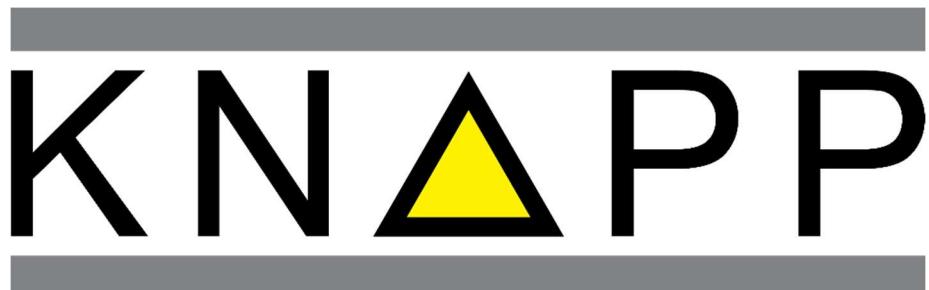


Höhere Technische Bundes-Lehr- und Versuchsanstalt Villach
Abteilung für Informatik

Diplomarbeit

KSB-Workflow-Generator B



Eingereicht am 04.04.2017 von
Maximilian Haider 5BHIFS
Lukas Kramer 5BHIFS
Martin Laubreiter 5BHIFS

Betreut und beurteilt von
Prof. Mag. Gerald Ortner

I. Eidesstattliche Erklärung

Wir versichern, dass wir diese Arbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt haben. Die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen haben wir alle gekennzeichnet und im Literaturverzeichnis angeführt.

Max Haider

Maximilian Haider, geb. am 04.07.1998

Lukas Kramer

Lukas Kramer, geb. am 17.04.1998

Martin Laubreiter

Martin Laubreiter, geb. am 28.08.1997

Villach am 04.04.2017

II. Danksagung

Wir bedanken uns bei unserem Betreuer Prof. Mag. Gerald Ortner für die ausgezeichnete Betreuung im Laufe der Diplomarbeit. Durch gezielte Verbesserungsvorschläge im Entstehungsprozess dieser Arbeit ist er maßgeblich an der hohen Qualität beteiligt. Ein besonderer Dank gilt unseren Familien, insbesondere unseren Eltern, die uns den Aufenthalt in Graz ermöglichten.

Des Weiteren danken wir Alexander Hödl, Gernot Wöber und Thomas Prem für die Unterstützung bei dieser Arbeit.

Ebenso gilt unser Dank Otto Uriach für das Korrekturlesen dieser Diplomarbeit.

III. Abstract

This thesis describes the development of the *KS-B-Workflow-Generator*.

At the Knapp AG Excel-Sheets are used to describe data of structures of warehouses. As this data is needed in XML-Format, those Excel-Sheets have always been converted from the Excel-Sheet to XML by hand. This used up a lot of time.

The *KS-B-Workflow-Generator* is an application developed to simplify this process by converting the Excel-Sheets to XML automatically.

Furthermore the program also allows to compare two files of different formats with one another and show the differences.

The planned result was a usable application. Additionally, an extended documentation was requested in case employees of Knapp AG have to extend the application.

Thanks to the automation, the process takes up less time and less prone to failure.

All of the components were programmed using Java and the JavaFX Framework. Moreover various APIs like Apache POI were used.

IV. Kurzfassung

In der Knapp AG wird das Microsoft Office Programm Excel dazu verwendet die Daten von Lagerhallen und deren Abläufen zu speichern. Um das Lagerhaus zu simulieren werden diese Daten jedoch in XML Format, benötigt.

Bisher mussten diese Daten immer händisch von einem Format in das andere gebracht werden, was einen enormen Zeitaufwand bedeutete.

Die Aufgabenstellung der Diplomarbeit war es, eine Java-Anwendung zu entwickeln, welche den Schritt der Konvertierung automatisiert.

Die Anwendung soll es neben der Hauptfunktion des Konvertierens zusätzlich ermöglichen, zwei Dateien unterschiedlichen Formates inhaltlich miteinander zu vergleichen und die Unterschiede aufzuzeigen.

Das geplante Ergebnis ist der produktive Einsatz der Anwendung. Des Weiteren war eine umfangreiche Dokumentation ein wichtiger Teil der Arbeit, damit die Anwendung in Zukunft problemlos von Mitarbeitern der Knapp AG erweitert werden kann. Für die Mitarbeiter, welche früher die Dateien händisch konvertieren mussten, bedeutete unsere Anwendung eine große Zeitersparnis, ebenso ist die Anwendung im Vergleich zur herkömmlichen Variante nicht so stark fehleranfällig.

V. Inhaltsverzeichnis

I.	Eidesstattliche Erklärung.....	2
II.	Danksagung	3
III.	Abstract	4
IV.	Kurzfassung	5
V.	Inhaltsverzeichnis.....	6
1.	Einleitung und Überblick.....	12
1.1.	Problemstellung	12
1.2.	Aufgabenstellung	13
1.3.	Vorgaben	14
1.4.	Kapitelübersicht	15
2.	Software–Architektur.....	17
2.1.	WarehouseModel.....	17
2.2.	Ladevorgang einer Excel-Datei.....	19
2.2.1.	ExcelReader	20
2.2.2.	XML-Generator	21
2.3.	Comparator	22
3.	Prozessmodell Scrum	23
3.1.	Kennzeichen von Agiler Softwareentwicklung.....	23
3.2.	Allgemein.....	23
3.3.	Sprint	24
3.4.	Rollen.....	25
3.4.1.	Product Owner.....	25
3.4.2.	Scrum Master	25
3.4.3.	Scrum Team	25
3.5.	Meetings.....	26
3.5.1.	Sprint Planning Meeting	26
3.5.2.	Daily Scrum Meeting.....	26
3.5.3.	Sprint Review Meeting	27
3.5.4.	Sprint Retrospective Meeting.....	27
3.6.	Backlogs.....	27
3.6.1.	Product Backlog	27
3.6.2.	Sprint Backlog	28
3.7.	Vor- und Nachteile von Scrum	28
3.7.1.	Vorteile:	28
3.7.2.	Nachteile:	29
3.8.	Warum Scrum?.....	29
4.	Frontend	31
4.1.	Graphical User Interface (GUI)	31
4.2.	Styleguide	31

4.3.	Views	32
4.3.1.	Hauptfenster.....	32
4.3.2.	Menüstruktur.....	32
4.3.3.	General Settings Dialog	33
4.3.4.	Log Dialog	34
4.3.5.	Close Dialog	35
4.3.6.	About Dialog	35
4.3.7.	Excel to	36
4.3.8.	Show Warnings	38
4.3.9.	Export to CFG Dialog.....	39
4.3.10.	Export to XML Dialog	40
4.3.11.	Preview Configuration	41
4.3.12.	XML to	42
4.3.13.	Compare Excel with XML	43
4.3.14.	Compare with Stationloader.....	46
4.3.15.	Stationtype Mapping	47
4.3.16.	Preview Stationtype-Mapping	49
4.4.	JavaFX	50
4.4.1.	Was ist JavaFX?	50
4.4.2.	Lebenszyklus	50
4.4.3.	public void init()	50
4.4.4.	public abstract void start(Stage primaryStage)	51
4.4.5.	public void stop()	51
4.4.6.	Scene-Graph	53
4.4.7.	Node	53
4.4.8.	Stage	53
4.4.9.	Scene.....	54
4.4.10.	FXML.....	55
4.4.11.	CSS.....	56
4.4.12.	RichTextFX.....	57
4.4.13.	Warum JavaFX?	58
4.4.14.	Vorteile von JavaFX im Vergleich zu WPF.....	58
4.4.15.	Nachteile von JavaFX im Vergleich zu WPF	58
5.	Excel-Reader.....	60
5.1.	ParseExcel.....	60
5.1.1.	Einlesen der Stationen.....	63
5.2.	Apache POI	63
5.2.1.	Öffnen von Workbooks.....	63
5.2.2.	Einlesen von Spreadsheets.....	64
5.2.3.	Cell	64
5.2.4.	FORMULA.....	65
6.	XML-Generierung	66

