEnumerationParameterForRadioButtons);
```

```
List<UIImage> radioButtonImages = new ArrayList<>();
radioButtonImages.add(UIImage.create(resources.getImageUrl(resources.
    getString("UI.Image_RB_1"))));
radioButtonImages.add(UIImage.create(resources.getImageUrl(resources.
    getString("UI.Image_RB_2"))));
radioButtonPanelVertical.addOptionImages(radioButtonImages);

uiContainer.addComponent(radioButtonPanelVertical);
```

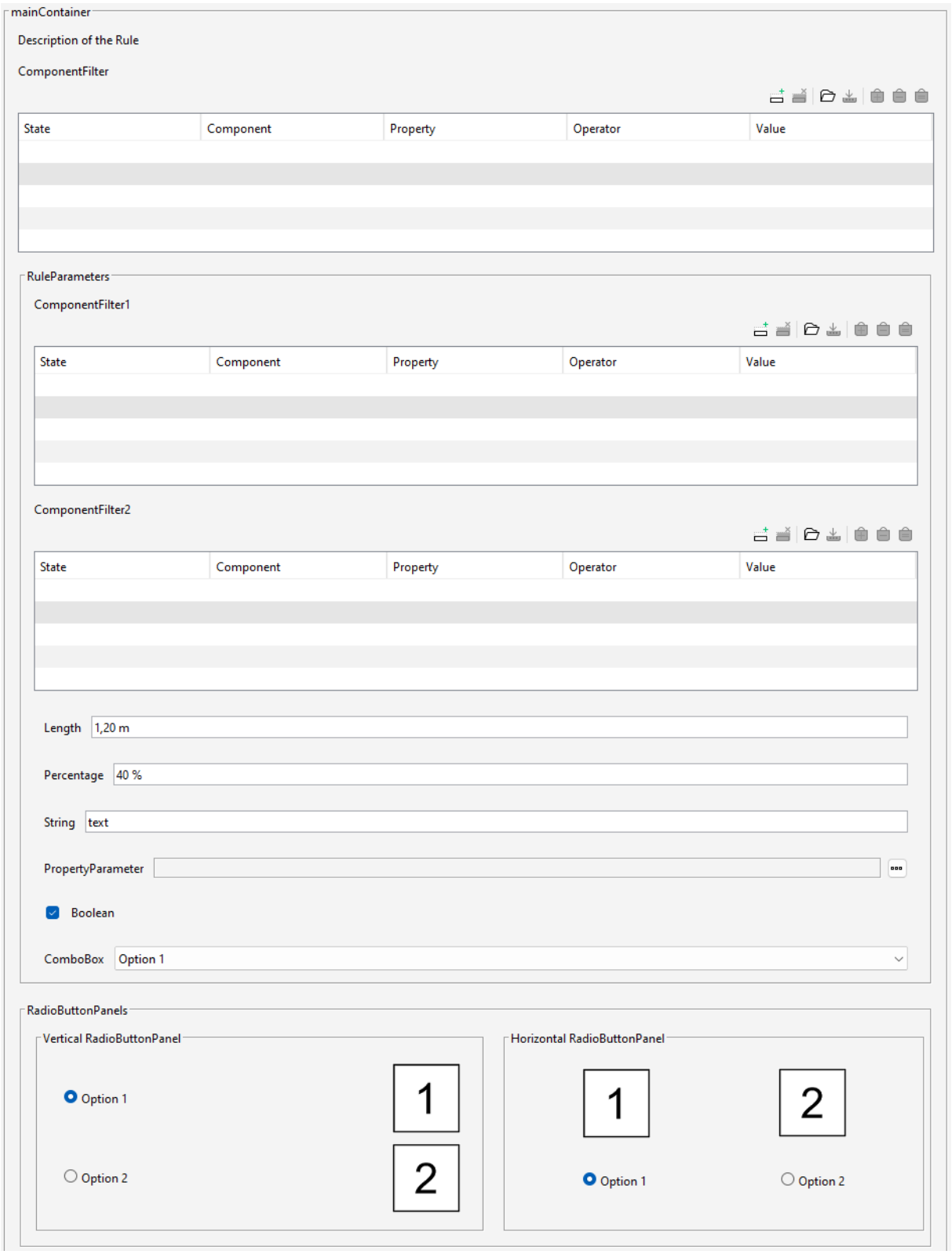

Abb. 5.7: RuleParameter im User Interface

6 Kochrezept-Prüfregel

Dieses Kapitel zeigt einen Musterprozess für die Programmierung einer Prüfregel, wie er den Autoren sinnvoll erscheint. Als erster Schritt sollte die Problemstellung genau untersucht werden. Wichtige Anfangsgedanken bzw. Fragen sind vor allem:

- Eingabewerte – Welche Eingabewerte (Input-Werte) sind erforderlich?
- Komponenten – Auf welche Komponenten ist die Prüfregel ausgelegt? Kann es zu Fehlern bei anderen Komponenten kommen?
- Verzweigungen, Szenarien – Gibt es besondere Fälle oder Szenarien, die unterschiedliche Herangehensweisen erfordern. Beispiele dafür sind verschiedene Berechnungskonzepte für unterschiedliche Komponenten oder Änderungen bei Überschreiten von Grenzwerten.
- Variablen - Welche Variablen werden im Programmcode zur Abwicklung der Prüfregel bzw. zur Strukturierung der Ergebnisse benötigt?
- *preCheck* (Vorprüfung) - Gibt es Benutzereingaben, die zu einem Programmabsturz führen und daher vor der eigentlichen Prüfung im *preCheck* abzufangen sind?

Als zweiten Schritt empfiehlt es sich, den Prozess bzw. die Funktionsweise der Prüfregel grafisch in einem Flussdiagramm abzuhandeln. So kann zu Beginn ein Überblick über den Ablauf des Prüfvorgangs geschaffen werden.

Danach kann als dritter Schritt mit der eigentlichen Programmierung begonnen werden. Aber wie beginnt man nun am besten mit der Programmierung einer neuen Prüfregel in Java? Allgemein handelt es sich ähnlich wie bei den meisten programmiertechnischen Aufgaben um einen iterativen Prozess. Zu Beginn sollte das UI erstellt werden um alle benötigten Eingangswerte abrufen zu können, jene aus dem ifc-Modell (benötigte Komponenten und dazugehörige Informationen) sowie direkte Eingaben im UI (beispielsweise fixe Grenzwerte). Anschließend können die ersten Zeilen des Codes in der check-Methode geschrieben werden. Dabei hilft es Zwischenstände als Ergebnis in der check-Methode zu deklarieren und in Solibri ausgeben zu lassen. Einzelne funktionstüchtige Methoden sollten dann aus der check-Methode ausgelagert werden, um einen besseren Überblick über den Code zu bewahren. Ebenfalls müssen sich die Programmierenden in die Lage der Benutzerinnen und Benutzer versetzen, um sukzessive eine anwendbare Prüfregel zu erstellen.

7 Beispielregel 1: einfacher ClashCheck

Das Anwendungsziel der ersten Beispielregel ist die Durchführung einer einfachen Kollisionsprüfung. Dafür müssen zwei Gruppen von Komponenten angegebenen werden, die gegeneinander auf Kollision zu prüfen sind. Für auftretende Kollisionen wird ein Ergebnis erstellt, welches die beiden betroffenen Komponenten enthält.

7.1 User Interface

Das User Interface der ersten Beispielregel besteht aus zwei Komponentenfiltern (siehe Abb. 7.1).

1. Der erste Komponentenfilter dient zur Eingabe der Komponenten, für die überprüft werden soll, ob eine Kollision vorliegt. Die hier eingegebenen Komponenten bilden die Grundlage der check-Methode.

2. Der zweite Komponentenfilter muss die Komponenten enthalten, mit denen die Ausgangskomponenten aus dem vorherigen Filter auf Kollision überprüft werden sollen.

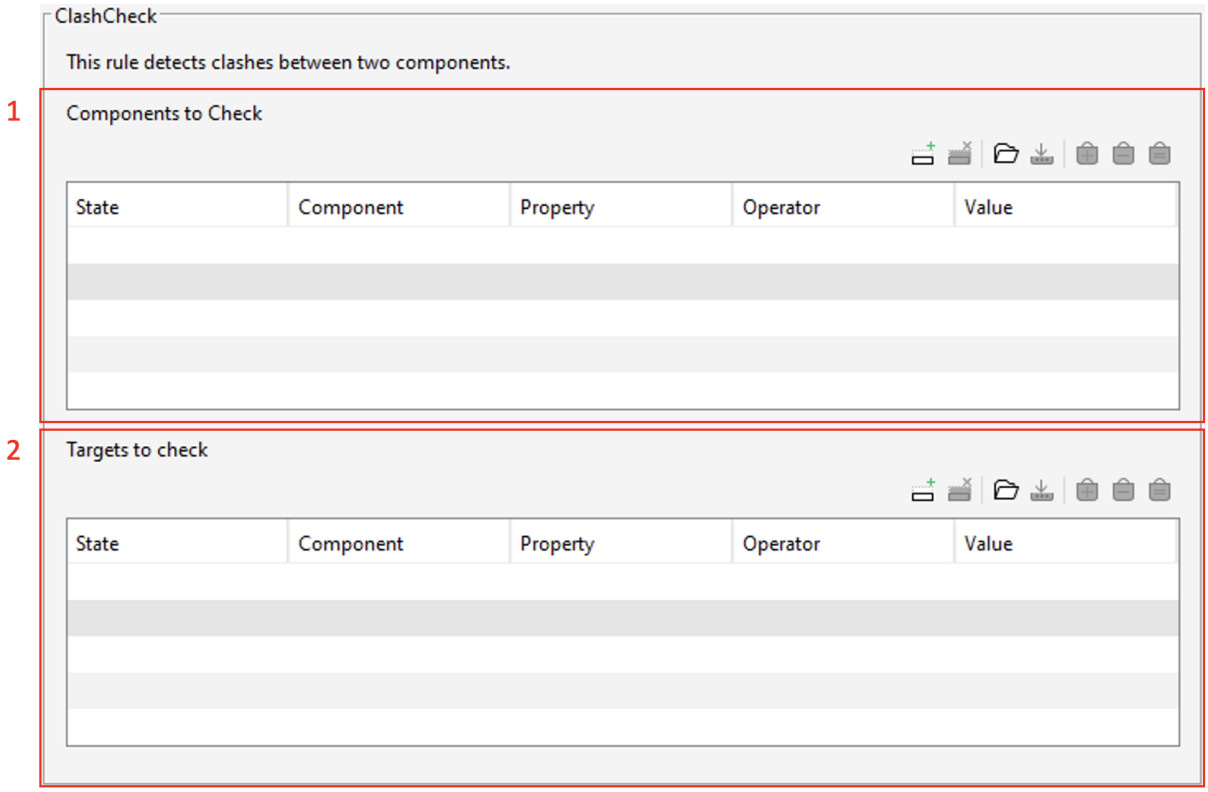

Abb. 7.1: UI der Beispielregel 1

7.2 Programmcode

Wie zu Beginn jeder Regel wird mit dem Package `com.solibri.rule` gestartet. Alle weiteren importieren Packages ergeben sich aus den im Programmcode verwendeten Befehlen von selbst. In der ersten Beispielregel gibt es beispielsweise allgemeine Packages für Java (Zeilen 3-5) sowie spezielle Packages der Solibri-API (Zeilen 7-20).

```
1  package com.solibri.rule;
2
3  import java.util.ArrayList;
4  import java.util.Collection;
5  import java.util.Set;
6
7  import com.solibri.smc.api.SMC;
8  import com.solibri.smc.api.checking.FilterParameter;
9  import com.solibri.smc.api.checking.OneByOneRule;
10 import com.solibri.smc.api.checking.Result;
11 import com.solibri.smc.api.checking.ResultFactory;
12 import com.solibri.smc.api.checking.RuleParameters;
13 import com.solibri.smc.api.checking.RuleResources;
14 import com.solibri.smc.api.intersection.Intersection;
15 import com.solibri.smc.api.model.Component;
16 import com.solibri.smc.api.ui.BorderType;
17 import com.solibri.smc.api.ui.UIContainer;
18 import com.solibri.smc.api.ui.UIContainerVertical;
19 import com.solibri.smc.api.ui.UILabel;
20 import com.solibri.smc.api.ui.UIRuleParameter;
```

Als nächstes beginnt die Klasse der Prüfregel, welche die `OneByOneRule` erweitert. Wie bereits im Kapitel 4 erläutert, definiert die Klasse `OneByOneRule` zum Beispiel schon den Ablauf der check-Methode, dass diese wie eine Schleife für jede eingegebene Komponente im *DefaultFilter* einzeln durchlaufen wird.

```
22 public final class ClashCheck extends OneByOneRule {
```

Im nächsten Schritt werden konstante IDs für die verschiedenen *Ruleparameter* definiert. Mittels Kommentaren wird zur besseren Übersicht eine Einteilung getroffen. Die Verwendung von Namenskonstanten wird von Solibri vorgeschlagen, wenn mehrere verschiedene *Ruleparameter* erstellt werden müssen und der Code übersichtlich gehalten werden soll. Die Namen der Variablen bestehen aus Großbuchstaben und beschreiben den Typ des Parameters. Die Strings selbst beginnen mit `rp` um zu verdeutlichen, dass es sich um *Ruleparameter* handelt. Bei dieser einfachen Regel wird nur eine ID benötigt, da neben dem *DefaultFilter* noch ein weiterer `ComponentFilter` verwendet wird. Die ID des `DefaultFilter` ist bereits in der `OneByOneRule` vordefiniert („rpComponentrFilter“). Die hier verwendeten IDs schaffen die Verbindung zu der Properties-Datei (siehe Code 7.1), wo der anzuzeigende Inhalt definiert wird.

```
25 // FilterParameter
26 private static final String COMPONENT_FILTER_TARGETS_ID = "
       rpComponentFilterTargets";
27 // PropertyParamater
28 // DoubleParameter
29 // EnumerationParameter
30 // BooleanParameter
31 // StringParameter
```

Nach der Erstellung der IDs wird der *RuleParameterHandler* der aktuellen Regel (`.of(this)`) erstellt und unter `params` abgespeichert. Dieser erlaubt, wie bereits in Kapitel 4 erklärt, die Erstellung der *Ruleparameter* mit den vorhin definierten IDs. Zu beachten ist, dass nicht auf den `DefaultFilterParameter` vergessen werden darf, welcher direkt von der `OneByOneRule` (mittels `this`) abgerufen wird. Der `DefaultFilter` ist der mindestens notwendige `ComponentFilter` und muss immer angeführt werden. Die Namensgebung der *Ruleparameter* orientiert sich an den vorhin festgelegten IDs.

```
33 // Create the parameter creation handler for the rule
34 private final RuleParameters params = RuleParameters.of(this);
35
36 // Parameter for the user input
37 // FilterParameter
38 final FilterParameter rpComponentFilter = this.getDefaultFilterParameter();
39 final FilterParameter rpComponentFilterTargets = params.createFilter(
       COMPONENT_FILTER_TARGETS_ID);
40 // PropertyReferenceParameter
41 // DoubleParameter
42 // EnumerationParameter
43 // BooleanParameter
44 // StringParameter
```

Ähnlich zum Aufruf des *RuleParameterHandler* werden auch die Ressourcen der aktuellen Instanz aufgerufen, um im Programmcode auf sie zugreifen und verweisen zu können.

```
47 private final RuleResources resources = RuleResources.of(this);
```

Nach der Definition aller Parameter, wird mit dem wesentlichen Teil der Regel gestartet: der check-Methode. Diese muss mit `@Override` überschrieben werden. Sie bekommt stets eine Komponente aus dem *DefaultFilter* übergeben, führt für diese den Code aus und erstellt ein Ergebnis. Dafür muss zusätzlich eine Instanz des Interface `ResultFactory` übergeben werden.

```
49 // Check method
50 @Override
51 public Collection<Result> check(Component component, ResultFactory
       resultFactory) {
```

Zu Beginn müssen die Komponenten des zweiten Komponentenfilter aus dem ifc-Modell abgerufen werden (`SMC.getModel().getComponents(rpComponentFilterTargets.getValue())`). Sie werden in einer *Collection* `targets` gespeichert. Eine zweite *Collection* `results` benötigt man zur Speicherung aller Ergebnisse, die am Ende der check-Methode als Rückgabewert dient (Zeile 84).

```
53 // Get the values from the model using the second filter from the UI.
54 Collection<Component> targets = SMC.getModel().getComponents(
       rpComponentFilterTargets.getValue());
55
56 // Creating a collection of results
57 Collection<Result> results = new ArrayList<>();
```

Nun wird mit der Überprüfung der einzelnen Komponenten `targets` innerhalb einer for-Schleife begonnen. In dieser Schleife findet der eigentliche *ClashCheck* zwischen der Ausgangskomponente und den Zielkomponenten statt.

Im ersten Schritt wird das Set `intersections` erstellt. Dieses enthält die jeweilige Überschneidung zwischen der Ausgangskomponente und der im Schleifendurchlauf gerade an der Reihe befindlichen Zielkomponente `target`. In Solibri kann es vorkommen, dass es Überschneidungen mit negativem Volumen gibt oder, dass Berührungen von zwei Komponenten als Überschneidungen erkannt werden. Beide Arten sind somit keine wirklichen Überschneidungen und sollen auch nicht im Ergebnis auftauchen. Daher muss eine Überprüfung auf ein positives (`> 0.000001` m^3) Volumen der *Intersection* durchgeführt werden. Diese Überprüfung findet in der Bedingung einer if-Verzweigung statt. Um auf die einzelnen Überschneidungen zugreifen zu können, findet der gesamte Prozess in einer for-Schleife über alle *Intersections* statt.

Bei Erfüllung der Bedingung der if-Verzweigung, wird zur Ergebniserstellung übergegangen. Dazu ist die `resultFactory` vorgesehen. Bei der Erstellung sind der Name des Ergebnisses `title` sowie eine Beschreibung `description` mitzugeben. Da in Solibri nicht nur die Ausgangskomponente, sondern auch das Hindernis angezeigt werden soll, muss dieses noch zum Ergebnis hinzugefügt werden (`.withInvolvedComponent(target)`). Der Titel sowie die Beschreibung des Ergebnisses sind jeweils Strings, die beschreiben, welche Komponente (`component.getName()`) mit welchem Hindernis (`target.getName()`) kollidiert. Der Mittelteil des Strings wird aus den Ressourcen bezogen, um der eingestellten Sprache zu entsprechen. Nachdem der Schweregrad der Ergebnisse nicht explizit angegeben wurde, besitzen diese den Grad `moderate` und werden in Solibri mit der Farbe orange gekennzeichnet.

Nach den Überprüfungen und Erstellung der Ergebnisse werden die beiden for-Schleifen beendet.