6.1.	Java DOM.....	71
6.1.1.	Elemente in DOM	71
6.1.1.1.	Node.....	71
6.1.1.2.	Document.....	72
6.1.1.3.	Element.....	72
6.1.1.4.	Comment.....	72
6.1.1.5.	Attr	72
6.1.1.6.	Text.....	72
6.1.2.	Suche von Elementen im DOM.....	72
6.1.3.	Konvertierung	73
6.1.3.1.	Umwandlung von DOM in STRING.....	73
6.1.3.2.	Transformer	73
6.1.3.3.	Source-Baum.....	73
6.1.3.4.	Result.....	73
6.1.3.5.	OutputProperties	74
6.1.3.6.	LsSerializer	74
6.1.3.7.	Konvertierung von String zu DOM	75
6.1.3.8.	Einlesen einer Datei in ein DOM-Objekt.....	75
6.1.3.8.1.	Sax Parser	75
6.1.3.8.2.	DOM Parser	76
6.1.3.9.	Validierung von DOM-Objekten.....	76
7.	Validierung	77
7.1.	XSD.....	77
7.1.1.	Namespace	77
7.1.2.	Aufbau	78
7.1.2.1.	Elemente	78
7.1.2.1.1.	Complex Type.....	78
7.1.2.1.2.	Global Types	78
7.1.2.1.3.	Compositors	78
7.1.2.1.4.	Sequence	78
7.1.2.1.5.	Choice	78
7.1.2.1.6.	Attribute	79
7.1.2.1.7.	Use	79
	XSD-Schema Beispiel	80
	Passendes XML-Element	80
7.1.3.	Attribute	80
7.1.3.1.	Name	80
	XSD-Schema Beispiel	81
	Passendes XML-Element	81
7.1.3.2.	Type.....	81
7.1.3.3.	Data Type	81
7.1.3.4.	Kardinalität.....	82

7.1.4.	Validierung einer XML-Datei mittels XSD in Java	83
8.	Lambda-Ausdrücke.....	84
8.1.	Was sind Lambda-Ausdrücke?	84
8.1.1.	Body	84
8.1.2.	Argument List.....	84
8.2.	Rückgabetypen.....	85
8.3.	Lambda-Ausdrücke oder anonyme Klassen?	85
8.4.	Instanziierung während der Laufzeit.....	86
8.5.	Vergleich zwischen Lambda-Ausdrücken und anonymen Klassen	86
8.5.1.	Syntax.....	86
8.5.2.	Variablenbindung	86
8.5.3.	Aufwand zur Laufzeit	87
8.5.4.	Sichtbarkeitsbereich (Scope) von Variablen.....	87
8.6.	Funktionale Interfaces.....	88
8.7.	Warum wurden Lambdas eingeführt?	89
8.7.1.	Neue Collection-Funktionen.....	89
8.7.1.1.	Iterieren von Collections mithilfe von Lambdas	89
8.7.1.1.1.	Externes Iterieren	90
8.7.1.1.2.	Internes Iterieren	90
8.8.	Throw-und Return-Statements in Lambda-Ausdrücken	90
8.9.	Break-Statement	91
9.	Comparator	92
9.1.	Einlesen	92
9.2.	Vergleichen.....	93
9.2.1.	Vergleich von WarehouseModelManagern	93
9.2.2.	Vergleich von Workflows.....	94
9.2.3.	Vergleich von Stationen.....	94
9.3.	Anzeigen	94
10.	Testing	95
10.1.	TestNG	95
10.1.1.	Vorteile von TestNG	95
10.1.2.	TestNG im Vergleich zu JUnit	96
10.1.3.	Annotationen	96
10.1.3.1.	Vorteile von Annotationen	98
10.1.4.	Generierte Dateien	98
10.1.4.1.	HTML Bericht (index.html).....	99
10.1.5.	testng.xml & ant.....	101
10.1.6.	Beispielhafter Aufbau eines TestNG-Testfalles	103
11.	Verwendete Tools	105
11.1.	SonarQube	105
11.1.1.	Allgemein	105
11.1.2.	Qualitätsmerkmale	106

11.1.3.	Quality-Gates	107
11.1.4.	Statische Code-Analyse.....	108
11.1.4.1.	Software Metriken	108
11.1.4.1.1.	Objektorientierte Metriken.....	108
11.1.4.1.2.	Umfangs Metriken.....	109
11.1.4.1.3.	Metriken zur Komplexität	110
11.1.5.	Dynamische Code-Analyse.....	110
11.1.6.	SonarQube im Vergleich	111
11.1.6.1.	Checkstyle	111
11.1.6.2.	PMD.....	111
11.1.6.3.	FindBugs.....	111
11.1.7.	Warum SonarQube	112
11.2.	Jenkins	112
11.2.1.	Allgemein	112
11.2.2.	Kontinuierliche Integration.....	113
11.2.3.	Plugin: SonarQube	114
11.3.	Log4J	115
VI.	Resümee	116
VII.	Glossar	117
VIII.	Abbildungsverzeichnis.....	118
IX.	Literaturverzeichnis.....	120

1. Einleitung und Überblick

Diese Diplomarbeit beschreibt folgende Teile der Anwendung:

- Benutzeroberfläche
- *Comparator*
- *Excel-Reader*

Des Weiteren werden folgende Teile des Entwicklungsprozesses beschrieben:

- Prozessmodell Scrum
- Verwendete Drittsoftware

Die anderen Teile der Anwendung werden in der Diplomarbeit von Daniel Kuglitsch,

Bernhard Rader, Marco Reitbrecht, Stefan Piber veranschaulicht und beschrieben.

Da auf diese Diplomarbeit im Verlauf dieser noch öfter referenziert wird, wird sie aus Gründen der sprachlichen Vereinfachung als „Diplomarbeit A“ bezeichnet.

1.1. Problemstellung

Die Knapp AG verwendet eine eigens entwickelte Software, um ihre Lagerhäuser zu simulieren. Dieses Programm benötigt, um diese Simulation durchführen zu können, die Struktur der Lagerhäuser. Da diese Strukturen teils sehr komplex sein können, wird das Microsoft Office-Programm Excel für die Erfassung verwendet.

Abbildung 1: Beispiel Excel Tabelle (Quelle Excel-Logo: [35])

Das Simulationsprogramm der Knapp AG kann jedoch die Simulation nicht mit Excel-Dateien durchführen, dafür sind XML-Dateien notwendig.



```

<simulator xmlns="http://www.knapp.com/sandbox/simulator">

    <workflow name="congoodsin">

        <default-driver>
            <interval-driver interval="500"/>
        </default-driver>

        <element className="com.knapp.sandbox.simulator.warehouse.element.RangeSource" name="rLC01">

            <successors>
                <element-ref element="${congoodsin.rLC01_t_LC01}"/>
            </successors>
            <configuration>
                <number-low>10000</number-low>
                <number-high>19999</number-high>
                <format>TOTE&06</format>
                <meta-data>
                    <entry key="length" type="java.lang.Integer" value="0"/>
                    <entry key="width" type="java.lang.Integer" value="0"/>
                    <entry key="height" type="java.lang.Integer" value="0"/>
                    <entry key="tara_weight" type="java.lang.Integer" value="1500"/>
                    <entry key="tote_type_id" type="java.lang.Integer" value="1"/>
                </meta-data>
            </configuration>
        </element>
        <transport bufferSize="5" name="rLC01_t_LC01" successor="${congoodsin.LC01}"/>
    </workflow>
</simulator>
```

Abbildung 2: Beispiel XML Datei (Quelle: XML Icon: [62])

Der Aufbau dieser zwei Dateitypen unterscheidet sich jedoch sehr stark, was dazu führte, dass Mitarbeiter der Knapp AG die Excel-Dateien händisch in ein XML-Format bringen mussten. Dieser Prozess ist enorm zeit- und arbeitsaufwändig und deshalb auch stark fehleranfällig.

1.2. Aufgabenstellung

Zu Beginn des Projektes war es unsere Aufgabe ein Programm in Java zu realisieren welches die in dem Abschnitt Problemstellung erläuterten Probleme auf ein Minimum reduziert und den Mitarbeitern die zeitaufwändige Arbeit des händischen Übertragens der Daten abnimmt. Die Konvertierung soll von Excel zu XML und umgekehrt möglich sein.

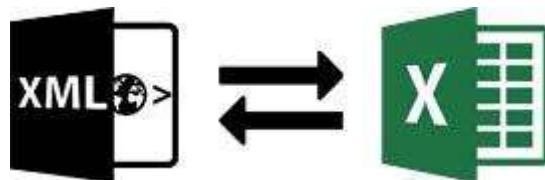


Abbildung 3: XML to Excel

In den späteren Entwicklungsphasen war die Dokumentation des Programms für die eventuelle spätere Erweiterung durch Mitarbeiter zusätzlich gefordert.

Als Prozessmodell entschieden wir uns für Scrum (siehe „Scrum“).

1.3. Vorgaben

Die Knapp AG hat uns vor Start des Projekts bereits ein Lastenheft zukommen lassen, in welchem die Anforderungen mit Mockups beschrieben waren.

1.4. Kapitelübersicht

- Prozessmodell Scrum (Kapitel 2)

Im Kapitel „Prozessmodell Scrum“ wird auf das während der Softwareerstellung angewendete Prozessmodell eingegangen.

- Software-Architektur (Kapitel 3)

Im Kapitel „Software-Architektur“ wird der Aufbau und das Zusammenspiel der einzelnen Komponenten der Anwendung beschrieben.

- Frontend (Kapitel 4)

Im Kapitel „Frontend“ wird das „Graphical User Interface“ und die damit einhergehenden FX-Elemente beschrieben und auf die grundlegende Bedienung der Benutzeroberfläche eingegangen.

- Excel-Reader (Kapitel 5)

Das Kapitel „Excel-Reader“ umfasst die Beschreibung, wie Excel-Dateien mit Hilfe von Apache POI eingelesen werden können.

- XML-Generierung (Kapitel 6)

Im Kapitel „XML-Generierung“ geht es darum, wie die eingelesenen Excel-Dateien in das XML-Format umgewandelt und als Datei gespeichert werden können.

- Validierung (Kapitel 7)

Im Kapitel „Validierung“ wird auf die Validierung der generierten XML-Dateien mithilfe des XSD-Schemas eingegangen.

- Comparator (Kapitel 8)

Im Kapitel „Comparator“ wird erläutert wie eingelesene Workflows und deren Stationen miteinander verglichen werden können.

- Testing (Kapitel 9)

Im Kapitel Testing wird das Verwendete Framework TestNG und die Unit Tests der Applikation beschrieben.

- Verwendete Tools (Kapitel 10)

Im Kapitel “Verwendete Tools” werden die während der Arbeit verwendeten Tools beschrieben.

Anmerkung:

Aus Gründen der sprachlichen Vereinfachung sind alle Aussagen in diesem Dokument als geschlechtsneutral anzusehen.

2. Software–Architektur

Folgende Komponenten der Anwendung werden beschrieben:

- *WarehouseModel*
- Ladevorgang einer Excel-Datei
 - *Excel-Reader*
 - *XML-Generator*
- *Comparator*

Die restlichen Funktionalitäten werden in der Diplomarbeit A beschrieben.

2.1. WarehouseModel

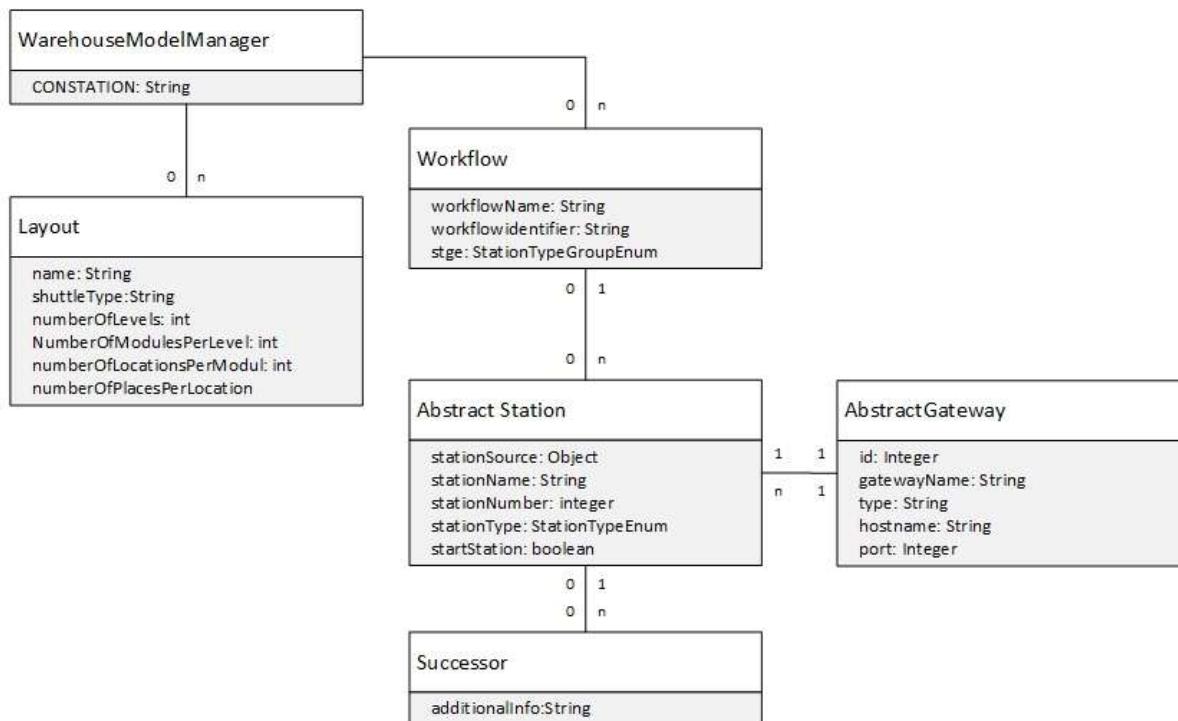


Abbildung 4: Aufbau WarehouseModel

Die Klasse *WarehouseModelManager*, auch *WarehouseModel* genannt, repräsentiert ein gesamtes Lagersystem und ist somit die wichtigste Klasse, da in ihr alle weiteren Daten, wie zum Beispiel Workflows und deren Stationen, gespeichert werden.

Zudem können einem *WarehouseModelManager* mehrere Layouts zugewiesen werden.

Ein Layout beschreibt den Aufbau und die interne Struktur des Lagersystems.

Die *AbstractStation* ist in dieser Abbildung symbolisch für alle Stationen gewählt, da sie die Superklasse aller Stationen ist. Ein Workflow kann also verschiedenste Stationen beinhalten.

Ein *Successor* stellt den Nachfolger einer Station dar. In den meisten Fällen handelt es sich dabei um eine weitere Station.

Der genauere Aufbau und Nutzen des *WarehouseModels* und dessen Komponenten wird in der Diplomarbeit A beschrieben.

2.2. Ladevorgang einer Excel-Datei

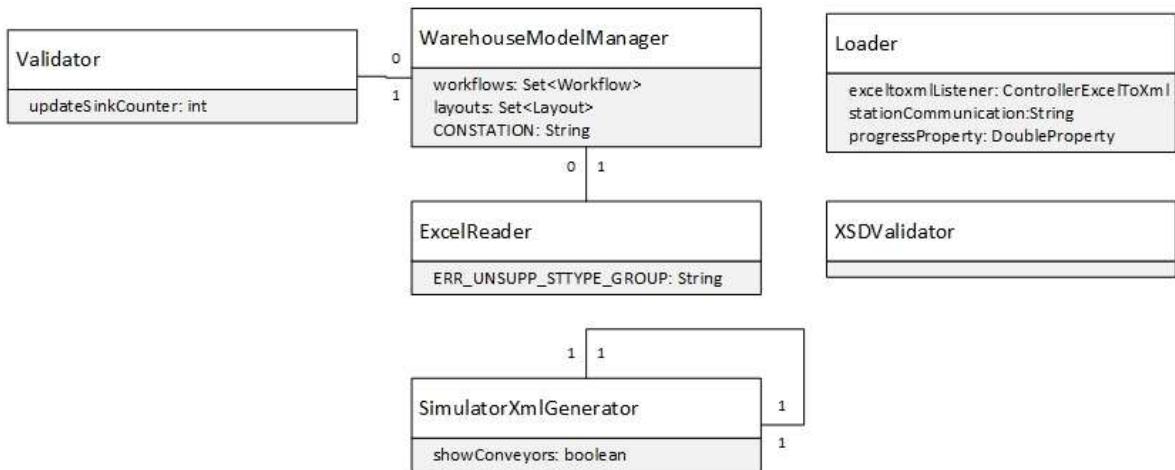


Abbildung 5:Ladevorgang einer Excel-Datei

Um die Implementierung einer *Progress Bar* zu ermöglichen wird ein Thread benötigt. Die Klasse *Loader* stellt diesen Thread dar. Sie führt alle zum Einlesen benötigten Schritte durch. Im ersten Schritt wird eine Excel-Tabelle vom *ExcelReader* auf die formale Korrektheit überprüft.

Anschließend wird das Dokument von der Klasse *ExcelReader* eingelesen und aus den daraus gewonnenen Informationen wird ein *WarehouseModel* erzeugt.

Der *Validator* prüft im nächsten Schritt die Workflows des gesamten *WarehouseModels* auf dessen Struktur und vermerkt auftretende Fehler um diese später auszugeben.

Anschließend wird das Warehouse-Model mit Hilfe der *SimulatorXmlGenerator*-Klasse in das XML-Format übertragen.

Mit Hilfe des *XSDValidators* werden die erzeugten XML-Objekte auf ihre semantische Korrektheit überprüft.