```
60  for (Component target : targets) {
61
62      // Creating a set containing the intersections
63      Set<Intersection> intersections = component.getIntersections(target);
64
65      // Loop through each intersection to create the results
66      for (Intersection intersection : intersections) {
67
68          if (intersection.getVolume() > 0.000001) {
69
70              String title = component.getName() + ' ' + resources.getString("RE
                    .ClashResult.TITLE") + ' '
71              + target.getName();
72
73              String description = component.getName() + ' ' + resources.
                    getString("RE.ClashResult.DESCRIPTION")
74              + ' ' + target.getName();
75
76              Result result = resultFactory.create(title, description).
                    withInvolvedComponent(target);
77
78              results.add(result);
79          }
80
81      }
82
83  }
```

Nach Überprüfung aller Ausgangskomponenten, wird die Ergebnis-Collection zurückgegeben (`return results`) und die check-Methode beendet.

```
84      return results;
85
86  }
```

Nach der check-Methode wird das UI definiert, wobei auch hier die Voreinstellung der `OneByOneRule` überschrieben werden müssen.

```
88  @Override
89  public UIContainer getParametersUIDefinition() {
```

Zuerst benötigt man einen `mainContainer`, der alle Inhalte des UI umfasst. Dieser wird als vertikaler Container mit dem Titel aus den Ressourcen (`UI.TITLE`) und einer umlaufenden Linie (`BorderType.Line`) definiert. Im Anschluss ist eine Beschreibung als `UILabel` angeführt.

```
91  // Vertical container with title
92  UIContainer mainContainer = UIContainerVertical.create(resources.getString(
        "UI.TITLE"), BorderType.LINE);
93
94  // Description vertical container
95  mainContainer.addComponent(UILabel.create(resources.getString("UI.
        DESCRIPTION")));
```

Nun werden zum vorhin erstellten `mainContainer` die beiden benötigten Ruleparameter hinzugefügt (`mainContainer.addComponent()`). Zuerst müssen diese jedoch jeweils noch erstellt werden (`UIRuleParameter.create()`). Gestartet wird mit dem verpflichtenden Komponentenfilter (`rpComponentFilter`), gefolgt vom zusätzlichen Komponentenfilter (`rpComponentFilterTargets`).

```
97  // Default filter
98  mainContainer.addComponent(UIRuleParameter.create(rpComponentFilter));
99
100 // Component filter
101 mainContainer.addComponent(UIRuleParameter.create(rpComponentFilterTargets)
        );
```

Schließlich wird der `mainContainer` zurückgegeben und die Methode zur UI-Definition sowie die gesamte Klasse werden beendet.

```
103         return mainContainer;
104     }
105
106 }
```

7.3 Ressourcen

Für diese Prüfregeln bestehen die Ressourcen lediglich aus den Properties-Dateien, da es keine Bilder im UI gibt. In der Properties-Datei werden zuerst Metadaten zur Prüfregel angeführt (Zeile 1-7). Dazu zählen der Titel und eine Beschreibung der Regel, sowie der Autor und die Versionsnummer. Diese sind in Solibri im Ruleset Manager in dem eigenen Fenster *INFO* zu sehen (siehe Abb. 7.2).

Danach werden für die beiden Komponentenfilter, die Namen und die Beschreibungen definiert. Auch der Name sowie die Beschreibung des `mainContainer` werden angeführt. Aufgrund der gewählten Ausgabe der Ergebnisse, muss der dafür benötigte String ebenfalls noch angegeben werden.

Program Code 7.1: Properties-Datei der Beispielregel 1

```
1   DEFAULT_NAME=Clash Check Example Rule
2   DEFAULT_DESCRIPTION=This example rule detects clashes between two
        components.
3   AUTHOR=ZDB TU Wien
4   AUTHOR_TAG=ZDB
5   UID=12345
6   VERSION=1.0
7   DATE=2023-02-02
8
9   rpComponentFilter.NAME = Components to Check
10  rpComponentFilter.DESCRIPTION = This filter specifies the set of
        components to check.
11
12  rpComponentFilterTargets.NAME= Targets to check
13  rpComponentFilterTargets.DESCRIPTION= This filter specifies the set of
        components to check.
14
15  UI.TITLE = ClashCheck
16  UI.DESCRIPTION = This rule detects clashes between two components.
17
18  RE.ClashResult.TITLE = clashes with
19  RE.ClashResult.DESCRIPTION = clashes with
```

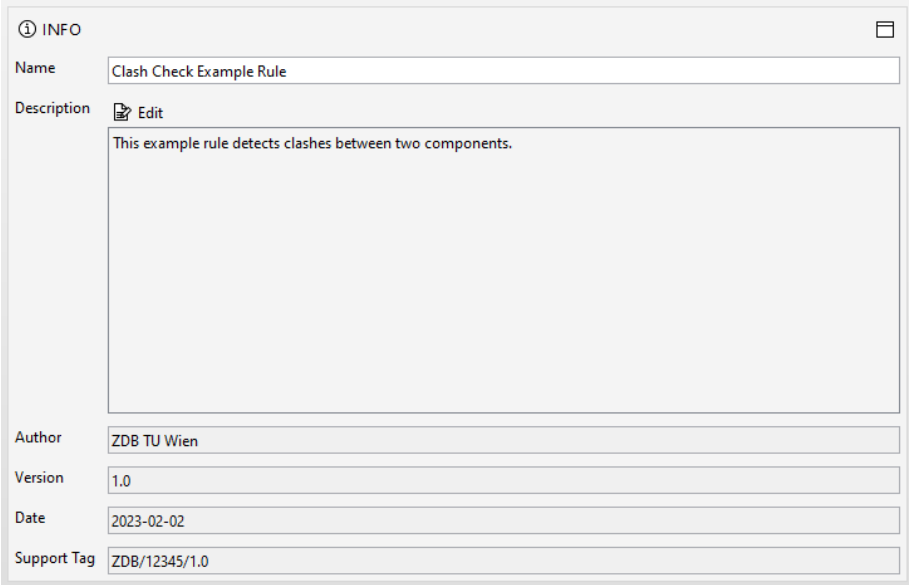

Abb. 7.2: Metadaten der Beispielregel 1 in Solibri

8 Weiterführende Themen

In diesem Kapitel werden einige weiterführende Themen erklärt, die entweder in der darauffolgenden zweiten Beispielregel verwendet werden oder allgemein von Wichtigkeit sind.

8.1 RuleInterface

Das `RuleInterface`, definiert im Paket `com.solibri.smc.api.checking`, dient als Interface des Prüfregelcodes. Es definiert grundlegende Funktionen einer Prüfung wie die preCheck-, check- und postCheck-Methode, sowie die Funktionalität des *DefaultFilter* und die Bewertung von geprüften Komponenten. Eine Prüfregel kann entweder direkt das *RuleInterface* implementieren, um die Zugriff auch die notwendige Funktionalität für eine Prüfung zu haben, oder von einer Klasse erben, die das *RuleInterface* implementiert. Eine solche Klasse ist die *OneByOneRule*, welche standardmäßig verwendet werden sollte. Dabei wird zu Beginn der Prüfung, wie bereits in Kapitel 4 beschrieben, die Elternklasse *OneByOneRule* erweitert:

```
public final class ExampleRule extends OneByOneRule{
```

Bei der *OneByOneRule* werden, wie der Name bereits verrät, alle gefilterten Komponenten (über den *DefaultFilter*) einzeln überprüft. Daher bietet diese abstrakte Klasse eine gute Vorlage für Prüfregeln vieler Anwendungsfälle und Problemstellungen. Die wichtigsten Methoden der Klasse sind hier aufgelistet:

`Collection<Result> check(Component component, ResultFactory resultFactory)`

Die check-Methode ist die main-Methode der *OneByOneRule*. Sie wird also von jeder Regel, die von der *OneByOneRule* erbt, aufgerufen. Sie ist ebenso eine abstrakte Methode und daher ist sie in jeder neuen Klasse, die eine Regel definiert, zu überschreiben. Alle Befehle, die auszuführen sind, sind in der check-Methode zu implementieren. Als Rückgabewert gibt die check-Methode eine `Collection<Result>`, also eine Sammlung der Ergebnisse zurück.

`getDefaultFilterParameter()`

Gibt den Standard Filterparameter für eine Regel zurück. Die Komponenten dieses Filters werden während der *OneByOneRule* automatisch nacheinander überprüft. Diese Methode muss für die Funktionsweise der *OneByOneRule* implementiert werden. Da es um eine Instanzmethode handelt, wird sie mit `this.getDefaultFilterParameter aufgerufen.`

`preCheck()`

Die preCheck-Methode wird vor der check-Methode durchgeführt. Sie eignet sich einerseits für Voruntersuchungen (z. B. die Überprüfung der Benutzereingaben) und andererseits für Methoden, die in der check-Methode nicht für jede Komponente eigens durchgeführt werden müssen. Schaltet man solche allgemeine Methoden vor, kann die Performance der Regel optimiert werden. Als Ergebnis der preCheck-Methode muss ein `PreCheckResult` zurückgegeben werden. Es gibt dabei zwei Möglichkeiten: `PreCheckResult.createIrrelevant` um die Überprüfung abzubrechen und eine Fehlermeldung anzuzeigen und `PreCheckResult.createRelevant` um die check-Methode normal zu starten.

Neben der *OneByOneRule* gibt es auch die abstrakte Klasse *ConcurrentRule*, bei welcher

Komponenten einen Filter parallel und gleichzeitig passieren. Die *ConcurrentRule* nutzt dazu einen eigenen *Thread* für jede Komponente des Standardfilters, indem sie auf einen *Thread Pool* zurückgreift. Je nachdem wie viele *Threads* im Pool zur Verfügung stehen, können mehrere Komponenten gleichzeitig behandelt werden. Bei der Verwendung der *ConcurrentRule* ist darauf zu achten, dass Variablen Thread-sicher verwendet werden, also dass mehrere *Threads* nicht gleichzeitig auf die gleichen Daten zugreifen. Bei richtiger Verwendung sollte die *ConcurrentRule* eine Performancesteigerung herbeiführen. Die *ConcurrentRule* besitzt die gleichen Methoden wie die *OneByOneRule*.

Sowohl die *ConcurrentRule* als auch das *RuleInterface* sollten nur von fortgeschrittenen Solibri-API Nutzerinnen und Nutzern als Grundlage für ihre Prüfregel verwendet werden. Üblicherweise ist die *OneByOneRule* eine gute Wahl.

8.2 Debugging

Erstellter Code funktioniert in den seltensten Fällen auf Anhieb genau so wie gewollt. Fehler im Code werden als sogenannte Bugs bezeichnet. Das *Debugging* ist jener Vorgang, bei dem Fehler im Code identifiziert und bereinigt werden. Entwicklungsumgebungen bieten dafür als Hilfsmittel Debugger an, die es ermöglichen, den Code Schritt für Schritt auszuführen und Zwischenergebnisse zu analysieren. Solibri bietet die Möglichkeit, den Debugger aus *Eclipse* mit Solibri zu verbinden, um den Code mit den Eingangswerten aus dem Modell durchzugehen. Die Erklärung des *Debugging* in diesem Abschnitt beruht auf den Beschreibungen auf der *Solibri Developer Platform*, welche unter dem Reiter *Manual* und dem Abschnitt *Debugging* zu finden sind.

8.2.1 vmoptions-Datei

Der erste Schritt umfasst eine Änderung der Datei *Solibri.vmoptions.txt*. Wenn die Installation von Solibri laut Abschnitt 2.1 erfolgte, ist jene unter `C:/Program Files/Solibri/SOLIBRI` zu finden. Die Datei kann mit jedem beliebigen Texteditor geöffnet und bearbeitet werden. Um das Debuggen zu ermöglichen, müssen in die Datei zwei weitere Argumente hinzugefügt werden, welche in Abb. 8.1 rot markiert sind. Dieser Schritt muss, im Unterschied zu den weiteren Schritten, nur einmal durchgeführt werden und kann mittel Copy & Paste erfolgen.

8.2.2 Debug Configurations

Im nächsten Schritt wird die Verbindung zwischen *Eclipse* und Solibri konfiguriert. Dafür müssen beide Programme gleichzeitig geöffnet sein. In *Eclipse* werden die *Debug Configurations*, die mittels des Pfeils beim kleinen grünen Käfer zu finden sind (siehe Abb. 8.2), aufgerufen. In dem sich öffnenden Fenster wird mit einem Rechtsklick auf *Remote Java Application* und der Auswahl von *New Configuraion* eine neue Konfiguration angelegt (siehe Abb. 8.3).

Eine Konfiguration bezieht sich immer auf ein *JavaProject*. Ist beim Öffnen der *Debug Configurations* das richtige Projekt aktiv, erscheint auch hier bereits das richtige Projekt. Ansonsten kann mit *Browse...* noch das gewünschte Projekt ausgewählt werden. Der Name der Konfiguration kann frei gewählt werden. Wichtig zu ändern ist auch der Port auf *8888*, so wie in der Datei vorhin eingegeben. Durch einen Klick auf *Debug* wird die Verbindung konfiguriert und aufgebaut.

Die Konfiguration der Verbindung zwischen *Eclipse* und Solibri muss für jedes Projekt neu durchgeführt werden. Wenn beim nächsten Öffnen der beiden Programme mit dem *Debugging* fortgefahren werden soll, muss die Verbindung jedoch nicht neu konfiguriert, sondern nur erneut aufgebaut werden. Dies erfolgt durch erneutes Klicken des Pfeils beim kleinen grünen Käfer und Auswahl der bereits erstellten Konfiguration.

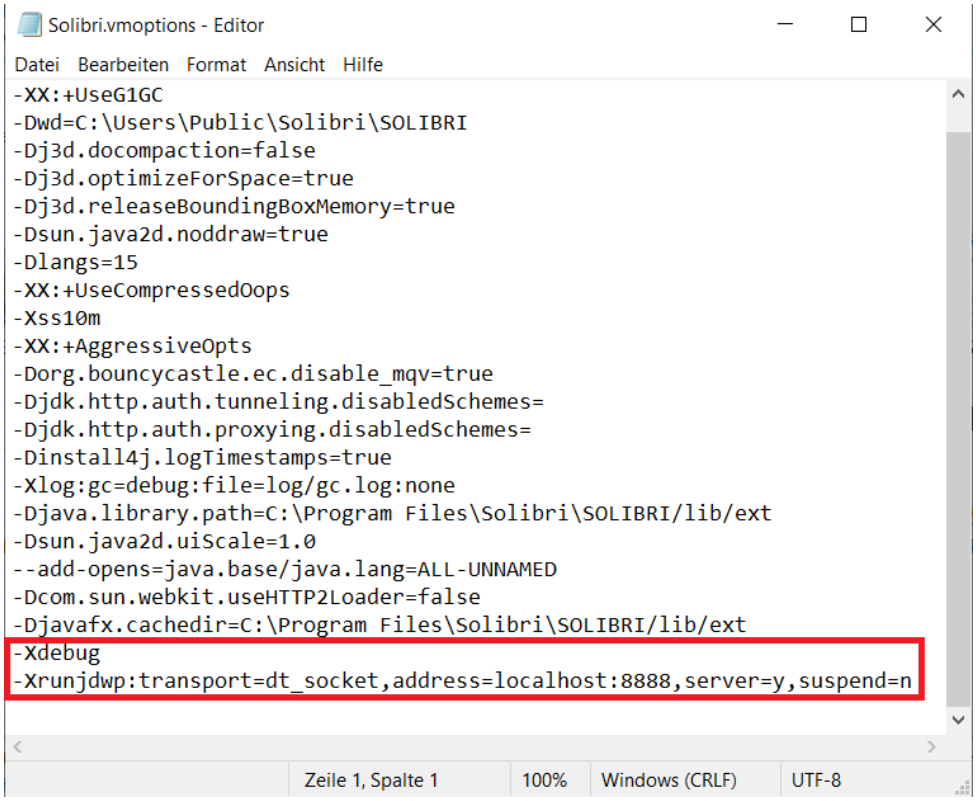

Abb. 8.1: *Solibri.vmoptions.txt*-Datei mit eingefügten Zeilen

8.2.3 Debug-Funktionen

Bevor der eigentliche Debug-Prozess beginnen kann, werden in diesem Abschnitt noch die wesentlichen Funktionen erklärt. Zu finden sind all diese in der Debug-Oberfläche in *Eclipse*. Diese kann durch klicken auf den kleinen grünen Käfer in der oberen rechten Ecke des Programms aufgerufen werden (in der Standardeinstellung von Eclipse). Mithilfe des Java-Symbols links daneben kommt man zurück zur Programmieroberfläche. Dadurch erscheinen die Debug-Funktionen in der oberen Bedienleiste (siehe Abb. 8.4). Sollten noch keine Verbindung hergestellt oder keine *Breakpoints* gesetzt sein, sind die einzelnen Buttons noch teilweise grau hinterlegt. *Breakpoints* sind Punkte an denen der Programmablauf angehalten werden soll. Die Methode in der betroffenen Zeile wird nicht mehr ausgeführt. Gesetzt werden können *Breakpoints* durch einen Doppelklick auf den linken Bereich neben der Zeilennummer und sind durch einen grünen Punkt erkennbar.

Abb. 8.4: Debug Funktionen

Die wichtigsten Funktionen sind:

1. **Breakpoints aktivieren/deaktivieren:** Diese Funktion aktiviert bzw. deaktiviert alle gesetzten *Breakpoints*.

2. **Resume:** Hat das Programm an einem *Breakpoint* angehalten, wird es durch Drücken dieses Buttons wieder bis zum nächsten *Breakpoint* oder weiter bis zum Programmende ausgeführt.

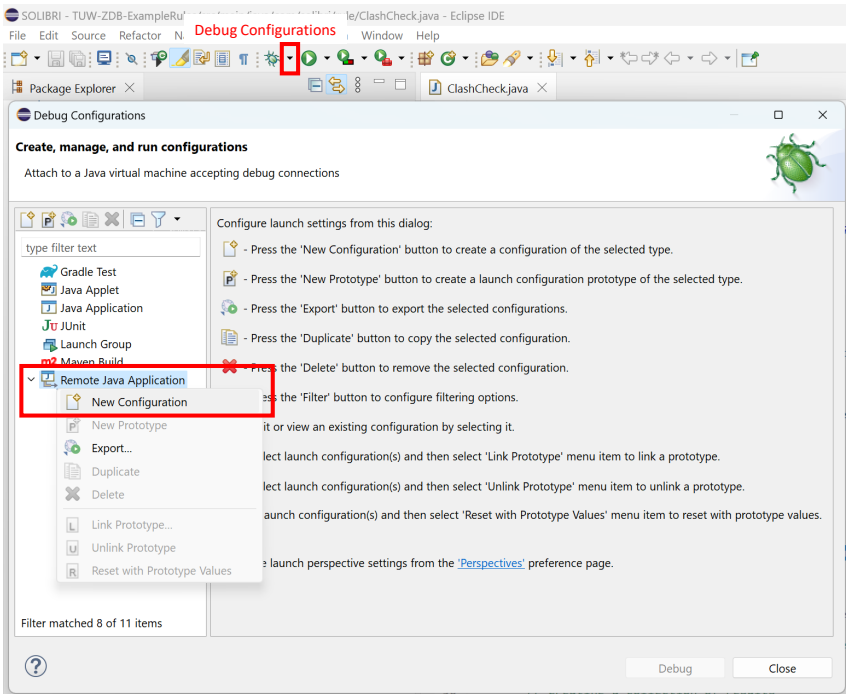

Abb. 8.2: Debug Configurations öffnen und neue Konfiguration anlegen

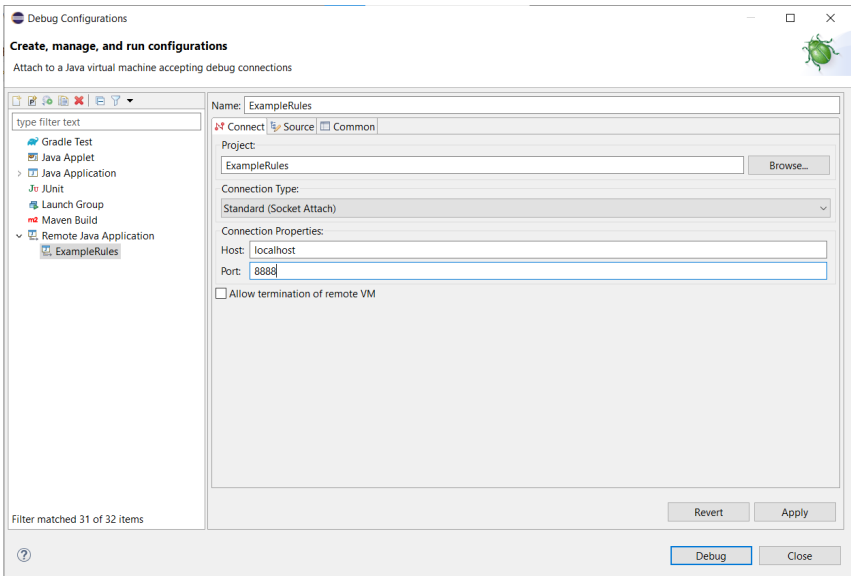

Abb. 8.3: Eingabewerte neue Konfiguration

3. **Disconnect:** Dieser Button dient zum Beenden der Verbindung zwischen *Eclipse* und Solibri. Wird dieser in einem laufende Debug-Prozess gedrückt, wird der Prozess abgebrochen und die Verbindung muss neu hergestellt werden.

4. **Step into:** Diese Funktion führt bei dem Aufruf von Methoden dazu, dass in diese gesprungen wird und näher untersucht werden kann. Sinnvoll ist diese Funktion vor allem bei eigenen Methoden. Allgemeine Java-Methoden kann man sich jedoch auch interessehalber ansehen, allerdings nicht bearbeiten. In Methoden der Solibri-API kann man nicht hineinspringen, da deren Code nicht öffentlich ist.

5. **Step over:** Dieser Button dient zum Springen von Zeile zu Zeile und zum Ausführen dieser. Wird dabei eine Methode aufgerufen, wird diese nur ausgeführt und das Ergebnis zurückgegeben, aber im Unterschied zum Punkt davor nicht näher untersucht.

6. **Step return:** Diese Funktion springt nach dem Betreten einer Methode zu deren Ende.

8.2.4 Debug-Prozess

In diesem Abschnitt wird das richtige Vorgehen beim *Debugging* erklärt.

Zu Beginn muss die Verbindung zwischen *Eclipse* und Solibri durch Ausführung der Debug-Konfiguration hergestellt werden. Danach wird in Solibri die Regel wie gewohnt vorbereitet. Bevor aber der Check in Solibri gestartet wird, muss in *Eclipse* mindestens ein *Breakpoint* gesetzt werden. Dieser soll dort platziert sein, bis zu welchen Punkt der Check das erste Mal durchlaufen werden soll. Diese Stelle kann entweder irgendwo in einer Methode liegen, wenn man den Bereich des Problems schon eingrenzen kann, oder auch direkt nach Beginn der preCheck-, check-, oder postCheck-Methode sein, damit man die gesamte Methode von Anfang an überprüfen kann. Darauffolgend kann der Check in Solibri gestartet werden. Hier werden dann nicht sofort die Ergebnisse angezeigt, sondern es können in Eclipse Schritt für Schritt die verschiedenen *Collections*, Variablen und deren Werte untersucht sowie von Methode zu Methode gesprungen werden. Sobald der erste *Breakpoint* erreicht wird, wechselt *Eclipse* in den Debug-Modus.

Ist die check-Methode für alle Komponenten fertig durchgelaufen und das *Debugging* soll nochmal gestartet werden, müssen in Solibri die Regelparameter geändert und der Check erneut gestartet werden. Es reicht auch aus, z.B. zusätzliche Komponenten hinzuzufügen und diese wieder zu löschen. Ist das *Debugging* abgeschlossen, kann die Verbindung getrennt werden und Solibri arbeitet wieder normal.

Nun wird anhand der ersten Beispielregel der Prozess erklärt. Die Regel bekommt sowohl im ersten als auch im zweiten Komponentenfilter alle Komponenten übergeben. In *Eclipse* wird der erste *Breakpoint* in Zeile 54 gesetzt, sodass nach dem Aufruf der check-Methode noch nichts weiteres ausgeführt wird. Der Zustand nach dem Beginn des Checks in Solibri ist in Abb. 8.5 zu sehen. Hier sind vor allem `this` und die übergebene Komponente `component` interessant, für welche die check-Methode nun ausgeführt wird. `this` ist ein Bezug auf die derzeit ausgeführte Regel und enthält alle Klassenvariablen und Regelparameter. Mit einem Klick auf *Step over* wird die Zeile 54 ausgeführt und die *Collection* `targets` erstellt. Mit einem Klick auf die kleinen Pfeile kann die große Anzahl an enthaltenen Komponenten untersucht werden (siehe Abb. 8.6). Weitere Klicks auf *Setp over* führen zur for-Schleife über die Hindernisse sowie zu Erstellung der *Intersections*. Ist keine *Intersection* vorhanden, führt der nächste Klick wieder zum nächsten Hindernis. Dies kann entweder für jedes Hindernis wiederholt werden, bis eine *Intersection* gefunden wird, oder es wird ein *Breakpoint* in Zeile 70 gesetzt. Bei einem Klick auf *Resume* läuft nun der Check so lange weiter, bis eine *Intersection* gefunden wird, deren Volumen ausreichend groß ist. Zu dieser *Intersection* können dann die ausgehende Komponente, das Hindernis und die Eigenschaften der *Intersection* (z.B. deren Volumen) selbst näher untersucht werden (siehe Abb. 8.7). Beim nächsten Klick auf *Resume* läuft der Check wieder bis zum *Breakpoint* in Zeile 70

oder in Zeile 54, wenn schon alle Hindernisse abgearbeitet wurden. Je nach Arbeitsfortschritt kann mit weiteren Funktionen fortgefahren werden oder das *Debugging* beendet werden.

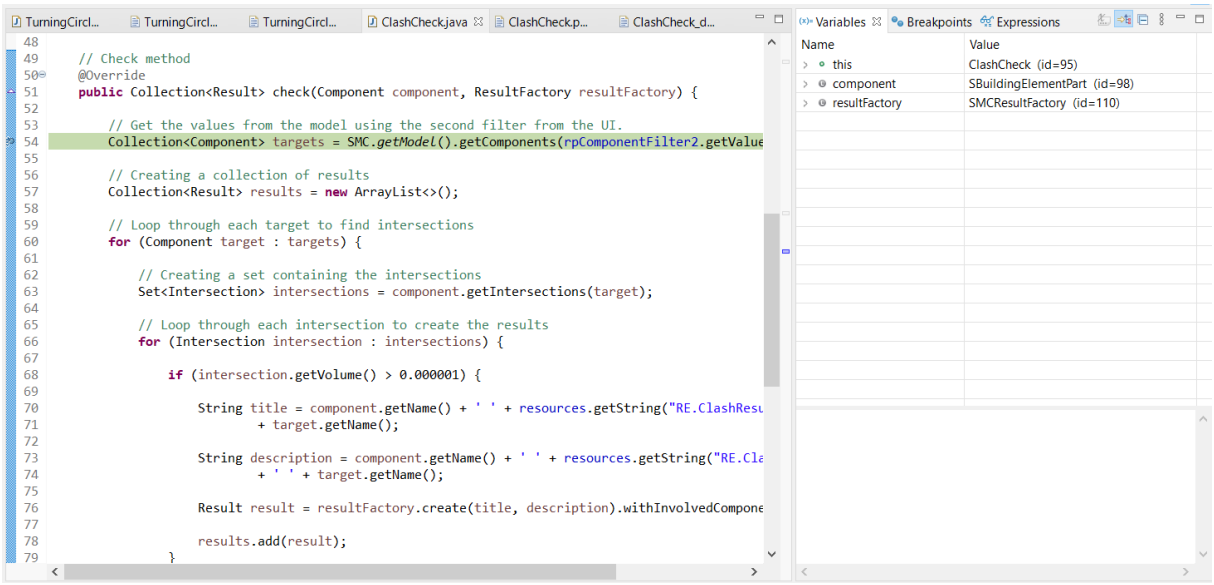

Abb. 8.5: Debug Funktionen

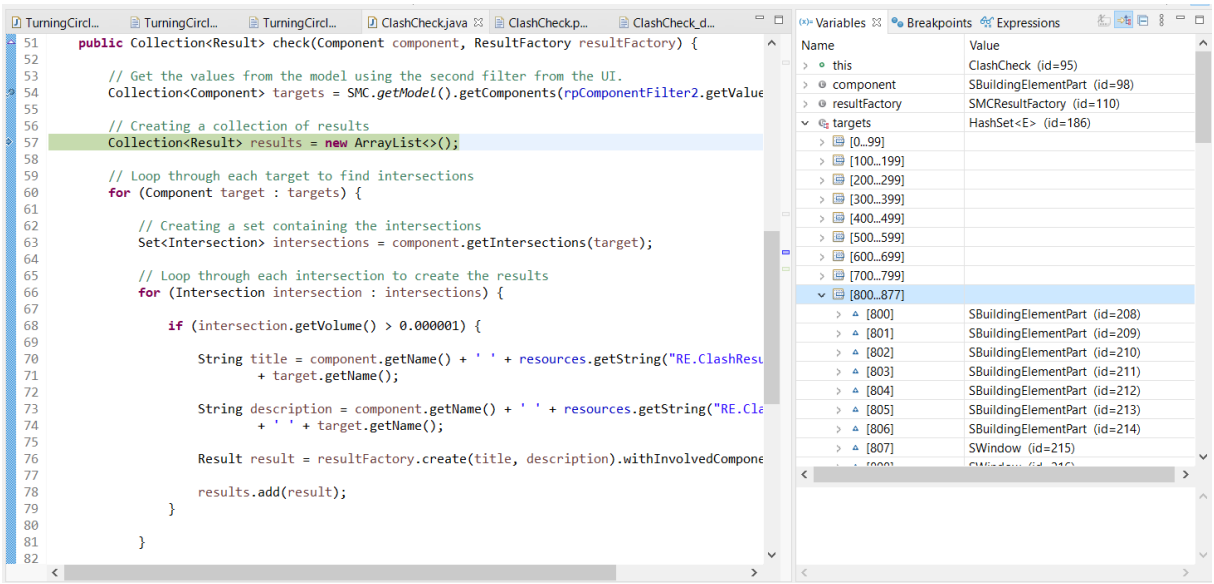

Abb. 8.6: Debug Funktionen

8.3 Geometrie

Solibri bietet in der Paketserie `com.solibri.geometry` diverse Interfaces, Klassen und Methoden für den Umgang mit Geometrie. Dieses Kapitel gibt einen Überblick über die wichtigsten Möglichkeiten in diesem Bereich.

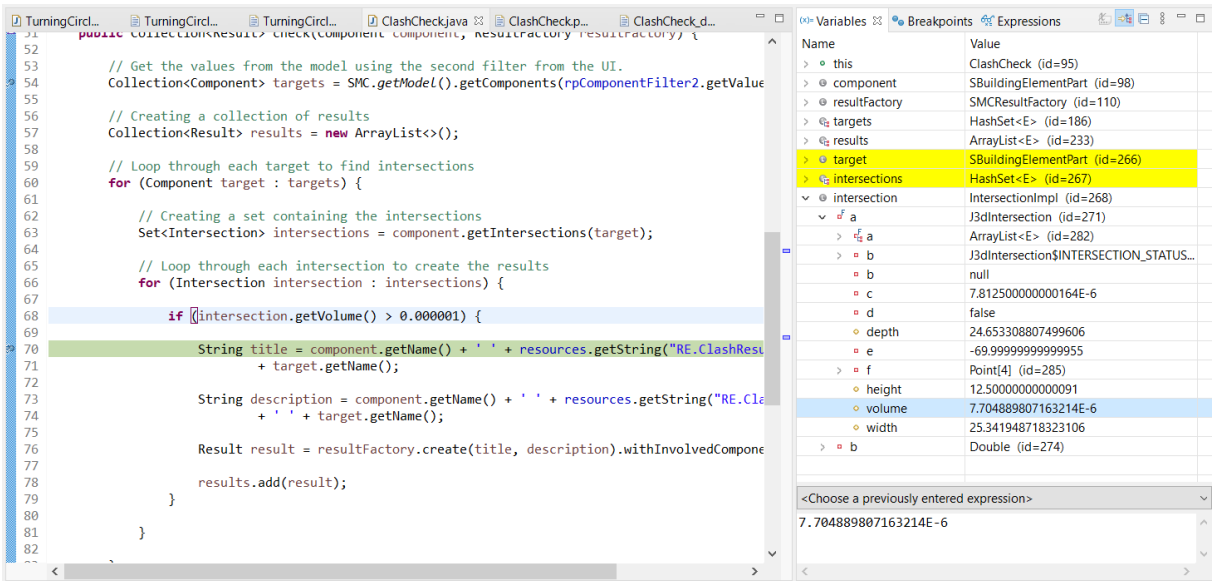

Abb. 8.7: Debug Funktionen

8.3.1 Vektoren

Vektoren bilden das kleinste Element vieler geometrischer Problemstellungen (Beispiele siehe Kapitel 9). Prinzipiell wird in 2d- und 3d-Vektoren unterschieden. 2d-Vektoren befinden sich immer in der XY-Ebene, wohingegen 3d-Vektoren räumlich definiert werden können. Ebenfalls unterscheidet Solibri in modifizierbare (vorangestelltes „M“) und nicht modifizierbare Vektoren (dieses Konzept kommt auch bei vielen anderen Klassen zur Anwendung). Werden modifizierbare Vektoren verändert, wird jenes Objekt verändert, auf welchem die Methode aufgerufen wurde. Nicht modifizierbare Vektoren können nicht verändert werden. Methoden, die eine Veränderung vermuten lassen (z. B. Skalieren), erstellen stattdessen einen neuen modifizierbaren Vektor. Ebenso können aus ursprünglich nicht modifizierbare Vektoren einfach neue modifizierbare Vektoren deklariert werden. Folgender Beispielcode zeigt einfache Methoden von Vektoren:

```
// non modifiable vectors
Vector2d vector2d = Vector2d.create(0,1);
Vector3d vector3d = Vector3d.create(1,1,1);
// scaling a Vector2d creates a MVector2d
MVector2d vector2d_scaled = vector2d.scale(factor);

// modifiable vectors
MVector2d mVector2d = MVector2d.create(1,2);
MVector3d mVector3d_1 = MVector3d.create(2,2,2);
// scaling a MVector2d modifies the object
mVector2d.scaleInPlace(factor);

// convert non modifiable to modifiable
MVector3d mVector3d_2 = MVector3d.create(vector3d);
```

Allgemein sind vor allem folgende Methoden im Zusammenhang mit Vektoren wichtig:

- Koordinaten: `.getX(v1)`, `.getY(v1)`, `.getZ(v1)`
- Länge: `.getLength(v1)`
- Skalieren: `.scale(double ...)`
- Normieren: `.normalize(v1)`

- Umkehren: `.negate()`
- Rechenoperationen: [-] `.subtract(v2)`, [+] `.add(v2)`
- Skalarprodukt: `.dotProduct(v1,v2)`
- Kreuzprodukt: `.crossProduct(v1,v2)`

8.3.2 Segmente

Erstellte Vektoren beziehen sich immer auf den Koordinatenursprung. Um eine Linie mit beliebigem Anfangs- und Endpunkt erstellen zu können, muss diese als Segment definiert werden. Dabei wird wieder zwischen 2d und 3d bzw. modifizierbar und nicht modifizierbaren Segmenten unterschieden:

```
// segments
Segment2d segment2d = Segment2d.create(vector2d, mVector2d);
MSegment3d segment3d = MSegment3d.create(mVector3d_2, mVector3d_1);
```

8.3.3 Polygone

Polygone sind als 3d- und 2d-Polygone in Solibri implementiert und werden in modifizierbare und nicht modifizierbare Polygone unterschieden. Zur Erstellung werden die Vektoren in eine Liste gespeichert, mit welcher das Polygon erstellt werden kann. Zu einer der wichtigsten Methoden von Polygonen zählt der `EdgeIterator`. Dieser durchläuft alle Kanten (Segmente) eines Polygons. Weiter können `.getVertices()` zur Ermittlung aller Ecken und `.getCentroid()` zu Ermittlung des Schwerpunkts hilfreich sein.