Daraufhin werden die eingelesenen Workflows auf der Benutzeroberfläche angezeigt (siehe „Frontend – Excel to ...“)

2.2.1. ExcelReader

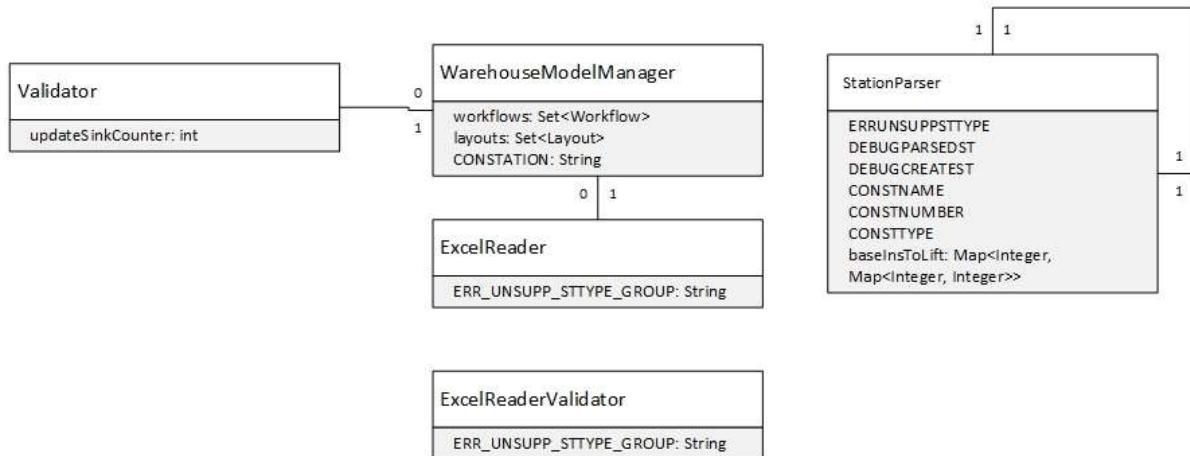


Abbildung 6: Aufbau ExcelReader

Der *Excel-Reader* dient dem Einlesen und Speichern der Excel-Dateien in den *WarehouseModelManager*.

Die dafür wichtigste Klasse ist die *ExcelReader*-Klasse. Das Einlesen selbst übernimmt die *StationParser*-Klasse.

Mithilfe der *ExcelReaderValidator*-Klasse kann überprüft werden, welche Art von Element beziehungsweise welche Art von Station eingelesen werden soll.

Der gesamte Ablauf wird im Kapitel *Excel-Reader* ausführlicher beschrieben.

2.2.2. XML-Generator

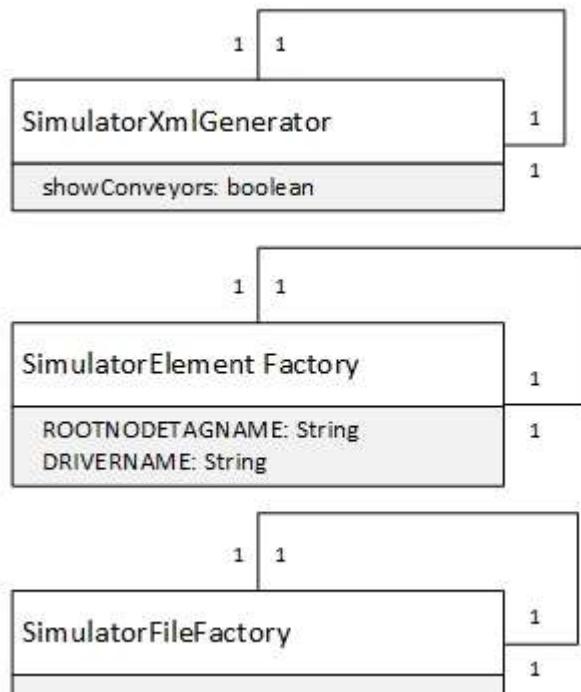


Abbildung 7:Aufbau XML-Generator

Aus dem WarehouseModel werden mit Hilfe des *SimulatorXMLGenerators* die zugehörigen XML-Dokumente erstellt. Dabei wird auf das *Factory Method Pattern* insofern zurückgegriffen, als dass die einzelnen XML-Dokumente mithilfe einer *SimulatorFileFactory* erstellt werden. Zusätzlich greift die *SimulatorFileFactory* wiederum auf die *SimulatorElementFactory* zu, um die Einzelteile der Dokumente zu erstellen. Des Weiteren werden alle zusätzlich vom Simulator benötigten Dateien, wie die *workflow-main*, die *Gateway* und die Properties Dateien von der *SimulatorFileFactory* erstellt. Weitere Informationen zum Ablauf sind im Kapitel XML-Generierung zu finden.

2.3. Comparator

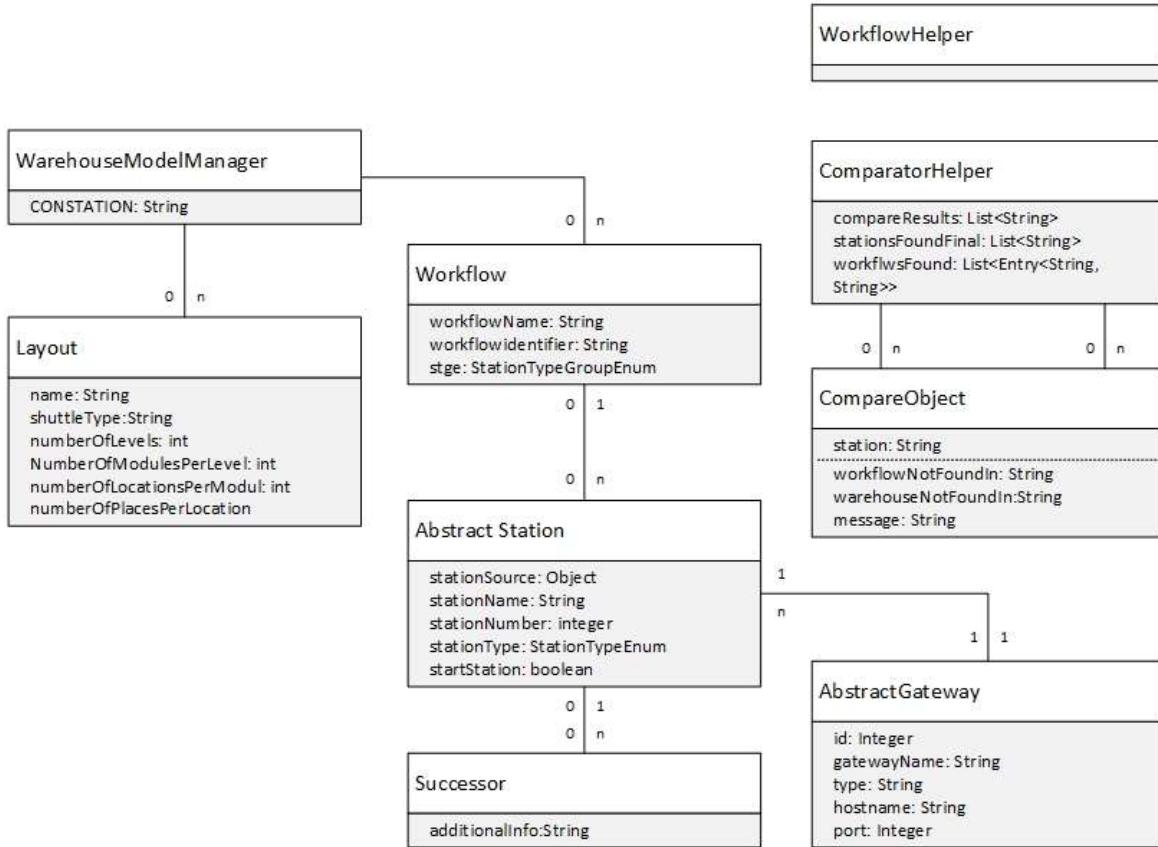


Abbildung 8: Aufbau Comparator

Der Comparator ist nicht als Java-Klasse zu verstehen, sondern eher als eine Ansammlung von Equals-Funktionen für die zu vergleichenden Klassen. Verglichen werden das *WarehouseModel*, die sich darin befindenden Workflows, die Stationen dieser und die verschiedenen Gateways. Gateways, deren Nutzen, Funktionalität und Aufbau werden in der Diplomarbeit A beschrieben. Die *AbstractStation* ist in dieser Abbildung wieder symbolisch für alle Stationen gewählt. Selbiges gilt für *AbstractGateway*.

Die Klasse *ComparatorHelper* ruft die Methoden für das Einlesen der zu vergleichenden Dateien auf und dient als Zwischenspeicher, in dem die gleichen bzw. ungleichen Elemente vermerkt werden.

Im *WorkflowHelper* befinden sich Equals- und Compare-Methoden, um die verschiedenen Workflows vergleichen zu können.

Das *CompareObject* wird auch für die Ausgabe des Ergebnisses auf der GUI verwendet.

3. Prozessmodell Scrum

Wie von der Firma Knapp gewünscht, wurde für die Projektentwicklung und Planung das agile Prozessmodell Scrum verwendet.

3.1. Kennzeichen von Agiler Softwareentwicklung

Agile Softwareentwicklung kommt, im Vergleich mit herkömmlichen Vorgehensmodellen wie Rational Unified Process oder Wasserfallmodell, mit wenigen Regeln und geringem bürokratischem Aufwand aus. Das Vorgehen ist iterativ. Es wird sich mehr auf den Kunden bzw. Kundengespräche und nicht auf vorher festgelegte Dokumentation fokussiert. Ziel ist es, auf Änderungen schnell und agil reagieren zu können. Ebenso soll der Entwicklungsprozess schlanker und somit flexibler gemacht werden. (Vgl.: [1])

3.2. Allgemein

Scrum ist eine Methode zur agilen Software-Entwicklung. Es hat einfache Regeln, wenige Rollen, ein iteratives und inkrementelles Vorgehen. Das Team organisiert seine Arbeitsweise selbst, die Konzentration wird auf Kundengespräche anstatt auf Dokumentation gesetzt. Des Weiteren ist die Veränderung der Anforderungen durch den Kunden ständig möglich.