```
// ArrayList with 2dVectors
List<MVector3d> vectorList = new ArrayList<>();

// Add vectors to list
vectorList.add(mVector3d_1);
vectorList.add(mVector3d_2);

// polygon3d
Polygon3d polygon3d = Polygon3d.create(vectorList);

// EdgeIterator to get all edges to segments
Iterator<Segment3d> edgeIterator = polygon3d.getEdgeIterator();
while (edgeIterator.hasNext()) {
   Segment2d edge = edgeIterator.next();
   //do something
}
```

8.3.4 Flächen

Flächen sind als 2d-Element in der XY-Ebene definiert. Es wird wie bei anderen Geometrieobjekten in modifizierbare und nicht modifizierbare Flächen unterschieden. `Areas` können aus einer Liste von Eckpunkten (Vektoren), einem Polygon oder anderen `Areas` erstellt werden. Ein `Area` kann Löcher besitzen oder sogar aus zwei `Areas` bestehen, die sich gar nicht berühren. Über den modifizierbaren `Area` können Überschneidungsflächen mit anderen `Areas` ermittelt oder andere `Areas` zu einem `Area` hinzugefügt werden. Durch diese Funktionalitäten ist der `Area` eine sehr mächtige Klasse.

```
//Create Areas from vector list
Area area1 = Area.create(vectorList1);
Area area2 = Area.create(vectorList2);
Area area3 = Area.create(vectorList3);

//Create MArea from Area
MArea mArea1 = MArea.create(area1);

//Add other Area to this Area
mArea1.add(area2);

//Intersect Areas
mArea1.intersect(area3);
```

8.3.5 TriangleMesh

Eine besonders wichtige Rolle in vielen Prüfregelcodes nimmt das `TriangleMesh` ein. Ein `TriangleMesh` ist ein Gittergewebe bestehend aus Dreiecken, das die Oberfläche eines dreidimensionalen Objekts repräsentiert. `TriangleMeshes` lassen sich individuell und einfach erstellen, indem eine Liste mit Polygonen oder Dreiecken aller Begrenzungsflächen als Grundlage herangezogen wird. Über `.getTriangleMesh` kann ein `TriangleMesh` auch sehr einfach aus einer bestehenden Komponente erstellt werden. Dies bietet zum Beispiel die Möglichkeit ein „Überschneidungsmesh“ der Komponente mit einem frei erstellten `TriangleMesh` zu generieren. Ein weiterer großer Vorteil ist, dass `TriangleMeshes` in Ergebnissen der Prüfregel visuell dargestellt werden können. Dazu muss zunächst das `TriangleMesh` in eine `Collection` aus Dreiecken umgewandelt werden. Dann kann daraus ein visualisierbares `Mesh` erstellt werden, welches in der Ausgabe angezeigt wird (siehe zweite Beispielregel in Kapitel 9).

```
//Create a TrianlgeMesh
TriangleMesh triangleMesh = TriangleMesh.create(polygonList);

//Retrieve a TriangleMesh from a Component
TriangleMesh componentTriangleMesh = component.getTriangleMesh();

//Determine Intersection
TriangleMesh intersection = triangleMesh.intersection(componentTriangleMesh
    );
```

8.3.6 Ableitung von Geometrien von Komponenten

Da die Komponenten des ifc-Modells genaue Koordinaten im Raum aufweisen, bilden diese eine gute Grundlage für geometrische Problemstellungen. Dabei können die Komponenten mit diversen, bereits in Solibri implementierten Methoden 2-dimensional und 3-dimensional als Flächen, Vektoren, Gittergewebe, etc. aufgelöst werden. Um die entsprechende Komponente 2-dimensional aufzulösen wendet man `.getFootprint()` an, wodurch ein Fußabdruck der Komponente erstellt wird. Ausgehend von diesem *Footprint* können über `.getArea()` und `.getArea().getLargestPolygon()` der Area und das umschließende Polygon des Footprints ermittelt werden. Vorsicht geboten ist bei direktem Zugriff auf das Polygon vom *Footprint* über `.getOutline()`. Diese Methode ist in der derzeitigen API-Version fehlerhaft. Sie erstellt ein Polygon, wo der erste und letzte Eckpunkt der gleiche sind. Dadurch erhält man bei Zugriff auf die Segmente über den *EdgeIterator* als letztes ein degeneriertes Segment. Da nicht klar ist, wann dieser Fehler behoben wird, sollte diese Methode nicht verwendet werden. Als Basis der 3-dimensionalen Auflösung empfiehlt sich `.getTriangleMesh()`.

Weitere wichtige Methoden im Zusammenhang mit der Geometrie von Komponenten sind:

- `.getGlobalTopElevation()`: ermittelt den höchsten Punkt einer Komponente.
- `.getGlobalBottomElevation()`: ermittelt den niedrigsten Punkt einer Komponente.
- `.getBoundingBox()`: gibt die umgrenzende Box eines Bauteils wider als *AABB3d* (AxisAlignedBoundingBox3d) wider. Die Außenkanten einer *AABB3d* orientieren sich an den globalen Koordinatenachsen und umschließen die Komponente vollständig. Sie kann daher für grobe Geometriechecks verwendet werden.

8.4 Visualisierung

Mithilfe der Visualisierungsoption können Komponenten, Abmessungen, Flächen, Winkel und ähnliches in einem Prüfergebnis dargestellt beziehungsweise farblich hervorgehoben werden. Durch die graphische Darstellung stehen eine Vielzahl von Möglichkeiten zur Verfügung, Ergebnisse besser zu kommunizieren.
Visualisierungen werden in der Solibri API über Lambda-Ausdrücke erstellt. Lambda-Ausdrücke sind kurze Code-Blöcke die wie Methoden Parameter als Eingangsgrößen definieren und anschließend einen Wert als Ergebnis liefern. Der entscheidende Unterschied zu Methoden ist, dass diese keinen eigenen Namen benötigen. Somit kann die Implementierung innerhalb einer Methode erfolgen. Ein weiteres typisches Merkmal ist der Pfeil, der die Parameter vom Ausdruck trennt.

```
(parameter1, parameter2) -> { code block }
```

8.4.1 Visualisierung von Abständen, Flächen und Winkeln

Jedem Ergebnis kann ein `VisualizationItem` übergeben werden. Dabei handelt es sich um ein List-Objekt, das abhängig vom zu visualisierenden Objekt einen geringfügig anderen Aufbau besitzt. Soll zum Beispiel ein Abstand visualisiert werden, ist die Angabe von Startpunkt und Endpunkt notwendig. Die Variable wird wie folgt initialisiert:

```
List<VisualizationItem> distanceVisualisation = VisualizationItem.
    createDimension(startPoint, endPoint);
```

Falls zu einem Ergebnis mehrere Visualisierungen notwendig sind, empfiehlt es sich diese in einer Liste zu speichern. Die Übergabe dieser Liste an das Ergebnis sieht wie folgt aus:

```
List<List<VisualizationItem>> distanceVisualizationList = new ArrayList
    <>();
List<VisualizationItem> distanceVisualisation1 = VisualizationItem.
    createDimension(startPoint1, endPoint1);
List<VisualizationItem> distanceVisualisation2 = VisualizationItem.
    createDimension(startPoint2, endPoint2);
distanceVisualizationList.add(distanceVisualisation1);
distanceVisualizationList.add(distanceVisualisation2);

Result result = resultFactory.create(resultName, resultDescription)
.withVisualization(visualization -> {
   for(List<VisualizationItem> distanceVisualization :
       distanceVisualizationList) {
      visualization.addVisualizationItems(distanceVisualization);
   }
});
```

8.4.2 Sonstige Visualisierungen

Neben `VisualizationItems` (Abstände, Flächen, Winkel) können auch Linien, Meshes, Punkte und Texte dargestellt werden. Die Übergabe an das Ergebnis ist nahezu identisch zu einem `VisualizationItems`. Der Einzige unterschied ist, dass kein extra Objekt(`VisualizationItem`) dafür erstellt werden muss. Der nachfolgende Code-Block zeigt beispielhaft die visuelle Darstellung eines `TriangleMeshes`.

```
    Mesh meshVisualization = Mesh.create(triangleMesh.toTriangleCollection()
        );

    Result result = resultFactory.create(resultName, resultDescription)

    .withVisualization(visualization -> {
       visualization.addVisualizationItem(meshVisualization);
    })
});
```

8.5 Relation

Oftmals ist es notwendig von bestimmten Komponenten des Modells auf andere, in einer Beziehung zueinander stehende, Komponenten zu schließen. In IFC und der Solibri-API wird dies durch *Relations* ermöglicht. Der genaue Vorgang wird detailliert im Codeausschnitt gezeigt. Zuerst wird eine Relation-Objekt `exampleRelation` mit einem bestimmten Typ `exampleType` (Verweis auf die IFC-Relation, z. B. NEAREST_SPACES) und einer bestimmten Richtung `exampleDirection` bestimmt. Dann können mit dieser Relation von einer Komponente `component` die dazu in der richtigen Beziehung stehenden Komponenten `exampleComponents` abgerufen werden.

```
    Relation.Type exampleType = Relation.Type.NEAREST_SPACES;
    Relation.Direction exampleDirection = Relation.Direction.FORWARD;

    Relation exampleRelation = Relation.of(exampleType, exampleDirection);

    Collection<Component> exampleComponents = component.getRelated(
        exampleRelation);
```

Es gibt eine Vielzahl an Typen für Beziehungen, weshalb hier nur eine Auswahl der häufig verwendeten erklärt wird.

- **NEAREST_SPACES:** Zu einer gegebenen Komponente werden die nächstgelegenen Räume ermittelt.
- **FILLS:** Zu einem ausfüllenden Objekt (z.B. Tür, Fenster) wird dessen Öffnung bestimmt.
- **VOIDS:** Zu einer Öffnung wird die Wand bestimmt, in der diese Öffnung vorhanden ist.
- **BOUNDED_BY:** Dieser Typ gibt zu einem Raum dessen begrenzenden Flächen an.
- **CONTAINS:** Es werden zu einem Geschoss die darin enthaltenen Komponenten angegeben.
- **CONTAINS_REFERENCED:** Es werden zu einem Raum die darin enthaltenen Komponenten angegeben.
- **DECOMPOSES:** Dieser Typ beschreibt die Einzelteile einer zusammengesetzten Komponente.

- **DEFINES_BY_TYPE:** Es wird der Typ einer Komponente angegebene.

Beziehungen besitzen eine von drei möglichen Richtungen. Zu beachten ist, dass die Richtung immer abhängig vom jeweiligen Typ der Relation ist. Es gibt folgende drei mögliche Richtungen:

- **FORWARD:** Diese Richtung weist von der ausgehenden Komponente zur gesuchten Komponente. Ist zum Beispiel ein Fenster gegeben und wird die zugehörige Öffnung gesucht, geht man ausgehenden von diesem Fenster zur Öffnung, die ausgefüllt wird (`Relation.Type.FILLS`).
- **BACKWARD:** Diese Richtung weist in die entgegengesetzte Richtung der Relation von der ausgehenden Komponente zur gesuchten Komponente. Im Unterschied zu vorhin wird diese Richtung verwendet, wenn eine Öffnung gegeben ist und das ausfüllende Fenster gesucht wird (ebenfalls `Relation.Type.FILLS`).
- **BOTH:** Diese Richtung ist die Zusammenfassung und geht in beide Richtungen. Somit ist sie universell anwendbar.

Ein weiteres Beispiel zur Veranschaulichung des Unterschiedes der Richtung: Ist eine Komponente gegeben und man will den nächstgelegenen Raum wissen, benötigt die Relation die Richtung `FORWARD`. (`Relation.Type.NEAREST_SPACES, Relation.Direction.FORWARD`). Hat man einen Raum gegeben und möchte wissen, von welchen Komponenten dieser Raum der nächstgelegene ist, geht man von dem gegebenen Raum rückwärts zu den gesuchten Komponenten (`Relation.Type.NEAREST_SPACES, Relation.Direction.BACKWARD`).

8.6 Project-Object-Model (POM)

Das POM beziehungsweise das POM-File ist die Basis der Maven-Funktionalität. Dabei handelt es sich um eine XML-Datei, die Informationen zu Abhängigkeiten, Konfigurationen und andere Projektinformationen enthält. Maven geht das POM-File Zeile für Zeile durch um die definierten Aufgaben auszuführen.
Prinzipiell befinden sich alle Angaben innerhalb einer *<project> </project>* Umgebung. Im Folgenden Punkt werden die einzelnen Elemente des POM-Files aus der Regelvorlage von Solibri näher erläutert

8.6.1 Allgemeine Angaben

```
<modelVersion>4.0.0</modelVersion>

<packaging>jar</packaging>

<!-- Change this to match your company name -->
<groupId>com.solibri</groupId>

<!-- Change this to match your project name -->
<artifactId>smc-api-template</artifactId>

<!-- Change this to match your project name -->
<name>Solibri API Template Project</name>

<version>1.0.0</version>
```

Angaben zu *<modellVersion>* und *<packaging>* können aus der Vorlage übernommen werden. Angaben zu *<groupId>* verändern den Namen des Packages in Eclipse. Eine Änderung muss

jedoch vor dem Maven-Import erfolgen. Nachträglich kann der Package-Name nur mehr in der IDE selbst geändert werden. *<artifactId>* bestimmt den Projektnamen. Wird der Maven-Export gestartet und die .jar-Datei erstellt, trägt diese Datei den gewählten Projektnamen. Die Elemente *<name>* und *<version>* stellen sichtbare Angaben in Solibri dar. Ersteres definiert den Namen des Ordners in Solibri, welcher die einzelnen Regeln enthält. Das zweite Element stellt eine einfache Art der Versionierung dar.

8.6.2 Properties

```
<properties>
<!-- Solibri installation path on Windows -->
<smc-dir>C:/Program Files/Solibri/SOLIBRI</smc-dir>

<maven-compiler-plugin.version>3.8.0</maven-compiler-plugin.version>
<maven-jar-plugin.version>3.1.0</maven-jar-plugin.version>
</properties>
```

Das Element *<properties>* fordert den Nutzer auf, den Installationspfad für Solibri anzugeben. Sollte wie in 2.1 empfohlen der Standardinstallationspfad für Solibri gewählt werden, kann dieser Punkt übersprungen werden.

8.6.3 Profiles

Unter *<profiles>* können für verschiedene Betriebssysteme unterschiedliche Profile angelegt werden. Bereits vordefiniert ist ein Profil macOs. Für jedes weitere Betriebssystem kann das Profil kopiert und unter *<smc-dir>* der Installationspfad für Solibri angegeben werden.

```
<profiles>
   <profile>
      <id>platform-mac</id>
      <activation>
         <os>
         <family>mac</family>
         </os>
      </activation>
      <properties>
         <!-- Solibri installation path on macOS -->
         <smc-dir>/Applications/Solibri</smc-dir>
      </properties>
   </profile>
</profiles>
```

8.6.4 Dependencies - Maven Repository

Maven ermöglicht die einfache Einbindung von externen Bibliotheken. Dafür wird automatisch während dem Buildprozess das Remote-Repository (Maven Repository) angesteuert, und die angegeben Bibliotheken heruntergeladen. In *<dependencies>* kann der Nutzer die benötigten externen Bibliotheken als Dependency angeben. Solibri gibt selbst drei Dependencies vor, um auf die Solibri interne Methoden und Klassen zugreifen zu können. Diese dürfen nicht geändert oder gelöscht werden, da sonst Solibri-interne Funktionen nicht mehr funktionieren. Sollten weitere Bibliotheken benötigt werden, können diese am einfachsten von

`https://mvnrepository.com/`

bezogen werden. Die nötigen Angaben sind dort vordefiniert und müssen nur kopiert und in das POM-File eingefügt werden. Allerdings muss bei der Wahl der Version drauf geachtet werden, dass Solibri diese umsetzen. Dazu gibt es jedoch keine Vorschrift und es muss für jede einzelne Bibliothek überprüft werden.

```
<dependencies>
   <dependency>
      <groupId>com.solibri.smc</groupId>
      <artifactId>smc-api</artifactId>
      <version>1.0</version>
      <scope>system</scope>
      <systemPath>${smc-dir}/lib/smc-api.jar</systemPath>
   </dependency>
   <dependency>
      <groupId>com.solibri.smc</groupId>
      <artifactId>smc-geometry</artifactId>
      <version>1.0</version>
      <scope>system</scope>
      <systemPath>${smc-dir}/lib/smc-geometry.jar</systemPath>
   </dependency>
   <!-- slf4j-api can be used for logging capabilities. -->
   <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-api</artifactId>
      <version>1.0</version>
      <scope>system</scope>
      <systemPath>${smc-dir}/lib/slf4j-api.jar</systemPath>
   </dependency>
</dependencies>
```

8.6.5 Build - Plugins

Unter *<plugins>* sind notwendige plugins definiert. Diese dürfen nicht geändert werden. Bei Definition externen Bibliotheken in den Dependencies muss folgendes *<plugin>* hinzugefügt werden:

```
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-shade-plugin</artifactId>
<version>3.2.1</version>
<executions>
<execution>
<phase>package</phase>
<goals>
<goal>shade</goal>
</goals>
<configuration>
</configuration>
</execution>
</executions>
</plugin>
```

An dieser Stelle noch einmal der Hinweis, dass sich die Version in Zukunft ändern kann.

8.7 Excel-Verwendung

Für das Exportieren von Daten und das Abspeichern von Ergebnissen hat sich die Anbindung zu Excel als sehr nützlich dargestellt. Um eine Schnittstelle zu erhalten, bietet sich die Java-Bibliothek Apache POI Common an. Diese ermöglicht den Zugriff auf Microsoft Format Files, und damit auch auf .xlsx-Files.
Wie in 8.6 beschrieben, muss die Maven-Dependency der Apache POI Common Bibliothek im POM-File hinzufügt werden. Bezug der aktuellen Version ist unter

`https://mvnrepository.com/artifact/org.apache.poi/poi`

möglich. Allerdings nutzt Solibri diese Bibliothek systemintern ebenfalls, weshalb nicht immer die aktuellste Version genutzt werden kann. Wird eine aktuellere Version verwendet als jene von Solibri, können folgende Fehler auftreten:

- Solibri lässt sich nicht mehr starten, stattdessen kommt eine zufällige Fehlermeldung
- Es werden keine Regelsets mehr im Rulesetmanager angezeigt

Sollten diese Fehler auftreten, muss eine frühere Version gewählt werden.

8.7.1 Arbeiten mit Excel

Im wesentlichen gibt es drei grundlegende Arbeitsschritte mit Excel-Files:

- Excel-File erstellen
- bestehende Excel-File einlesen
- in bestehende Excel-Files schreiben

Prinzipiell gilt das Öffnen/Schließen des Excel-Files als am Zeit intensivsten. Deswegen sollten alle Funktionen, die direkt auf das Excel-File angewiesen sind, in einem Methodenblock gesammelt aufgerufen werden. Die folgenden vier Angaben sind für jedes Excel-File notwendig:

- Dateipfad
- Dateiname
- Tabellenblatt
- Dateiformat

Bis auf das Dateiformat (.xlsx) sind die Angaben benutzerdefiniert beliebig wählbar und lassen sich am Besten durch Eingaben im UI steuern. Sobald diese Angaben zur Verfügung stehen, kann auch schon ein Excel-File erstellt werden (siehe nachfolgender Code-Block). Falls bereits ein Excel-File mit den gleichen Angaben existiert wird, dieses mit einem leeren File überschrieben.

```
String sheetName = "Tabelle1";
String fileName = "database";
String location = "C:\\Users\\Benutzername\\Desktop";
String format = "xlsx";

String directory = location + "\\\\" + fileName + "." + format;
try {
   Workbook workbook = new XSSFWorkbook();

   Sheet sh = workbook.createSheet(sheetName);
```

```
        FileOutputStream fileOut = new FileOutputStream(directory);
        workbook.write(fileOut);
        workbook.close();
        fileOut.close();
    }

    catch (Exception e) {
        e.printStackTrace();
    }
```

Existiert unter den gegeben Angaben ein File, können aus diesem Zellen eingelesen oder beschrieben werden. Die nachfolgenden Code-Blöcke zeigen, wie mit einer einzelnen Zelle umgegangen wird. Dies lässt sich allerdings unter Verwendung von for-Schleifen leicht erweitern um ganze Spalten/Zeilen zu bearbeiten. Der nachfolgende Codeblock zeigt, wie die Zelle A1 aus der Datei database eingelesen wird.

```
String sheetName = "Tabelle1";
String fileName = "database";
String location = "C:\\Users\\Benutzername\\Desktop";
String format = "xlsx";
String cellReference = "A1";

String directory = location + "\\\\" + fileName + "." + format;
try {
    FileInputStream file = new FileInputStream(new File(directory));
    Workbook workbook = new XSSFWorkbook(file);
    DataFormatter dataFormatter = new DataFormatter();
    Sheet sh = workbook.getSheet(sheetName);

    CellReference crA1 = new CellReference(cellReference);
    Row rowA1 = sh.getRow(crA1.getRow());
    Cell cellA1 = rowA1.getCell(crA1.getCol());

    returnable = dataFormatter.formatCellValue(cellA1);
    workbook.close();
}

catch (Exception e) {
    e.printStackTrace();
}
```

Ein wesentlicher Unterschied zwischen einlesen und befüllen besteht darin, dass beim Befüllen darauf geachtet werden muss, ob eine Reihe bereits beschrieben wurde oder nicht. Dieser Umstand lässt sich am einfachsten mithilfe eines try-catch-Blocks umgehen (siehe nachfolgender Codeblock). Zuerst wird versucht, die Reihe zu finden. Sollte diese noch nicht existieren, springt der Code in den Catch-Teil und erstellt die Reihe.

```
String sheetName = "Tabelle1";
String fileName = "database";
String location = "C:\\Users\\Benutzername\\Desktop";
String format = "xlsx";
String cellReference = "A1";

String directory = location + "\\\\" + fileName + "." + format;
try {
    FileInputStream file = new FileInputStream(new File(directory));
    Workbook workbook = new XSSFWorkbook(file);
    Sheet sh = workbook.getSheet(sheetName);
```

```
        CellReference crA1 = new CellReference(cellReference);

        try {
            Cell cell = sh.getRow(crA1.getRow()).createCell(crA1.getCol());

            cell.setCellValue(name);
        }
        // if the row doesn't exist
        catch (Exception e) {
            Cell cell = sh.createRow(crA1.getRow()).createCell(crA1.getCol());
            cell.setCellValue(name);
        }
        workbook.write(fileOut);
        workbook.close();
        fileOut.close();
    }

    catch (Exception e) {
        e.printStackTrace();
    }
```

8.8 Informationen aus dem IFC-Header abrufen

In dem Header einer IFC-Datei sind einige interessante Informationen zum Modell enthalten (siehe Abb. 8.8). Für verschiedene Anwendungsfälle können beispielsweise das Speicherdatum, das Gewerk oder das verwendete IFC-Schema erforderlich sein. In Solibri sind diese Informationen *PropertySets* des Projekts zugeordnet. Als Beispiel wird hier gezeigt, wie das Speicherdatum abgerufen werden kann. Es befindet sich in dem PropertySet *IFC FILE NAME* in der Eigenschaft *TIME_STAMP* (siehe Abb. 8.9).

```
ISO-10303-21;
HEADER;FILE_DESCRIPTION(('ViewDefinition [ReferenceView_V1.2]','ExchangeRequirement [Archit
type objects: Off]','Option [Element Properties: All]','Option [Building Material Properti
FILE_NAME('C:\\Users\\Simon\\Desktop\\Ohne Titel.ifc','2021-05-31T14:50:15',('Architect'),(
FILE_SCHEMA(('IFC4'));
ENDSEC;
```

Abb. 8.8: Header Informationen eines Testmodells im Texteditor

Um die Information über die API abrufen zu können, muss das Element der Klasse *IfcProject* abgerufen werden. Die Klasse eines Elements ist im *IfcEntityType* abgespeichert, welcher für jedes Element über die API-Methode *getIfcEntityType()* abgerufen werden kann. Nachdem das Element gefunden ist, kann auf seine *PropertySets* und *Properties* zugegriffen werden. Der Programmcode dazu ist nachfolgend abgebildet. Wichtig ist dabei zu beachten, dass sich die Schreibweise der *PropertySets* und der *Properties* im Code von jener in Solibri unterscheidet (vergleiche Abb. 8.8 und den nachfolgenden Codeausschnitt). Um die erforderliche Bezeichnung für den Code zu ermitteln, kann man sich alle *PropertySets* bzw. *Properties* des Elements in einem Testergebnis ausgeben lassen.

```
@Override
public Collection<Result> check(Component component, ResultFactory
    resultFactory) {

    // Creating a collection of results that will be returned at the end
        of the clashCheck method
    Collection<Result> results = new ArrayList<>();
```

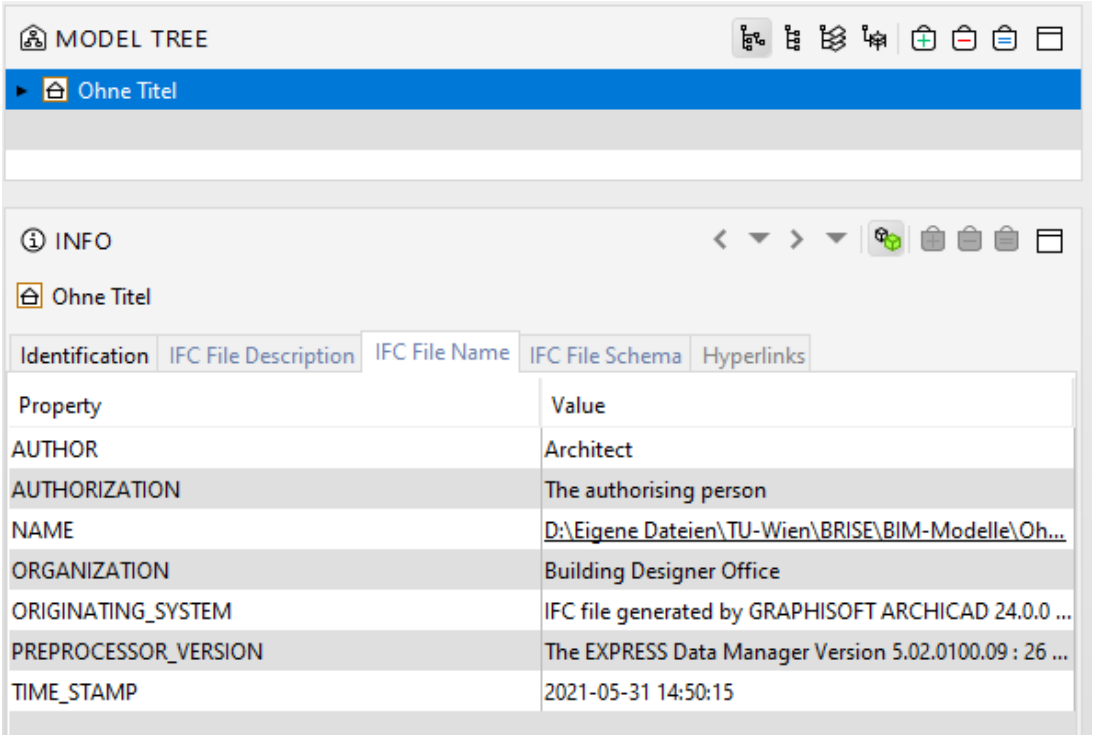

Abb. 8.9: Header Informationen eines Testmodells in Solibri

```
    //Find the component, that is from IFCEntityType.IfcProject
    Component projectComponent = SMC.getModel().getComponents(component1
        -> component1.getIfcEntityType().isPresent() && component1.
        getIfcEntityType().get().equals(IfcEntityType.IfcProject)).stream
        ().findFirst().get();

    //Retrieve all PropertySets with the name "FILE_NAME"
    Collection<PropertySet> propertySets = projectComponent.
        getPropertySets("FILE_NAME");

    //Get the first PropertySet (there should only be one)
    PropertySet propertySet = propertySets.iterator().next();
    //Get the Property with the name "time_stamp"
    Property property = propertySet.getProperty("time_stamp").get();
    //create a result
    Result result = resultFactory.create(propertySet.getName(), property.
        getValueAsString());
    results.add(result);

    return results;
}
```

9 Beispielregel 2

Das Anwendungsziel der zweiten Beispielregel ist die Überprüfung eines Wendekreises vor ausgewählten Türen. Dafür müssen die zu überprüfenden Türen sowie die möglichen Hindernisse angegeben werden. Des Weiteren müssen der Durchmesser des Wendekreises und die Höhe des daraus erzeugten Zylinders gewählt werden.

9.1 UI

Das UI der zweiten Beispielregel besteht aus zwei Komponentenfiltern sowie einem Container zur Eingabe der Größe des Wendekreises.

1. In den ersten Komponentenfilter müssen die Türen eingegeben werden, vor welchen der Wendekreis überprüft werden soll. Andere eingegebene Komponenten führen zu einer Fehlermeldung.
2. Der zweite Komponentenfilter dient zur Definition der zu beachtenden Hindernisse.
3. Nach den beiden Komponentenfiltern kommt ein vertikaler Container zur Eingabe der Größe des Wendekreises und des zugehörigen Zylinders.
   a) Zuerst wird der Durchmesser des Wendekreises in der gewünschten Einheit eingegeben.
   b) Dann kann innerhalb eines weiteren vertikalen Containers die Auswahl getroffen werden, ob die Höhe des Zylinders der Türhöhe entsprechen soll oder selbst festgelegt wird
   c) Bei Auswahl der zweiten Möglichkeit muss der Default-Wert von 0 mm überschrieben werden.
   d) Die Grafik stellt die Platzierung des Wendekreises auf beiden Seiten der Tür dar.

9.2 Erklärung Programmcode

Der Beginn der zweiten Beispielregel besitzt denselben Aufbau wie jener der ersten Beispielregel. Zuerst werden alle Importe der benötigten Packages durchgeführt. Dann werden die Namenskonstanten und *Ruleparameter* erstellt sowie die Ressourcen abgerufen. Aufgrund der vielen Funktionen der zweiten Beispielregel ergeben sich auch mehr Importe und Definitionen als bei der ersten Beispielregel. Anzumerken ist, dass die Namenskonstanten mit ihrem Namen gleichzeitig auch den Typ des Parameters (z.B. Filter, Double ...) beschreiben. Auch die *Ruleparameter*, deren genaue Funktionen bei der Erklärung des UI näher beschrieben werden, werden genauso wie bei der ersten Beispielregel erstellt.