Grundsätzlich werden große Projekte in Teile gegliedert. Diese Teile werden Inkremeante genannt. Die Inkremeante werden in „Sprints“ abgearbeitet.

Wichtig ist, dass das Inkrement am Ende eines Sprints für sich selbst funktionsfähig ist. Aufgrund der für sich fertigen Inkremeante kann man frühzeitig Fehler in der Entwicklung erkennen und ausbessern.

(Vgl.: [2])

Während unseres Projektes stellten unsere firmeninternen Betreuer die Kunden dar, welche uns die Anforderungen geliefert haben.

3.3. Sprint

Im Sprint wird das Produkt-Inkrement entwickelt. Im Normalfall dauert ein Sprint 30 Tage, während unseres Projektes jedoch dauerte ein Sprint nur knapp 2 Wochen.

Er startet immer mit dem Sprint Planning Meeting und wird mit dem Sprint Review Meeting abgeschlossen.

Am Beginn jeden Tages wird ein Daily Scrum Meeting gehalten. Während eines Sprints dürfen sich die Anforderungen vonseiten des Auftraggebers nicht ändern. Wichtig ist, dass die Arbeit am Ende eines Sprints fertig sein muss.

(Vgl.: [2])

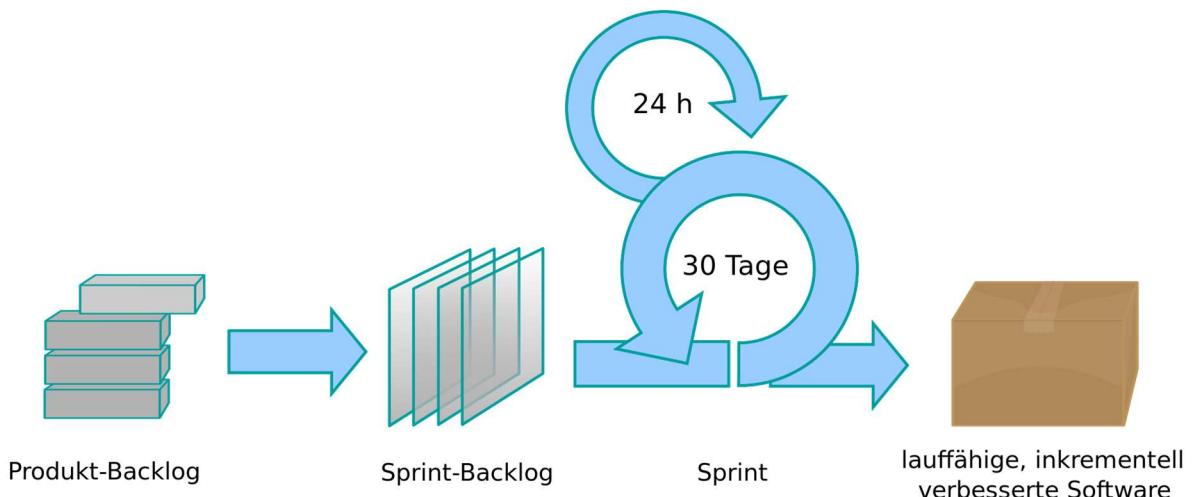


Abbildung 9: Ablauf von Scrum (Quelle: [2])

3.4. Rollen

In Scrum gibt es 3 Rollen.

3.4.1. Product Owner

Er ist der Auftraggeber vom Projekt. Der Product Owner pflegt den Product Backlog, nimmt an Scrum Meetings teil und steht für Rückfragen vom Scrum Team bereit. Er braucht nicht unbedingt Programmierkenntnisse, aber sollte in seinem Themenfeld ein Experte sein.

(Vgl.: [4])

Diese Rolle wurde von einem unserer firmeninternen Betreuer übernommen.

3.4.2. Scrum Master

Er managt den Prozess und beseitigt Hindernisse. Der Scrum Master regelt den Informationsfluss zwischen Product Owner und Team.

Außerdem hält er Scrum-Meetings.

Generell hat er eine beratende, aber keine kontrollierende Funktion. (Vgl.: [4])

Diese Rolle wurde von einem unserer firmeninternen Betreuer übernommen.

3.4.3. Scrum Team

Das Scrum Team besteht aus den Entwicklern. Sie setzen die Anforderungen in Produktfunktionalität um. Das Team muss interdisziplinär zusammengesetzt sein (Entwickler, Tester usw. in einem Team).

Damit Scrum funktioniert, sollten die Mitglieder des Scrum Teams sich möglichst in allen Bereichen auskennen, da sonst bei den Meetings den anderen Mitgliedern kaum geholfen werden kann.

Das Team ist sein eigener Manager, es bestimmt selbst über seine Arbeitsweise.

Die Rollenaufteilung ist in Scrum nicht so stark wie bei anderen Modellen, da das Scrum Team aus vielen verschiedenen Bereichen der Arbeiter besteht. Trotz der Unterschiedlichen Mitarbeiter des Scrum Teams wird es zu einer Rolle zusammengefasst.
(Vgl.: [4])

3.5. Meetings

3.5.1. Sprint Planning Meeting

Das Sprint Planning Meeting findet zu Beginn des Sprints statt. Bei diesem Meeting wird aus dem Product-Backlog der Sprint-Backlog entnommen.

Zuerst präsentiert der Product Owner den Product Backlog und evtl. ein Sprint Goal. Dann schätzt das Team den Aufwand der einzelnen Product Backlog Items und gibt an, wie viel es im Sprint erledigen kann.

Der Product Owner sucht sich dann aus dem Product Backlog aus, was alles erledigt wird.
(Vgl.: [2])

3.5.2. Daily Scrum Meeting

Es ist ein tägliches, 15-minütiges Meeting zwischen allen Rollen.
Während unseres Projektes kam es jedoch auch vor, dass Daily Scrum Meetings bis zu zwei Stunden lang waren, da es vieles zu besprechen galt.

Der Scrum Master notiert sich in diesem Meeting die aufgetretenen Probleme und der Product Owner kann teilnehmen, um Informationen über die aktuelle Lage des Projekts zu erhalten. Das Scrum Team tauscht Informationen über Probleme aus und teilt mit, was es am Vortag gemacht hat bzw. gedenkt heute zu tun.

(Vgl.: [2])

Für die Teammitglieder in unserem Projekt war dieses Meeting ein großer Vorteil, da sich so jeder seinen Tag einfach strukturieren konnte, weil jeder genau wusste was er zu erledigen hatte.

3.5.3. Sprint Review Meeting

Dieses Meeting findet am Ende eines Sprints statt. Hier wird die Funktionalität des Produkt-Inkремents dem Product Owner und möglichen Interessenten präsentiert. Das Scrum Team erhält dann ein Feedback über ihre erbrachte Leistung vom Product Owner. Es können ebenso Änderungen vom Product Owner festgelegt werden. (Vgl.: [2])

Im Rahmen unseres Projektes wurde beim Sprint Review Meeting das Programm den zukünftigen Anwendern und Mitarbeitern, die das Programm in Zukunft eventuell erweitern, vorgeführt. Diese gaben uns konstruktive Kritik zur Erweiterung und Verbesserung unseres Projektes.

3.5.4. Sprint Retrospective Meeting

Dieses Meeting findet am Ende eines Sprints nach dem Sprint Review Meeting statt. Das Scrum Team überprüft seine Arbeitsweise, damit es diese in Zukunft effektiver gestalten kann. Des Weiteren unterstützt der Scrum Master das Scrum Team bei diesem Meeting. (Vgl.: [2])

Das Sprint Retrospective Meeting wurde während unseres Projektes nie gehalten, es wird hier der Vollständigkeit halber trotzdem erwähnt.

3.6. Backlogs

3.6.1. Product Backlog

Der Product Backlog beinhaltet alle Anforderungen. Die Komplexität der einzelnen Anforderungen wird mit Schätzwerten (in Personentagen) dargestellt. Es ist kein Endgültiges Dokument, da es sich im Laufe der Zeit ändern kann.

(Vgl.: [3])

In einem Sprint realisierte oder verworfene Requirements werden entfernt, Schätzwerte aufgrund Teilbearbeitung oder neuer Erkenntnisse aktualisiert, neue Anforderungen aufgenommen, Prioritäten verändert, grob geschätzte Themen oder Funktionsblöcke verfeinert, also in kleinere heruntergebrochen etc..

(Quelle: [5])

3.6.2. Sprint Backlog

Der Sprint Backlog beinhaltet alle Tasks, welche während des Sprints vom Team abgearbeitet werden sollen.