```
1 package com.solibri.rule;
2
3 import java.util.ArrayList;
4 import java.util.Arrays;
5 import java.util.Collection;
6 import java.util.Collections;
7 import java.util.List;
```

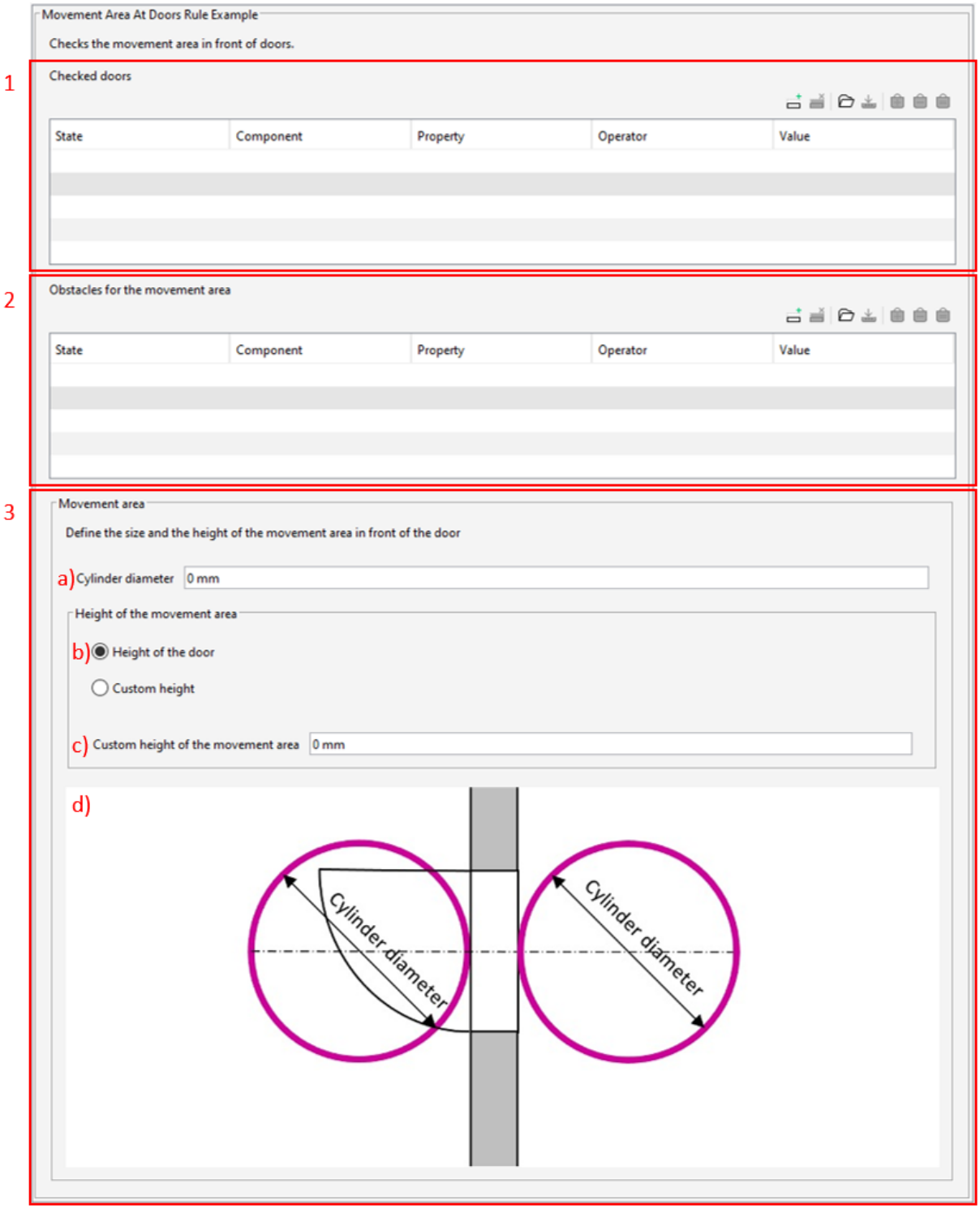

Abb. 9.1: UI der Beispielregel 2

```
8
9  import com.solibri.geometry.linearalgebra.MVector2d;
10 import com.solibri.geometry.linearalgebra.Vector2d;
11 import com.solibri.geometry.linearalgebra.Vector3d;
12 import com.solibri.geometry.mesh.TriangleMesh;
13 import com.solibri.geometry.primitive2d.Area;
14 import com.solibri.geometry.primitive2d.MPolygon2d;
15 import com.solibri.geometry.primitive2d.Polygon2d;
16 import com.solibri.geometry.primitive3d.AABB3d;
17 import com.solibri.geometry.primitive3d.Polygon3d;
18 import com.solibri.smc.api.SMC;
19 import com.solibri.smc.api.checking.DoubleParameter;
20 import com.solibri.smc.api.checking.EnumerationParameter;
21 import com.solibri.smc.api.checking.FilterParameter;
22 import com.solibri.smc.api.checking.OneByOneRule;
23 import com.solibri.smc.api.checking.PreCheckResult;
24 import com.solibri.smc.api.checking.Result;
25 import com.solibri.smc.api.checking.ResultCategory;
26 import com.solibri.smc.api.checking.ResultFactory;
27 import com.solibri.smc.api.checking.RuleParameters;
28 import com.solibri.smc.api.checking.RuleResources;
29 import com.solibri.smc.api.checking.Severity;
30 import com.solibri.smc.api.filter.AABBIntersectionFilter;
31 import com.solibri.smc.api.filter.ComponentFilter;
32 import com.solibri.smc.api.ifc.IfcEntityType;
33 import com.solibri.smc.api.model.PropertyType;
34 import com.solibri.smc.api.model.Quantities;
35 import com.solibri.smc.api.model.Relation;
36 import com.solibri.smc.api.model.Quantities.Type;
37 import com.solibri.smc.api.model.components.Covering;
38 import com.solibri.smc.api.model.components.Door;
39 import com.solibri.smc.api.model.Component;
40 import com.solibri.smc.api.ui.BorderType;
41 import com.solibri.smc.api.ui.UIComponent;
42 import com.solibri.smc.api.ui.UIContainer;
43 import com.solibri.smc.api.ui.UIContainerVertical;
44 import com.solibri.smc.api.ui.UIImage;
45 import com.solibri.smc.api.ui.UILabel;
46 import com.solibri.smc.api.ui.UIRadioButtonPanelVertical;
47 import com.solibri.smc.api.ui.UIRuleParameter;
48 import com.solibri.smc.api.visualization.ARGBColor;
49 import com.solibri.smc.api.visualization.Mesh;
50
51 public final class TurningCircleAtDoors extends OneByOneRule {
52
53    // Constant names - suggested by Solibri
54    // Filterparamter
55    private static final String COMPONENT_FILTER_OBSTACLES_ID = "
          rpComponentFilterObstacles";
56    // DoubleParameter
57    private static final String CIRCLE_DIAMETER_PARAMETER_ID = "
          rpCircleDiameter";
58    private static final String OBJECT_HEIGHT_PARAMETER_ID = "rpObjectHeight
          ";
59    // EnumerationParameter
60    private static final String CHOOSE_OBJECT_HEIGHT_ID = "
          rpChooseObjectHeight";
61    private static final String CHOOSE_DOOR_HEIGHT_ID = "rpChooseDoorHeight"
          ;
62    private static final String CHOOSE_CUSTOM_HEIGHT_ID = "
          rpChooseCustomHeight";
63
64    // Create the parameter creation handler for the rule
```

```
65      private final RuleParameters params = RuleParameters.of(this);


68      // FilterParameter
69      final FilterParameter rpComponentFilter = this.getDefaultFilterParameter
            ();
70      final FilterParameter rpComponentFilterObstacles = params.createFilter(
            COMPONENT_FILTER_OBSTACLES_ID);
71      // DoubleParameter
72      final DoubleParameter rpCircleDiameter = params.createDouble(
            CIRCLE_DIAMETER_PARAMETER_ID, PropertyType.LENGTH);
73      final DoubleParameter rpObjectHeight = params.createDouble(
            OBJECT_HEIGHT_PARAMETER_ID, PropertyType.LENGTH);
74      // EnumerationParameter
75      private final EnumerationParameter rpEnumerationObjectHeight = params.
            createEnumeration(CHOOSE_OBJECT_HEIGHT_ID,
76      Arrays.asList(CHOOSE_DOOR_HEIGHT_ID, CHOOSE_CUSTOM_HEIGHT_ID));
77
78      private final RuleResources resources = RuleResources.of(this);
```

Bevor mit der Überprüfung begonnen werden kann, muss noch eine globale Variable definiert werden, um später eine Toleranz bei Erstellung des Zylinders zur Verfügung zu haben.

```
80  private static final double toleranceGap = 0.01;
```

Bevor die check-Methode mit dem Durchlauf über alle Türen beginnt, wird ein *preCheck* ausgeführt. Ziel dieses ist die Prüfung der Nutzereingaben (Kreisdurchmesser). Sind diese nicht zulässig, ist die Durchführung der check-Methode nicht erforderlich. Die Prüfung erfolgt mit einer if-Verzweigung. Ist die Bedingung erfüllt (Kreisdurchmesser ist nicht zulässig), wird ein Ergebnis zurückgegeben (Zeile 87), welches dem Anwender den Fehler beschreibt und die check-Methode für überflüssig erklärt. Ist die Bedingung der Verzweigung nicht erfüllt, also ein erlaubter Wert für den Durchmesser eingegeben, wird ebenso ein Ergebnis zurückgegeben (Zeile 89). Mit diesem gilt der *preCheck* dann als bestanden und es wird die eigentliche check-Methode gestartet.

```
82  @Override
83  public PreCheckResult preCheck() {
84
85      // check user input
86      if (rpCircleDiameter.getValue() <= 0) {
87          return PreCheckResult.createIrrelevant(resources.getString("RE.
                WrongDimensionValues.NAME"));
88      }
89      return PreCheckResult.createRelevant();
90  }
91  \textit{}
```

Der Aufruf der check-Methode läuft genauso ab wie bei der ersten Beispielregel. Danach wird wieder eine *Collection* für die Ergebnisse erstellt. Im nächsten Schritt werden zwei unterschiedliche Ergebniskategorien mit den Namen und den Erklärungen aus den Ressourcen definiert. Diese Kategorien fassen weiterer Folge die jeweiligen Ergebnisse zusammen. Ergebniskategorien dienen zum besseren Überblick über die Fehler und werden in Solibri als eine Art Ordner, welche die Ergebnisse enthalten, dargestellt. Die erste Kategorie `componentResult` dient für die klassischen Ergebnisse in Solibri. Die zweite Kategorie `wrongInput` fasst Fehler in den Benutzereingaben zusammen.

```
93  @Override
94  public Collection<Result> check(Component component, ResultFactory
```

```
        resultFactory) {
95      // Creating a collection of results that will be returned at the end
96      // of the check method
97      Collection<Result> results = new ArrayList<>();
98      // ResultCategories
99      ResultCategory componentResult = resultFactory.createCategory(
100     component.getName() + resources.getString("RC.ClashCategory.NAME"),
101     component.getName() + resources.getString("RC.ClashCategory.NAME"));
102     ResultCategory wrongInput = resultFactory.createCategory(resources.
            getString("RC.WrongInput.NAME"),
103     resources.getString("RC.WrongInput.NAME"));
```

Die erste Überprüfung ist eine if-Verzweigung, ob die im ersten Komponentenfilter eingegebene Komponente eine Instanz des Interfaces Tür ist. Ist dies nicht der Fall, wird ein Ergebnis erstellt, das den Namen der falschen Komponente hat und eine Beschreibung aus den Ressourcen besitzt. Jenes wird dann der Ergebniskategorie `wrongInput` zugeordnet. Schließlich wird das Ergebnis als einziges Objekt in einer `Collection` zurückgegeben und mit der nächsten eingegebenen Komponente in der check-Methode fortgefahren. Ist die übergebene Komponente jedoch eine Tür, wird durch eine Typumwandlung (*typecasting*) aus der allgemeinen Komponente eine Tür mit dem Namen *door* gemacht. Der Grund für die Umwandlung ist, dass die Klasse `Door` die allgemeine Klasse `Component` um benötigte Tür-spezifische Methoden erweitert (z. B. Abrufen der Öffnungsrichtung).

```
105 // result if the given component is not from type door
106 if (!(component instanceof Door)) {
107     Result result = resultFactory
108     .create(component.getName(), resources.getString("RE.WrongComponentType.
            DESCRIPTION"))
109     .withCategory(wrongInput);
110     return Collections.singleton(result);
111 }
112 Door door = (Door) component;
```

Ausgehend von der vorher ermittelten Tür werden nun die zugehörigen Öffnungen bestimmt. Dies geschieht durch Ermittlung der zur Tür in Beziehung stehen Komponenten. Die Beziehung selbst soll den Typ einer Füllung haben und in beide Richtungen gehen, also von der vorgegebenen Komponente zur gesuchten und umgekehrt. So können die Öffnungen gefunden werden, wovon die betrachtete Tür die Füllung ist. Sollte es aus einem Grund keine passende Öffnung geben, wird ein Ergebnis der Kategorie *wrongInput* erstellt, wodurch die check-Methode für die aktuelle Tür beendet ist. Üblicherweise sollte in der Collection genau eine Öffnung vorhanden sein, weshalb durch eine einzelne Iteration auf der Collection das erste enthaltene Objekt, also die erste Komponente, ermittelt werden kann (Zeile 125).

```
114 // retrieve the related openings
115 Collection<Component> relatedFillings = component
116 .getRelated(Relation.of(Relation.Type.FILLS, Relation.Direction.BOTH));
117 // result if no filling was found
118 if (relatedFillings.isEmpty()) {
119     Result result = resultFactory
120     .create(component.getName(), resources.getString("RE.DoorWithoutWall.
            DESCRIPTION"))
121     .withCategory(wrongInput);
122     return Collections.singleton(result);
123 }
124 // getting the first and only opening out of the Collection
125 Component doorOpening = relatedFillings.iterator().next();
```

Im nächsten Abschnitt finden einige Schritte statt, die für die spätere Kontrolle notwendig sind. Zuerst wir der Fußabdruck der Öffnung als Fläche abgespeichert und von dieser dann die Größe bestimmt. Nach Bestimmung der Breite der Öffnung kann durch einfache Division die Dicke der Öffnung bestimmt werden. Danach wird noch die Öffnungsrichtung der Tür als Vektor bestimmt und in den zweidimensionalen Raum übergeführt. Schließlich werden ausgehend von der Öffnung die auf beiden Seiten liegenden nächsten Räume bestimmt.

```
127 // Area of the Opening
128 Area openingArea = doorOpening.getFootprint().getArea();
129 // Centroid of the openingArea
130 MVector2d openingAreaCentroid = openingArea.getCentroid();
131 // Getting the thickness of the opening
132 // First get the width from the quantities and then divide the area by the
        width
133 Double openingWidth = Quantities.of(doorOpening).get(Type.WIDTH).get();
134 Double openingAreaSize = openingArea.getSize();
135 Double openingThickness = openingAreaSize / openingWidth;
136
137 // Openingdirection of the door (elevator doors also have a
138 // referenceDirection perpendicular to the door)
139 Vector3d doorOpeningDirection = door.getReferenceDirection();
140 MVector2d doorOpeningDirection2d = doorOpeningDirection.to2dVector();
141
142 // check the movement area in all nearest spaces of the current component
143 Collection<Component> nearestSpaces = component
144 .getRelated(Relation.of(Relation.Type.NEAREST_SPACES, Relation.Direction.
        FORWARD));
```

Alle nun folgenden Prüfungen und Methoden werden jeweils für jeden der beiden Räume bzw. die beiden Seiten einer Tür extra durchgeführt. Dies erfolgt über die for-Schleife, welche zwei Durchläufe hat. Im ersten Durchlauf wird die ursprünglich ermittelte Öffnungsrichtung verwendet, im zweiten Durchlauf wird der Vektor mit der Methode `rotateVector` um π gedreht und zeigt dann in die entgegengesetzte Richtung. Zuerst wird der Mittelpunkt des Wendekreises bestimmt. Durch einfache Vektoraddition werden zur vorher bestimmten Mitte der Öffnung deren halbe Dicke sowie der halbe eingegebene Durchmesser hinzuaddiert. Im nächsten Schritt wird der passende Raum zum berechneten Mittelpunkt bestimmt. Hierfür wird von den Räumen der Fußabdruck, davon die Fläche und von dieser die Polygone ermittelt. Liegt der Mittelpunkt nun in einem der Polygone, ist es der richtige Raum.

```
147 // check the movement area at both sides of the door
148 for (int i = 0; i < 2; i++) {
149     Vector2d currentDirection = rotateVector(doorOpeningDirection2d, i *
            Math.PI);
150
151     // Center of the movementArea
152     Vector2d areaCenter = null;
153
154     areaCenter = openingAreaCentroid
155     .add(currentDirection.scale((openingThickness / 2 + rpCircleDiameter.
            getValue() / 2)));
156
157     // find the space at the current side of the door by checking whether
158     // the center is inside the space
159     Component relatedSpace = null;
160     for (Component nearestSpace : nearestSpaces) {
161         // use all polygons of the space, because a space can be build from
162         // two separate polygons
163         List<MPolygon2d> spacePolygons = nearestSpace.getFootprint().getArea
```

```
            ().getPolygons();
164         for (MPolygon2d spacePolygon : spacePolygons) {
165            if (spacePolygon.contains(areaCenter)) {
166               relatedSpace = nearestSpace;
167            }
168         }
169      }
```

Als nächstes muss die Höhe (z-Koordinate) der Grundfläche des Zylinders bestimmt werden. Als Ausgangswert wird die Höhe der Türunterkante genommen. Im Regelfall ist jedoch ein Fußbodenaufbau vorhanden. Trifft dies zu, wird durch die ausgelagerte Methode `defineFlooringElevation` die Fußbodenoberkante bestimmt. Hierzu müssen der aktuelle Raum, die Mitte des Wendekreises und die Info `top` übergeben werden. Diese gibt an, dass die Oberkante des Fußbodens zurückgegeben werden soll.

```
171   // define the elevation of the movementArea
172   // if no space is available in front of the door, the bottom elevation of
173   // the door is chosen
174   double areaElevation = door.getGlobalBottomElevation();
175   // otherwise the floor elevation of the space is determined
176   if (relatedSpace != null) {
177      areaElevation = defineFlooringElevation(relatedSpace, areaCenter, "top")
          ;
178   }
```

Vor der Durchführung der eigentlichen Kollisionsprüfung, muss das benötigte `TriangleMesh` erstellt werden. Hierzu wird zuerst ein `TriangleMesh` initialisiert und die Höhe der Deckfläche ausgehend von der Höhenlage der Grundfläche ermittelt. Je nach Auswahl im UI wird die Höhe gleich der Oberkante der Tür gesetzt oder ausgehenden von der Fußbodenhöhe mit dem eingegebenen Wert berechnet.

Mit diesen Werten kann nun mit der ausgelagerten Methode `createCylinderTriangleMesh` das `TriangleMesh` erzeugt werden. Die übergebenen Parameter sind der Mittelpunkt des Kreises, der gewünschten Durchmesser, die Zahl 72 sowie die Höhen der Grund- und Deckfläche. Der Durchmesser muss um die zu Beginn der Regel gewählte Toleranz abgemindert werden, um falsche Kollisionen zu verhindern. Der Wert 72 legt fest, dass der Kreis durch ein Polygon mit 72 Ecken zu approximieren ist.