Die Liste der Tasks wird während eines Sprints von den Team-Mitgliedern gepflegt. Der Sprint Backlog hilft auch dabei, die Übersicht über den Sprint-Fortschritt zu behalten. Im Notfall kann hier auch der Scrum Master steuernd eingreifen damit das Ziel des Sprints nicht gefährdet ist.

(Vgl.: [6])

Bei unserem Projekt wurde der Sprint Backlog mit Hilfe eines Microsoft Team Foundation Servers realisiert. Auf diesem wurden uns die einzelnen Aufgaben auf Wunsch zugewiesen.

3.7. Vor- und Nachteile von Scrum

3.7.1. Vorteile:

- *Wenige Regeln, leicht verständlich und schnell einführbar*
- *Kurze Kommunikationswege*
- *Hohe Flexibilität/Agilität durch adaptives Planen*
- *Hohe Effektivität durch Selbstorganisation*
- *Hohe Transparenz durch regelmäßige Meetings und Backlogs*
- *Zeitnahe Realisation neuer Produkteigenschaften bzw. Inkrementen*
- *Kontinuierlicher Verbesserungsprozess*

- *Kurzfristige Problem-Identifikation*
- *Geringer Administrations- und Dokumentationsaufwand*

(Quelle: [3])

Großer Vorteil innerhalb des Projektes war, dass jeder zu jedem Zeitpunkt über alles bestens Bescheid wusste. Für unsere Betreuer war die Flexibilität sehr angenehm, da sich während des Projektes einige neue Ziele ergeben haben.

3.7.2. Nachteile:

- *Kein Gesamtüberblick über die komplette Projektstrecke (immer nur Teil erledigt)*
- *Hoher Kommunikations- und Abstimmungsaufwand*
- *Zeitverluste bei zu „defensiven“ Sprintplanungen*
- *„Tunnelblick-Gefahr“ bei ausschließlicher Fokussierung auf Tasks*
- *Erschwerte Koordination mehrerer Entwicklungsteams bei Großprojekten*
- *Potenzielle Verunsicherung aufgrund fehlender Zuständigkeiten und Hierarchien*
- *Potenzielle Unvereinbarkeit mit bestehenden Unternehmensstrukturen*

(Quelle: [3])

Nachteile des Prozessmodells haben sich während unseres Projektes nicht bemerkbar gemacht.

3.8. Warum Scrum?

Scrum bot sich für unser Projekt an, da von Anfang an noch nicht klar war, welche Features benötigt werden. Auch die durch Scrum ermöglichte kurzfristige Problem-Identifikation war optimal, da sich das Projekt nur auf zwei Monate belief. Durch die geringe Laufzeit des Projektes war auch der durch Scrum ermöglichte geringe Dokumentations- und Administrationsaufwand ein großer Vorteil des Prozessmodells.

Dadurch, dass Scrum in Sprints unterteilt ist und am Ende jedes Sprints eine potenziell auslieferbare Version des Programms vorliegen muss, konnten Mitarbeiter der Knapp AG diese Version bereits testen. Dieses Vorgehen erwies sich als äußerst nützlich, da so schnell auf Fehler aufmerksam gemacht werden konnte und dadurch das Projekt fehlerfreier verlief.

Innerhalb der Knapp AG wurden schon gute Erfahrungen mit Scrum für firmeninterne Projekte in der Größe unseres Projektes gemacht, da gute Ideen, die während des Projektes entstehen, einfach und unkompliziert eingebaut werden können. Diese Erfahrungen trugen erheblich dazu bei, dass sich unsere Betreuer im Endeffekt für dieses Prozessmodell entschieden.

4. Frontend

Das Frontend wurde, wie von der Knapp AG vorgegeben, mit Java und insbesondere JavaFX realisiert.

4.1. Graphical User Interface (GUI)

Alle GUIs, die während des Projekts entwickelt wurden und im fertigen Programm enthalten sind, werden nun im Folgenden beschrieben und mithilfe von Screenshots dargestellt. Zusätzlich wird der Unterschied zu den im Lastenheft vorhandenen Mockups aufgezeigt. Die Größe des Programmfensters ist beliebig großenverstellbar (Siehe „JavaFX“).

4.2. Styleguide

Der Styleguide „*Easy Use*“ ist eine firmeninterne Design-Richtlinie und garantiert ein einheitliches und einfach zu verwendendes Layout für die meisten Programme, die innerhalb des Unternehmens Anwendung finden.

4.3. Views

4.3.1. Hauptfenster

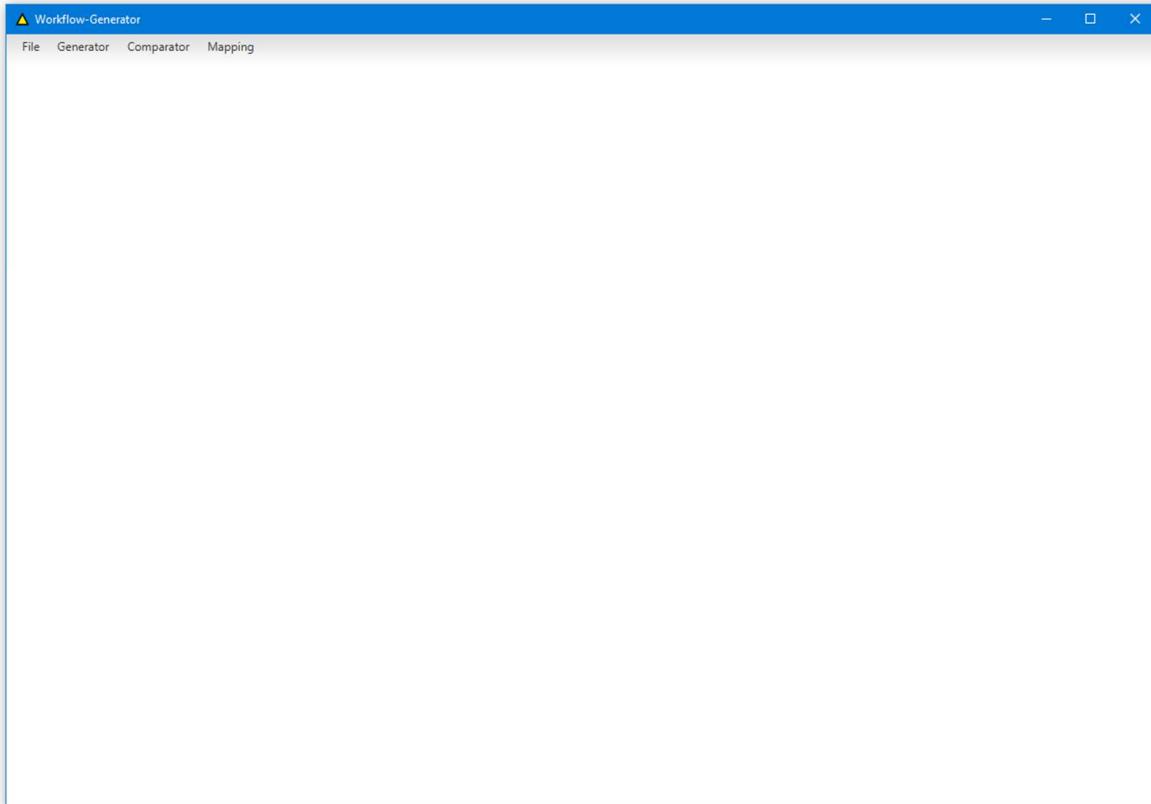


Abbildung 10: Hauptfenster

Nach Starten des Programms erhält man diese Ansicht, auf der man die einzelnen Features über eine Menüleiste anwählen kann. Nach Auswählen eines Menüpunktes wird entweder ein Dialog geöffnet oder die GUI und JavaFX Komponenten in das Hauptfenster geladen.

4.3.2. Menüstruktur

- File
 - General Settings
 - About
 - Close
- Generator
 - Excel to ...