```
180   // Create a TriangleMesh for the shape of the movementArea
181   TriangleMesh movementArea = null;
182   double boundingBoxDiameter = 0;
183   // topElevation dependent on user input
184   double topElevation = 0;
185   if (rpEnumerationObjectHeight.getValue().equals(CHOOSE_DOOR_HEIGHT_ID)) {
186      topElevation = component.getGlobalTopElevation();
187   } else {
188      topElevation = areaElevation + rpObjectHeight.getValue();
189   }
190   // Creating the vertices of the circle with an interval of 5 degrees
191   // the diameter is decreased by the toleranceGap to avoid clashes when the
192   // required distance is given exactly
193   movementArea = createCylinderTriangleMesh(areaCenter, rpCircleDiameter.
          getValue() - toleranceGap, 72,
194   areaElevation, topElevation);
195   boundingBoxDiameter = 2 * rpCircleDiameter.getValue();
```

Um nicht rechenaufwendig für alle vom User eingegebenen Komponenten einen *ClashCheck* durchführen zu müssen, werden Komponenten herausgefiltert, die sich weit von der aktuellen

Komponente entfernt befinden. Für eine solche Vorprüfung eignen sich *BoundingBoxes*. Diese Quader umhüllen die Geometrie des betrachteten Elements und können aufgrund ihrer vereinfachten Geometrie einfacher auf Kollision überprüft werden. Die Solibri API gibt dafür die Klasse *AABB3d* (Axis-aligned BoundingBox 3d) vor. Diese *BoundingBoxes* sind zusätzlich zur vereinfachten Geometrie nach den globalen Koordinatenachsen ausgerichtet. Die Klasse ermöglicht neben der Erstellung einer *BoundingBox* ausgehend von einer Komponente auch eine benutzerdefinierte Erstellung über die Angabe der Eckpunkte (erfolgt im Code mithilfe der ausgelagerten Methode `createBoundingBox`). Die hier zurückbekommene *BoundingBox* ist aufgrund des vorhin gewählten und übergebenen Durchmessers doppelt so groß wie der Zylinder. Für die Überprüfung wird ein neuer Filter erstellt, der die Komponenten enthält, welche die *BoundingBox* schneiden und im zweiten Komponentenfilter eingegeben wurden. Abschließend werden die Komponenten vom Modell abgerufen und in der Kollektion für die Hindernisse gespeichert.

```
197 // Getting the components of the target filter
198 // Creating a BoundingBox in front of the Door for a precheck
199 AABB3d customBoundingBox = createBoundingBox(areaCenter, areaElevation,
        component.getGlobalTopElevation(),
200 boundingBoxDiameter);
201 // Creating a filter that accepts the components of the filter which
202 // additionally intersect with the customBoundingBox
203 ComponentFilter targetComponentFilter = AABBIntersectionFilter.of(
        customBoundingBox)
204 .and(rpComponentFilterObstacles.getValue());
205 // Storing the components of the targetComponentFilter in the collection
        targets
206 Collection<Component> targets = SMC.getModel().getComponents(
        targetComponentFilter);
```

Schließlich wird nun der eigentliche Check innerhalb einer Schleife über die vorhin festgelegten Hindernisse durchgeführt. Durch eine if-Verzweigung werden noch jene Hindernisse herausgefiltert, die den Typ `IfcBuildingElementPart` haben. Jede Komponente ist aus mehreren `IfcBuildingElementPart` zusammengefügt. Da bereits die Komponenten als Ganzes geprüft werden, ist eine Berücksichtigung ihrer Einzelteile nicht erforderlich. Die Ergebnisse der ausgelagerten Methode `clashCheck` sind in einer *Collection* gespeichert und können deshalb direkt zu der Result-Collection in der check-Methode hinzugefügt werden. Als Übergabeparameter bekommt die Methode die eingegebene Tür (als allgemeine Komponente), das Hindernis, die Höhe der Grundfläche, das `TriangleMesh`, sowie die zu Beginn der Regel erstellte `resultFactory` und die Ergebniskategorie `componentResult`. Die Rückgabe der *Collection* markiert das Ende der check-Methode. Falls weitere Türen vorhanden sind, wird die Methode für die nächste Tür neu gestartet.

```
208         // running the clashcheck-method for all target components
209         for (Component target : targets) {
210             // exclude BuildingElementParts, because they cause random
                    StackOverflowErrors
211             if (!target.getIfcEntityType().get().equals(IfcEntityType.
                    IfcBuildingElementPart)) {
212                 results.addAll(
213                 clashCheck(component, target, areaElevation, movementArea,
                        resultFactory, componentResult));
214             }
215         }
216     }
217     return results;
218 } // End of the check-method (main-method)
```

Nach der check-Methode folgen im Programmcode ausgelagerte Methoden. Die kurze Methode `rotateVector` dient zur Drehung eines zweidimensionalen Vektors `inputVector` um einen Winkel `angle` in der Einheit Radiant. Dies erfolgt durch die Bestimmung der neuen x- und y-Koordinaten ausgehend von den ursprünglichen Koordinaten mittels mathematischer Formeln, welche der gängigen Literatur entnommen werden können. Schließlich wird der neue zweidimensionale Vektor `vector` erstellt und zurückgegeben.

```
223 private MVector2d rotateVector(MVector2d inputVector, double angle) {
224
225     double newX = inputVector.getX() * Math.cos(angle) - inputVector.getY()
            * Math.sin(angle);
226     double newY = inputVector.getX() * Math.sin(angle) + inputVector.getY()
            * Math.cos(angle);
227
228     MVector2d vector = MVector2d.create(newX, newY);
229
230     return vector;
231 }
```

Die Methode `defineFlooringElevation` dient im Allgemeinen zur Bestimmung der Oberkante oder Unterkante eines Fußbodenaufbaues, je nach übergebenem String. Neben dem String bekommt die Regel noch den zu betrachtenden Raum und einen darin befindlichen Punkt übergeben. Da es theoretisch auch Räume ohne Fußbodenaufbau geben kann, werden sowohl die Unterkante `bottomElevation` als auch die Oberkante `topElevation` initial mit der Unterkante des Raumes festgelegt. Im nächsten Schritt wird zuerst eine Beziehung erstellt, die den Typ `NEAREST_SPACES` hat und die Richtung `BACKWARD`. Damit werden vom Raum diejenigen Komponenten ermittelt, zu denen der Raum der nächstgelegene ist. Diese Komponenten werden dann in einer `Collection` gespeichert.

Darauffolgend läuft über all diese Komponenten eine for-Schleife. Die jeweilige Komponente wird jedoch nur näher betrachtet, wenn sie den Typ `Covering` besitzt. Um die speziellen Funktionen dieses Typs nutzen zu können, wird die allgemeine Komponente darauffolgend in eine `Covering` gecastet. Schließlich wird noch der `Covering.Type` bestimmt, der als String ausgewertet und gespeichert wird. Wenn der gespeicherte Typ `FLOORING` ist, handelt es sich um einen Fußboden und die Methode wird fortgesetzt. Es folgt die Ermittlung des Fußabdrucks und Kontrolle, ob der übergebene Punkt innerhalb des Fußabdruckes ist und die Oberseite des `Covering` nicht höher als der höchste Punkt des Raums liegt. Beides dient zur erneuten Kontrolle, ob es sich um den richtigen Fußboden handelt (für den Fall, dass ein Raum verschiedene Fußbodenaufbauten aufweist). Sind die Kontrollen erfüllt, werden die beiden z-Koordinaten neu festgelegt. Je nachdem, ob durch die Übergabe die obere oder untere Koordinate verlangt wurde, wird die passende zurückgegeben.

```
238 private double defineFlooringElevation(Component spaceComponent, Vector2d
        point, String output) {
239     // set elevation of the space as default if no flooring is available
240     double topElevation = spaceComponent.getGlobalBottomElevation();
241     double bottomElevation = spaceComponent.getGlobalBottomElevation();
242     // Get all components that are connected with the space with the
243     // relation Nearest_Spaces
244     Relation nearestSpacesRelation = Relation.of(Relation.Type.
            NEAREST_SPACES, Relation.Direction.BACKWARD);
245     Collection<Component> nearestComponentsCollection = spaceComponent.
            getRelated(nearestSpacesRelation);
246
247     // for all coverings
248     for (Component boundaryComponent : nearestComponentsCollection) {
249         if (boundaryComponent instanceof Covering) {
```

```
250             // cast the component into a covering
251             Covering targetCovering = (Covering) boundaryComponent;
252             // get the type of the covering
253             Covering.Type typeOfCovering = targetCovering.getType();
254             String typeOfCoveringString = String.valueOf(typeOfCovering);
255
256             // check whether the covering is from type "FLOORING"
257             if (typeOfCoveringString.equals("FLOORING")) {
258                 // if so, get it's footprint
259                 Polygon2d coveringFootprint = targetCovering.getFootprint().
                        getArea().getLargestPolygon();
260                 // check whether the point is inside the polygon and whether
261                 // the flooring is not above the space
262                 if (coveringFootprint.contains(point)
263                 && targetCovering.getGlobalTopElevation() < spaceComponent.
                        getGlobalTopElevation()) {
264                     // if so, get the covering's top elevation as elevation of
                            the node
265                     topElevation = targetCovering.getGlobalTopElevation();
266                     bottomElevation = targetCovering.getGlobalBottomElevation();
267                 }
268             }
269         }
270     }
271     // return either the top elevation or the bottom elevation
272     if (output.equals("top")) {
273         return topElevation;
274     } else {
275         return bottomElevation;
276     }
277 }
```

Nachdem der Wendekreis dreidimensional in Form eines Zylinders kontrolliert werden soll, wird dieser Zylinder mit der Methode `createCylinderTriangleMesh` als `TriangleMesh` erstellt. Dieses Konstrukt wurde bereits in Kapitel 8 erklärt. Als Übergabeparameter bekommt die Methode den Mittelpunkt und den Durchmesser des Wendekreises, die Präzision sowie die z-Koordinaten der Grund- und der Deckfläche. Zu Beginn der Methode wird die Umrandungen der Grund- und der Deckfläche mit der ab Zeile 331 definierten Methode `createCircle` erstellt und die Rückgaben in den beiden erstellten Listen gespeichert. Der Unterschied zwischen den beiden Methodenaufrufen liegt bei den verschiedenen z-Koordinaten.

Ein `TriangleMesh` wird durch seine Oberflächen definiert, die entweder als Dreiecke oder Polygone zu repräsentieren sind. In diesem Beispiel verwenden wir Polygone, zu deren Erstellung die Eckpunkte benötigt werden. Neben den Eckpunkten von Grund- und Deckfläche, sind zusätzlich die Eckpunkte je Mantelfläche erforderlich. Diese Eckpunkte werden je Polygon in eine Liste zusammengefasst in eine übergeordnete Liste für alle Oberflächen gespeichert. In den Zeilen 291-295 erfolgt die Erstellung der übergeordneten List und Speicherung der Listen für Grund- und Deckfläche. Die Erstellung der Eckpunkte der Mantelflächen erfolgt in einer for-Schleife mit so vielen Durchläufen, wie durch die Präzision festgelegt. Je Schleifendurchlauf wird eine Liste mit den Eckpunkten der Polygone `polygonVertices` erstellt (ausgehend von links unten im Uhrzeigersinn). Beim letzten Polygon muss darauf geachtet werden, dass es mit den Eckpunkten des ersten Polygons verbunden ist, damit sich die Mantelfläche schließt (siehe else-Block). Schließlich wird die Liste für das einzelne Polygon noch in die übergreifende Liste `polygonVerticeLists` hinzugefügt.

In einer neuen for-Schleife werden dann aus den Listen mit den Eckpunkten die einzelnen dreidimensionalen Polygone, durch Iteration über die übergreifende Liste `polygonVerticeLists`, erstellt und in einer zuvor definierten Liste `polygons` gespeichert. Nachdem die Polygone in der geforderten Reihenfolge erstellt worden sind, kann das `TrianlgeMesh` *mesh* erstellt und dann

zurückgegeben werden.

```
285 private TriangleMesh createCylinderTriangleMesh(Vector2d circleCenter,
        double circleDiameter,
286 double circlePrecision, double bottomElevation, double topElevation) {
287     // calculate the vertices of the bottom and top circle of the cylinder
288     List<Vector3d> bottomVertices = createCircle(circleCenter,
            circleDiameter, circlePrecision, bottomElevation);
289     List<Vector3d> topVertices = createCircle(circleCenter, circleDiameter,
            circlePrecision, topElevation);
290
291     List<List<Vector3d>> polygonVerticeLists = new ArrayList<>();
292
293     // add vertices of the bottom and the top polygon to the overall list
294     polygonVerticeLists.add(bottomVertices);
295     polygonVerticeLists.add(topVertices);
296
297     // add the vertices of the side polygons
298     for (int i = 0; i < circlePrecision; i++) {
299         List<Vector3d> polygonVertices = new ArrayList<>();
300         if (i != circlePrecision - 1) {
301             polygonVertices.add(bottomVertices.get(i));
302             polygonVertices.add(bottomVertices.get(i + 1));
303             polygonVertices.add(topVertices.get(i + 1));
304             polygonVertices.add(topVertices.get(i));
305         } else {
306             polygonVertices.add(bottomVertices.get(i));
307             polygonVertices.add(bottomVertices.get(0));
308             polygonVertices.add(topVertices.get(0));
309             polygonVertices.add(topVertices.get(i));
310         }
311         polygonVerticeLists.add(polygonVertices);
312     }
313
314     // create the polygons
315     List<Polygon3d> polygons = new ArrayList<>();
316     for (List<Vector3d> polygonVertices : polygonVerticeLists) {
317         polygons.add(Polygon3d.create(polygonVertices));
318     }
319
320     TriangleMesh mesh = TriangleMesh.fromPolygons(polygons);
321
322     return mesh;
323 }
```

Die bei der Erstellung des `TriangleMesh` benötigten Kreise für die Grund- und Deckfläche werden in folgender Methode erstellt. Als Übergabewerte werden der Mittelpunkt, der gewählte Durchmesser, die Präzision und die z-Koordinate benötigt. Genauer gesagt, wird kein gesamter Kreis sondern eine Liste von Punkten erstellt, die den Kreis als Polygon approximiert.

Der erste Schritt der Methode ist die Erstellung der benötigten Liste `vertices`. Darauffolgend werden in einer for-Schleife die einzelnen Punkte erstellt. Hierfür werden jeweils die x- und y-Koordinaten ausgehend von der jeweiligen Mittelpunktskoordinate (`circleCenter.getX()` bzw. `.getY()`) bestimmt. Zu diesen wird der Cosinus (`Math.cos()`) bzw. Sinus (`Math.sin()`) mal dem halben Durchmesser hinzuaddiert. Die Winkelfunktionen werden je Schleifendurchlauf durch ein Vielfaches von π bestimmt. Schließlich wird aus den beiden Koordinaten der gesuchte Punkt erzeugt, der Liste hinzugefügt und diese nach dem letzten Schleifendurchlauf zurückgegeben.

```
331 private List<Vector3d> createCircle(Vector2d circleCenter, double
        circleDiameter, double circlePrecision,
```

```
332 double zCoordinate) {
333
334     // List for the vertices (Eckpunkte)
335     List<Vector3d> vertices = new ArrayList<Vector3d>();
336
337     // Creating the vertices of the circle
338     for (int i = 0; i < circlePrecision; i++) {
339         double xCoordinate = circleCenter.getX()
340         + Math.cos(i * Math.PI / (circlePrecision / 2)) * (circleDiameter /
              2);
341         double yCoordinate = circleCenter.getY()
342         + Math.sin(i * Math.PI / (circlePrecision / 2)) * (circleDiameter /
              2);
343         Vector3d vertex = Vector3d.create(xCoordinate, yCoordinate,
              zCoordinate);
344         vertices.add(vertex);
345     }
346     return vertices;
347 }
```

Ziel der folgenden Methode ist die Erstellung einer rechteckigen *BoundingBox*, die den Zylinder umschreibt. Für die Erstellung benötigt die Methode den Mittelpunkt des Wendekreises sowie die z-Koordinaten der Deck- und Grundfläche. Zuerst wird eine Liste für die Eckpunkte angelegt. Diese werden mit der vorhin erklärten Methode `createCircle` erstellt. In diesem Fall wird jedoch als Präzision der Wert 4 übergeben, was zu einem Quadrat führt. Je nach übergebener Höhe werden die Punkte der Grund- oder Deckfläche erstellt. Die Rückgabe der Methode sind die einzelnen Punkte, die in die Liste `bbVertices` gespeichert werden und aus denen im letzten Schritt die *BoundingBox* erstellt wird.

```
356 private AABB3d createBoundingBox(Vector2d middlePoint, double
        bottomElevation, double topElevation,
357 double diameter) {
358
359     // List where the vertices are stored
360     List<Vector3d> bbVertices = new ArrayList<Vector3d>();
361
362     bbVertices.addAll(createCircle(middlePoint, diameter, 4.,
            bottomElevation));
363     bbVertices.addAll(createCircle(middlePoint, diameter, 4., topElevation))
            ;
364
365     // Creating a boundingBox with the given edges
366     AABB3d boundingBox = AABB3d.create(bbVertices);
367
368     return boundingBox;
369 }
```

Um schlussendlich ein Hindernis im Wendekreis feststellen zu können, muss ähnlich zur ersten Beispielregel noch eine Kollisionsprüfung durchgeführt werden. Die verwendete Methode unterscheidet sich jedoch etwas zur ersten Beispielregel, da in diesem Fall keine zwei Komponenten sondern eine Komponente und ein TriangleMesh überschnitten werden.

Die Methode bekommt die zu betrachtende Tür, eine Komponente als mögliches Hindernis, die z-Koordinate, das `TriangleMesh` des Zylinders sowie die `resultFactory` und eine `ResultCategory` übergeben. Wie in der check-Methode wird zuerst eine `Collection` für die Ergebnisse erstellt.

Bevor der eigentliche *ClashCheck* beginnt, erfolgt durch eine if-Verzweigung eine vorhergehende Höhenkontrolle. Es wird überprüft, ob die Unterseite der Komponente niedriger liegt als die Oberseite der Tür und umgekehrt, und somit eine Überschneidung überhaupt möglich wäre. Ist diese Bedingung erfüllt, wird aus der Komponente ein `TriangleMesh` gemacht, welches mit

dem vorhin erstellten und übergebenen `TriangleMesh` überschnitten wird. Die entstehende Überschneidung ist selbst wieder ein `TriangleMesh`. Um mögliche Fehler durch ein negatives Volumen (siehe Beispielregel 1) zu verhindern, muss eine Kontrolle auf positives Volumen eingebaut werden. Ist diese erfüllt, wird schließlich ein Ergebnis erstellt, der übergebenen Ergebniskategorie hinzugefügt und mit dem Schweregrad moderat versehen. Zusätzlich wird dem Ergebnis noch eine Visualisierung des Wendekreises sowie der *Intersection* hinzugefügt. Das genaue vorgehen der Erstellung einer `Visualization` wurde in Kapitel 8 bereits näher erklärt.

```
376 private Collection<Result> clashCheck(Component door, Component
        targetComponent, double floorElevation,
377 TriangleMesh movementAreaMesh, ResultFactory resultFactory, ResultCategory
        resultCategory) {
378     Collection<Result> results = new ArrayList<>();
379
380     // if-condition for the height of the target & the door component
381     if ((targetComponent.getGlobalBottomElevation() < door.
            getGlobalTopElevation())
382     && (targetComponent.getGlobalTopElevation() > floorElevation)) {
383
384         // retrieve the TriangleMesh of the targetComponent
385         TriangleMesh targetMesh = targetComponent.getTriangleMesh();
386
387         // Intersection between the movementAreaMesh and the targetMesh and
388         // the targetArea
389         TriangleMesh intersectionMesh = movementAreaMesh.intersection(
                targetMesh);
390         Mesh visualMovementArea = Mesh.create(movementAreaMesh.difference(
                intersectionMesh).toTriangleCollection());
391         Mesh visualIntersection = Mesh.create(ARGBColor.create(255, 0, 0,
                255),
392         intersectionMesh.toTriangleCollection());
393
394         // if there is no intersection >> continue with the loop
395         // if there is a intersection (tolerance 1cm^3; without tolerance
396         // there are bugs) >> create a result
397         if (intersectionMesh.getVolume() > 0.000001) {
398             String name = door.getName() + resources.getString("RE.ClashResult
                    .NAME") + targetComponent.getName();
399             String description = door.getName() + resources.getString("RE.
                    ClashResult.NAME")
400             + targetComponent.getName();
401             // adds the component in the dropdown menu of the result
402             Result result = resultFactory.create(name, description).
                    withInvolvedComponent(targetComponent)
403             .withVisualization(visualization -> {
404                 visualization.addVisualizationItem(visualMovementArea);
405                 visualization.addVisualizationItem(visualIntersection);
406             }).withSeverity(Severity.MODERATE) // severity of the result
407             .withCategory(resultCategory); // assign result to ResultCategory;
408             results.add(result);
409         }
410     }
411     return results;
412 }
```

Nach der check-Methode wird das UI definiert, wobei auch hier die Voreinstellungen überschrieben werden müssen. Das UI der zweiten Beispielregel besteht im wesentlichen aus zwei Teilen. Einerseits aus den beiden Komponentenfilter, andererseits aus dem extra Container für die geometrischen Eingaben für den Wendekreis. Auch der Programmcode ist so aufgebaut.

Zuerst wird der `mainContainer` als vertikaler Container erstellt, der dann die beiden Kompo-

nentenfilter und die weiteren Container enthält. Danach werden die ersten *Ruleparameter* in Form der beiden Komponentenfilter zum `mainContainer` hinzugefügt. Im nächsten Schritt wird der innere Container (siehe 3. in Abb. 9.1) mit der Methode `movementAreaContainer` erstellt und dem `mainContainer` hinzugefügt. Anschließend wird wieder der `mainContainer` zurückgegeben und das UI ist somit erstellt.

```
416 @Override
417 public UIContainer getParametersUIDefinition() {
418     // Vertical container with title
419     UIContainer mainContainer = UIContainerVertical.create(resources.
            getString("UI.TurningCircleRule.TITLE"),
420     BorderType.LINE);
421     // Description vertical container
422     mainContainer.addComponent(UILabel.create(resources.getString("UI.
            TurningCircleRule.DESCRIPTION")));
423
424     // Door filter
425     mainContainer.addComponent(UIRuleParameter.create(rpComponentFilter));
426     // Obstacle filter
427     mainContainer.addComponent(UIRuleParameter.create(
            rpComponentFilterObstacles));
428     // self-defined container
429     mainContainer.addComponent(movementAreaContainer());
430
431     return mainContainer;
432 }
```

In der Methode `movementAreaContainer` wird der innere Container definiert. Zuerst wird ein vertikaler Container und unter dem Namen `uiContainer` abgespeichert. Zu diesem wird der `DoubleParameter` für den Kreisdurchmesser hinzugefügt. Danach wird nochmals ein neuer vertikaler Container erstellt, zu dem das vertikale *RadioButtonPanel* und der *DoubleParameter* für die Höhe hinzugefügt werden. Nach diesem Container wird noch ein erklärendes Bild angeordnet.

```
434 // Container for the movement area
435 private UIComponent movementAreaContainer() {
436     UIContainer uiContainer = UIContainerVertical.create(resources.getString
            ("UI.MovementAreaContainer.TITLE"),
437     BorderType.LINE);
438     uiContainer.addComponent(UILabel.create(resources.getString("UI.
            MovementAreaContainer.DESCRIPTION")));
439     uiContainer.addComponent(UIRuleParameter.create(rpCircleDiameter));
440
441     UIContainer uiContainer1 = UIContainerVertical.create(resources.
            getString("rpChooseObjectHeight.DESCRIPTION"),
442     BorderType.LINE);
443     uiContainer1.addComponent(UIRadioButtonPanelVertical.create(
            rpEnumerationObjectHeight));
444     uiContainer1.addComponent(UIRuleParameter.create(rpObjectHeight));
445     uiContainer.addComponent(uiContainer1);
446
447     uiContainer.addComponent(UIImage.create(resources.getImageUrl(resources.
            getString("UI.AreaOptionsImage"))));
448
449     return uiContainer;
450 }
```

9.3 Ressourcen

Die Properties-Datei der zweiten Beispielregel ist durch einfache optische Mittel in drei Teile aufgeteilt. Im ersten Teil der Datei werden allgemeine Metadaten angeführt (Zeilen 1-7). Diese sind wiederum im *Ruleset Manager* in Solibri sichtbar, wie bereits bei der ersten Beispielregel dargestellt. Der zweite Abschnitt befasst sich mit dem UI. Hier sind der Titel und die Beschreibung des Container und *Ruleparameter* angeführt. Für die *Ruleparameter* sind zum Teil Standardwerte festgelegt. Für das eingefügte Bild im UI wird der passende Pfad innerhalb der Ressourcen angegeben. Im letzten Abschnitt wird die Ergebnisausgabe definiert. Das sind Namen und/oder Beschreibungen für Ergebniskategorien und Ergebnisse selbst.

Program Code 9.1: Properties-Datei der Beispielregel 2

```
1   DEFAULT_NAME = Turning Circle At Doors Example Rule
2   DEFAULT_DESCRIPTION = Checks the turning circle in front of doors.
3   AUTHOR=ZDB TU Wien
4   AUTHOR_TAG=ZDB
5   UID=12345
6   VERSION=1.0
7   DATE=2023-02-02
8
9   UI**********************************************************************
10  UI.TurningCircleRule.TITLE = Movement Area At Doors Rule Example
11  UI.TurningCircleRule.DESCRIPTION = <html>Checks the movement area in
        front of doors.<html>
12
13  rpComponentFilter.NAME = Checked doors
14  rpComponentFilter.DESCRIPTION = Only works with a component of the type
        door
15
16  rpComponentFilterObstacles.NAME = Obstacles for the movement area
17  rpComponentFilterObstacles.DESCRIPTION = Choose components which
        obstruct the movement area
18
19  UI.MovementAreaContainer.TITLE = Movement area
20  UI.MovementAreaContainer.DESCRIPTION = <html>Define the size and the
        height of the movement area in front of the door<html>
21
22  rpCircleDiameter.NAME = Cylinder diameter
23  rpCircleDiameter.DESCRIPTION = Cylinder diameter
24  rpCircleDiameter.DEFAULT_VALUE = 0
25
26  rpChooseObjectHeight.NAME = Height of the movement area
27  rpChooseObjectHeight.DESCRIPTION = Height of the movement area
28  rpChooseDoorHeight = Height of the door
29  rpChooseCustomHeight = Custom height
30
31  rpObjectHeight.NAME = Custom height of the movement area
32  rpObjectHeight.DESCRIPTION = Custom height of the movement area
33  rpObjectHeight.DEFAULT_VALUE = 0
34
35  UI.AreaOptionsImage = images/MovementAreaAtDoorsRule-AreaOptions.JPG
36
37  Results*****************************************************************
38  RC.ClashCategory.NAME = : Movement area is obstructed
39  RC.WrongInput.NAME = Wrong Input
40
41  RE.ClashResult.NAME = : Movement area is obstructed by
42  RE.WrongComponentType.DESCRIPTION = The given component is not from type
        door!
```

```
43    RE.DoorWithoutWall.DESCRIPTION = The given door is not inside a wall.
          The rule only works for doors inside walls.
44    RE.WrongDimensionValues.NAME = Please enter positive values greater than
           0 for the dimensions of the movement area.
```

10 Anhang

Program Code 10.1: Code zur Erstellung der UI RuleParameters

```
//Constant IDs of parameters
private static final String COMPONENT_FILTER_PARAMETER_ID1 = "
    rpComponentFilter1";
private static final String COMPONENT_FILTER_PARAMETER_ID2 = "
    rpComponentFilter2";

private static final String PROPERTY_PARAMETER_ID = "rpPropertyParameter";

private static final String DOUBLE_PARAMATER_LENGTH_ID = "
    rpDoubleParamaterLength";
private static final String DOUBLE_PARAMETER_PERCENTAGE_ID = "
    rpDoubleParameterPercentage";

private static final String STRING_PARAMETER_ID = "rpStringParameter";

private static final String ENUMERATION_COMBOBOX_ID = "
    rpEnumerationParameterForComboBox";
private static final String ENUMERATION_COMBOBOX_OPTION1_ID = "rpCBOption1"
    ;
private static final String ENUMERATION_COMBOBOX_OPTION2_ID = "rpCBOption2"
    ;

private static final String ENUMERATION_RADIOBUTTON_ID = "
    rpEnumerationParameterForRadioButton";
private static final String EUNMERATION_RADIOBUTTON_OPTION1_ID = "
    rpRBoption1";
private static final String EUNMERATION_RADIOBUTTON_OPTION2_ID = "
    rpRBoption2";

private static final String BOOLEAN_PARAMATER_ID = "rpBooleanParamater";

// RuleParameters
// Retrieve the parameters creator of OneByOneRule
private final RuleParameters params = RuleParameters.of(this);

// FilterParameter
final FilterParameter rpComponentFilter = this.getDefaultFilterParameter();
final FilterParameter rpComponentFilter1 = params.createFilter(
    COMPONENT_FILTER_PARAMETER_ID1);
final FilterParameter rpComponentFilter2 = params.createFilter(
    COMPONENT_FILTER_PARAMETER_ID2);

// PropertyReferenceParameter
final PropertyReferenceParameter rpPropertyParameter = params.
    createPropertyReference(PROPERTY_PARAMETER_ID);

// DoubleParameter
final DoubleParameter rpDoubleParamaterLength = params.createDouble(
    DOUBLE_PARAMATER_LENGTH_ID, PropertyType.LENGTH);
final DoubleParameter rpDoubleParameterPercentage = params.createDouble(
    DOUBLE_PARAMETER_PERCENTAGE_ID, PropertyType.PERCENTAGE);
```

```
// StringParameter
final StringParameter rpStringParameter = params.createString(
    STRING_PARAMETER_ID);

// EnumerationParamater
final EnumerationParameter rpEnumerationParameterForComboBox = params.
    createEnumeration(ENUMERATION_COMBOBOX_ID,
Arrays.asList(ENUMERATION_COMBOBOX_OPTION1_ID,
    ENUMERATION_COMBOBOX_OPTION2_ID));

final EnumerationParameter rpEnumerationParameterForRadioButtons = params.
    createEnumeration(ENUMERATION_RADIOBUTTON_ID,
Arrays.asList(EUNMERATION_RADIOBUTTON_OPTION1_ID,
    EUNMERATION_RADIOBUTTON_OPTION2_ID));

// BooleanParameter
final BooleanParameter rpBooleanParamater = params.createBoolean(
    BOOLEAN_PARAMATER_ID);

// Zugriff auf die Ressourcen
private final RuleResources resources = RuleResources.of(this);


// UI
@Override
public UIContainer getParametersUIDefinition() {

UIContainer mainContainer = UIContainerVertical.create("mainContainer");

mainContainer.addComponent(UILabel.create(resources.getString("UI.
    mainContainer.Description")));

// Include ComponentFilter
mainContainer.addComponent(UIRuleParameter.create(rpComponentFilter));

// Include RuleParameters and RadioButtonPanels to mainContainer
mainContainer.addComponent(RuleParameters());
mainContainer.addComponent(RadioButtonPanels());

return mainContainer;
}

private UIContainer RuleParameters() {

UIContainer uiContainerVertical = UIContainerVertical.create(resources.
    getString("UI.RuleParameters.TITLE"));

// Include ComponentFilters
uiContainerVertical.addComponent(UIRuleParameter.create(rpComponentFilter1)
    );
uiContainerVertical.addComponent(UIRuleParameter.create(rpComponentFilter2)
    );

// Include RuleParameters
uiContainerVertical.addComponent(UIRuleParameter.create(
    rpDoubleParamaterLength));
uiContainerVertical.addComponent(UIRuleParameter.create(
    rpDoubleParameterPercentage));
uiContainerVertical.addComponent(UIRuleParameter.create(rpStringParameter))
    ;
uiContainerVertical.addComponent(UIRuleParameter.create(rpPropertyParameter
    ));
```

```
uiContainerVertical.addComponent(UIRuleParameter.create(rpBooleanParamater)
    );
uiContainerVertical.addComponent(UIRuleParameter.create(
    rpEnumerationParameterForComboBox));

return uiContainerVertical;
}

private UIContainer RadioButtonPanels() {

UIContainer uiContainerHorizontal = UIContainerHorizontal.create(resources.
    getString("UI.RadioButtonPanels"));

UIContainer uiContainerVertical1 = UIContainerVertical.create(resources.
    getString("UI.RadioButton_vertical"));
UIContainer uiContainerVertical2 = UIContainerVertical.create(resources.
    getString("UI.RadioButton_horizontal"));

// Include RadioButtonPanels
UIRadioButtonPanel radioButtonPanelVertical = UIRadioButtonPanelVertical.
    create(rpEnumerationParameterForRadioButtons);
UIRadioButtonPanel radioButtonPanelHorizontal =
    UIRadioButtonPanelHorizontal.create(
    rpEnumerationParameterForRadioButtons);

// Include images to RadioButtonPanels
List<UIImage> radioButtonImages = new ArrayList<>();
radioButtonImages.add(UIImage.create(resources.getImageUrl(resources.
    getString("UI.Image_RB_1"))));
radioButtonImages.add(UIImage.create(resources.getImageUrl(resources.
    getString("UI.Image_RB_2"))));

radioButtonPanelVertical.addOptionImages(radioButtonImages);
radioButtonPanelHorizontal.addOptionImages(radioButtonImages);


uiContainerVertical1.addComponent(radioButtonPanelVertical);
uiContainerVertical2.addComponent(radioButtonPanelHorizontal);

uiContainerHorizontal.addComponent(uiContainerVertical1);
uiContainerHorizontal.addComponent(uiContainerVertical2);

return uiContainerHorizontal;
}
}
```

Program Code 10.2: Propterties Datei der RuleParameters

```
UI.mainContainer.Description = Description of the Rule

rpRBoption1 = Option 1
rpRBoption2 = Option 2

UI.RadioButton_vertical = Vertical RadioButtonPanel
UI.RadioButton_horizontal = Horizontal RadioButtonPanel

UI.Image_RB_1 = images/UI_RB_1.png
UI.Image_RB_2 = images/UI_RB_2.png

UI.RuleParameters.TITLE = RuleParameters
```

```
rpComponentFilter.NAME = ComponentFilter
rpComponentFilter.DESCRIPTION = This filter specifies the set of components
     to check.

rpComponentFilter1.NAME = ComponentFilter1
rpComponentFilter1.DESCRIPTION = This filter specifies a set of components.


rpComponentFilter2.NAME = ComponentFilter2
rpComponentFilter2.DESCRIPTION = This filter specifies another set of
    components.

UI.RadioButtonPanels = RadioButtonPanels

rpCBOption1 = Option 1
rpCBOption2 = Option 2

rpDoubleParamaterLength.NAME = Length
rpDoubleParamaterLength.DESCRIPTION = Enter the length
rpDoubleParamaterLength.DEFAULT_VALUE = 1.20 m

rpDoubleParameterPercentage.NAME = Percentage
rpDoubleParameterPercentage.DESCRIPTION = Enter the percentage
rpDoubleParameterPercentage.DEFAULT_VALUE = 0.40

rpStringParameter.NAME = String
rpStringParameter.DESCRIPTION = Enter the string
rpStringParameter.DEFAULT_VALUE = text

rpPropertyParameter.NAME = PropertyParameter
rpPropertyParameter.DESCRIPTION = Enter the PropertyParameter

rpBooleanParamater.NAME = Boolean
rpBooleanParamater.DESCRIPTION = Enter the Boolean
rpBooleanParamater.DEFAULT_VALUE = true

rpEnumerationParameterForComboBox.NAME = ComboBox
rpEnumerationParameterForComboBox = Enter the ComboBox
```