- XML to ...
- Comparator
 - Compare Excel with XML
 - Compare Excel with Stationloader
- Mapping
 - *Stationtype Mapping*

4.3.3. General Settings Dialog

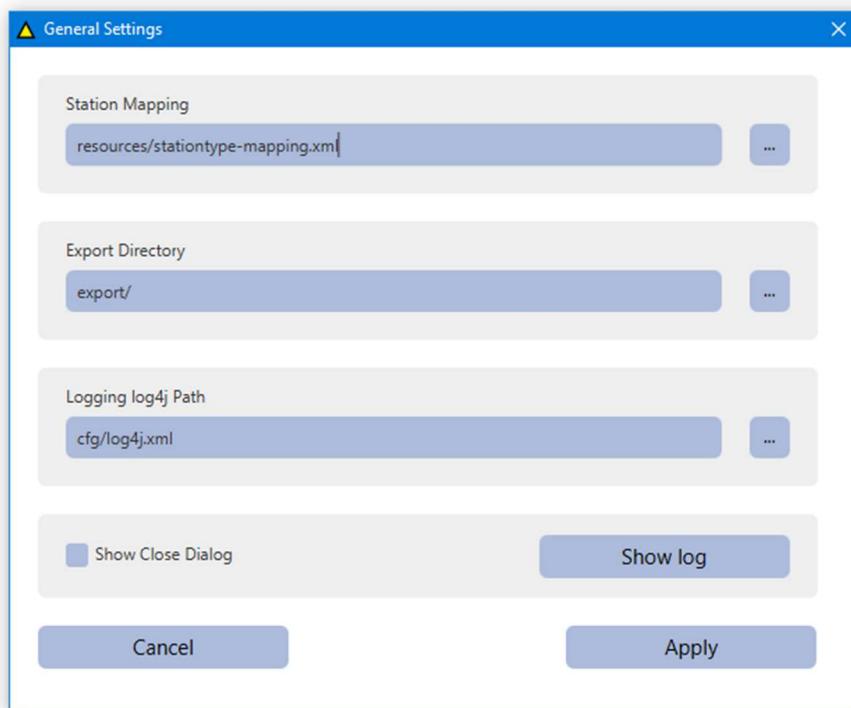


Abbildung 11: General Settings Dialog

Der Menüpunkt “General Settings” bietet die Möglichkeit, allgemeine Einstellungen für das Programm vorzunehmen. Die Einstellungen werden in der *workflow-generator.xml*-Konfigurationsdatei gespeichert und beim Start des Programms eingelesen. Sollte diese Datei fehlen, wird automatisch eine neue mit Standardwerten erstellt.

Station Mapping	Hier kann der User die gewünschte Stationtype Konfigurations-Datei angeben.
-----------------	---

Export Directory	Zeigt den Pfad des vom Benutzer festgelegten Export Directoys an.
Logging log4j Path	Legt den Pfad für das aktuell verwendete Logfile fest.
Show Close Dialog	Bestimmt, ob vor Schließen des Programms ein Popup-Fenster angezeigt werden soll (siehe „Close Dialog“), auf dem der User das Beenden nochmals bestätigen muss.
Show log	Öffnet ein weiteres Fenster, dass den Log anzeigt (siehe „Log Dialog“).

4.3.4. Log Dialog

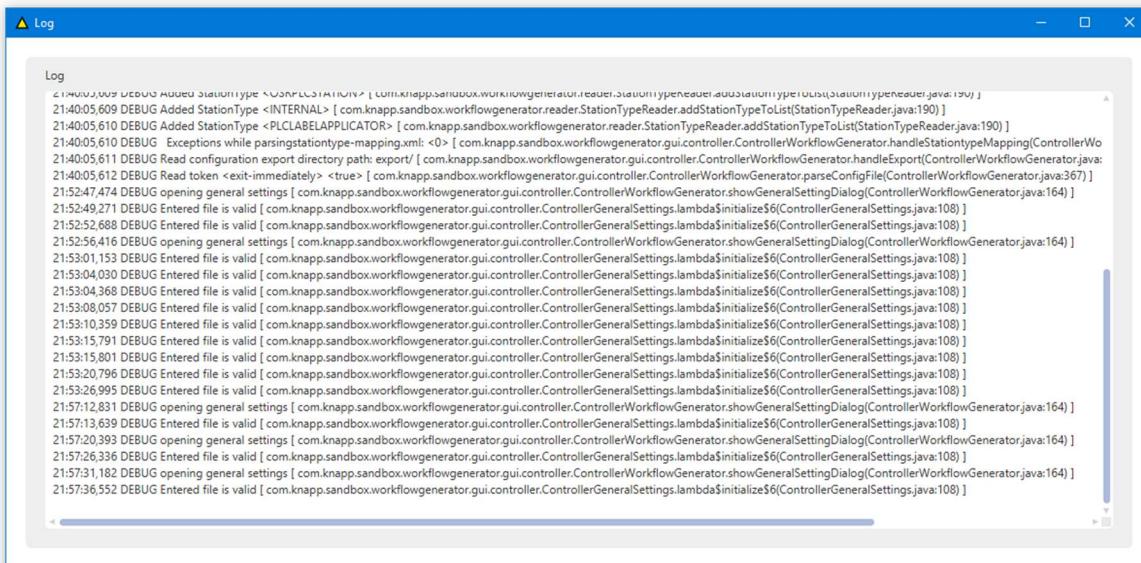


Abbildung 12: Log Dialog

In diesem Fenster wird der von Log4j generierte Output in einem Fenster innerhalb des Programms dargestellt, um eventuell übersehenen, beziehungsweise zukünftig noch dazukommenden Fehlern vorzubeugen.

4.3.5. Close Dialog

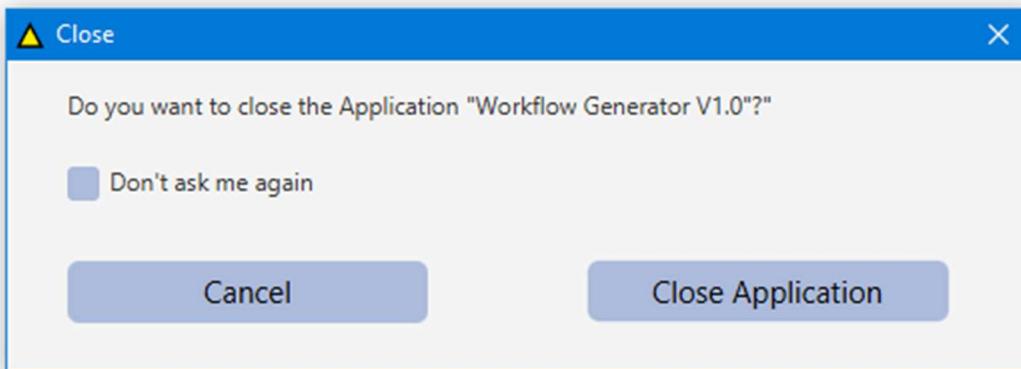


Abbildung 13: Close Dialog

Bei erstmaliger Ausführung des Programms oder wenn im General Settings Dialog (siehe „General Settings Dialog“) der Punkt „Show Close Dialog“ ausgewählt wurde, wird nach einem Klick auf X (Schließen) dieses Fenster angezeigt, auf welchem der User nochmals bestätigen muss, ob er das Programm wirklich beenden möchte.

Wählt man „Don't ask me again“, wird dieser Dialog nicht mehr angezeigt und in der *workflow-generator.xml* Datei vermerkt.

4.3.6. About Dialog

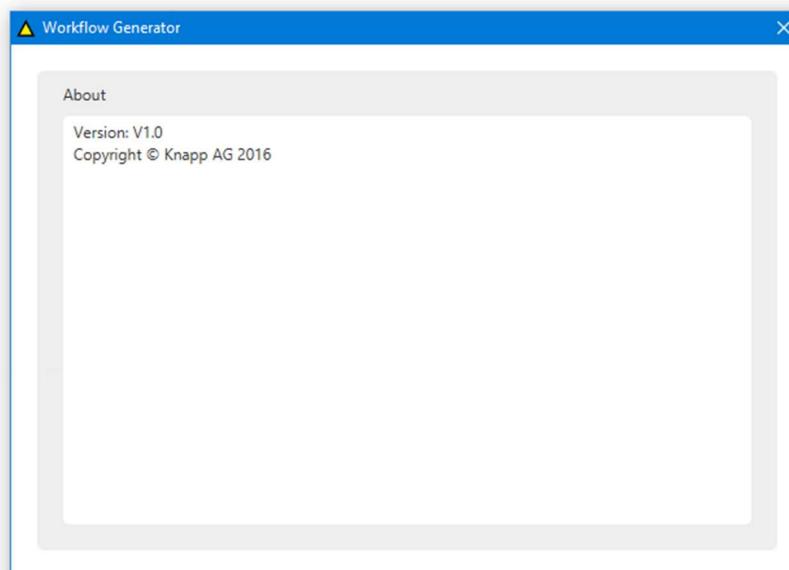


Abbildung 14: About Dialog

In diesem Dialog finden sich allgemeine Informationen, wie beispielsweise die Version des Programms und die Copyright Rechte zum Programm.

4.3.7. Excel to ...

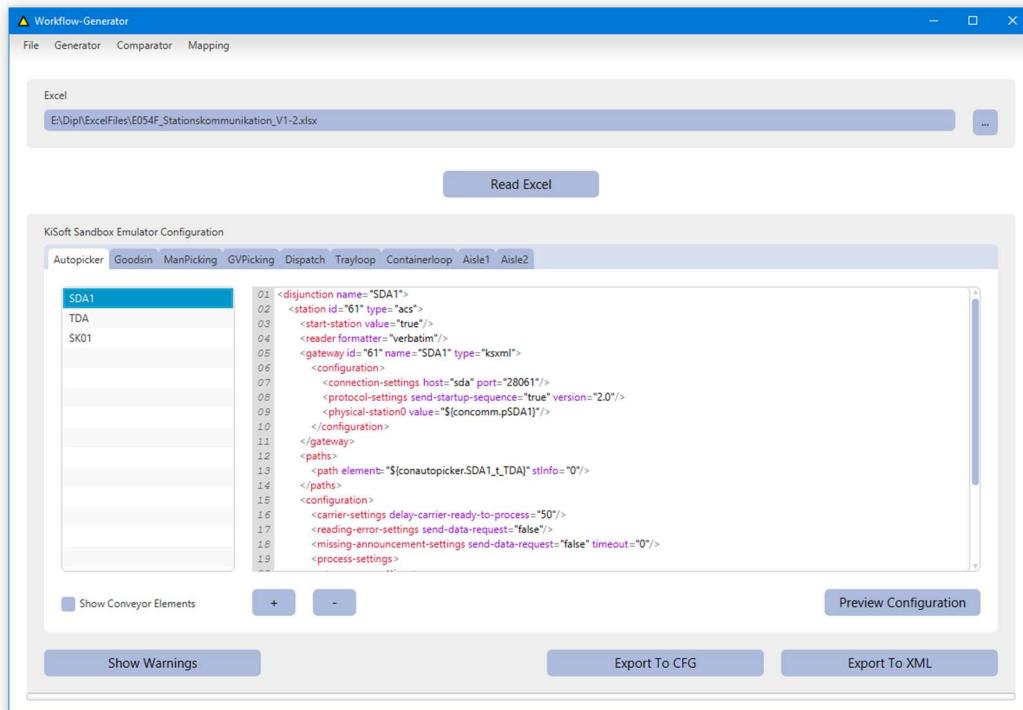


Abbildung 15: Excel to ...

Im Abschnitt “Excel to ...” können Excel-Dateien eingelesen und in verschiedene Dateitypen konvertiert werden.

Die verfügbaren Dateitypen sind:

- *Station-Definition*
- XML

Bevor die Funktionen zum Konvertieren freigegeben werden, muss der Benutzer über einen File-Chooser eine Excel-Datei auswählen und den Button “Read Excel” drücken.

Daraufhin wird die Excel-Datei in das XML-Format umgewandelt (siehe „ExcelReader“).

Der Fortschritt der Konvertierung wird dabei durch die *Progress Bar*, die sich am unteren Rand des Fensters befindet, angezeigt.

Die verschiedenen eingelesenen Workflows werden in einer *TabBar* angezeigt und sind dadurch auswählbar.



Abbildung 16: *TabBar*

Über eine Liste können die Stationen, aus denen der ausgewählte Workflow besteht, ausgewählt werden.

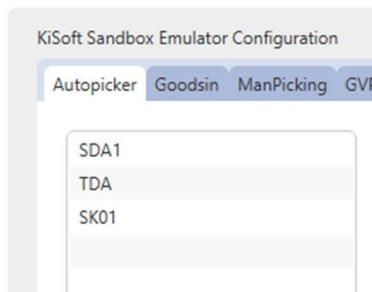


Abbildung 17: *Workflows in Liste*

Bei Auswahl einer Station wird der entsprechende XML-Text rechts neben der Liste in einer RichTextFX Code-Area (siehe „RichTextFX“) mit entsprechendem Highlighting der XML Elemente angezeigt. Des Weiteren wird links auch die zugehörige Zeile angezeigt.



Abbildung 18: *Show Warnings, Conveyor Elements, Zoom Buttons*

Durch einen Klick auf den „Preview Configurations“ Button, wird das Preview Configurations Fenster geöffnet (siehe „Preview Configurations“).

Durch einen Klick auf den Plus bzw. Minus Button ist es möglich, den angezeigten Text zu vergrößern oder zu verkleinern.

Durch die Auswahl der „Show Conveyor Elements“ Checkbox ist es möglich, in der Liste der Stationen zusätzlich die *Conveyor Elemente* anzuzeigen.

Bei Klick auf den Button „Show Warnings“ ist es möglich, jene Probleme und Fehler, die während der Konvertierung von Excel zu XML aufgetreten sind, anzuzeigen (siehe „Show Warnings“).



Abbildung 19: Export to CFG bzw. XML und Preview Configuration Button

Durch Klicken auf “Export To CFG” wird das Fenster “Configure what to generate” geöffnet (siehe „Export to CFG“).

Durch Klicken auf “Export To XML” wird das Fenster “Generate Configurations for Exporting XML” geöffnet (siehe „Export to XML“).

4.3.8. Show Warnings

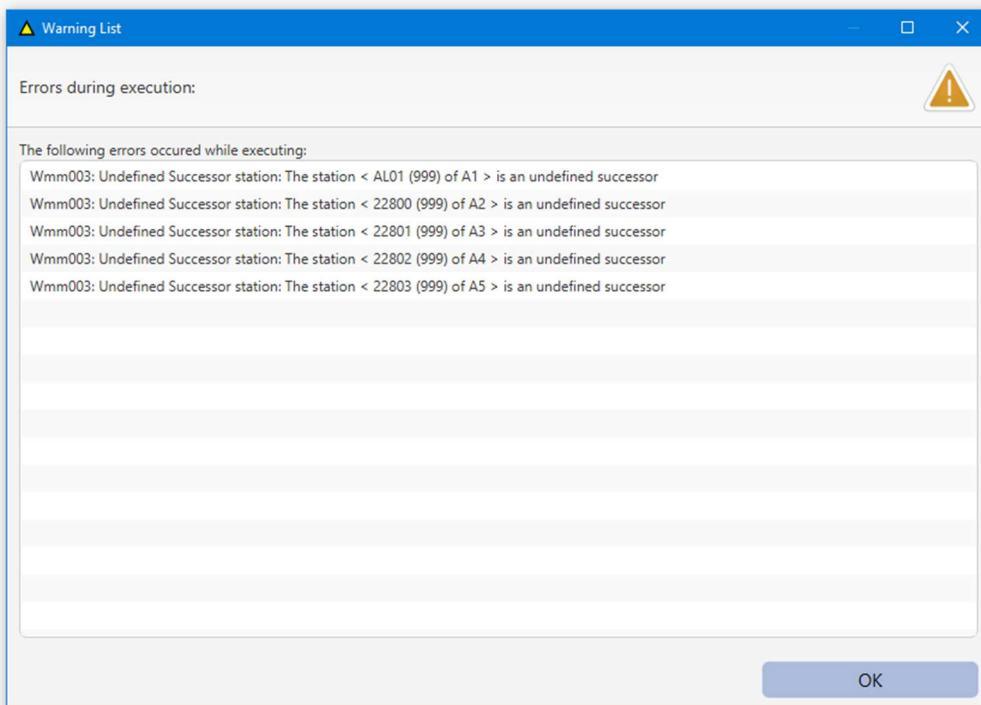


Abbildung 20: Show Warnings

Sollten beim Einlesen oder Konvertieren Fehler, wie zum Beispiel nicht definierte Nachfolger-Stationen, auftreten, werden diese automatisch nach dem Einlesen angezeigt.

Der Dialog kann jedoch auch über einen separaten Button in dem “Excel to ...” Fenster angezeigt werden (Siehe „Excel to ...“).

4.3.9. Export to CFG Dialog

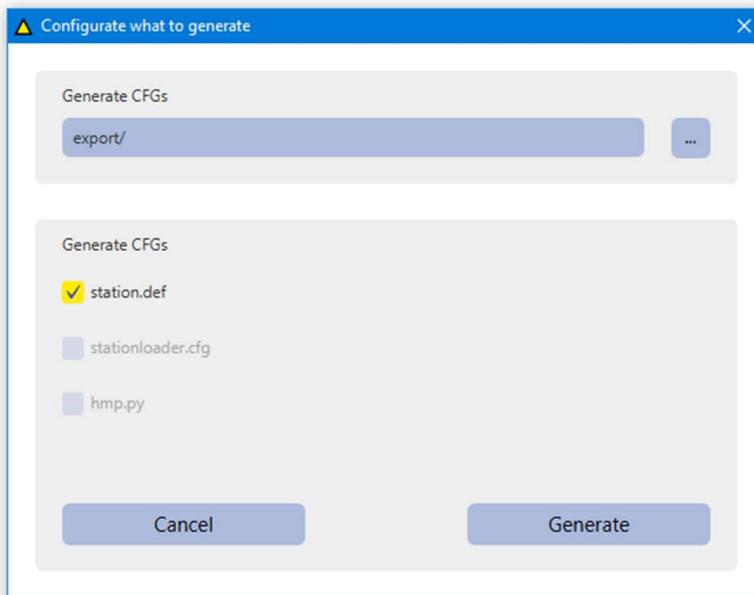


Abbildung 21: Export to CFG Dialog

Die erste der zwei Generierungsoptionen ist das Generieren von Konfigurations-Dateien.

Da diese Option erst spät in der Entwicklung gefragt war und nicht mehr genug Zeit war alles zu implementieren, ist es nur möglich eine sogenannte Station Definition zu erzeugen (siehe „Compare-Stationloader“).

Mithilfe eines File-Choosers kann der Dateipfad für die generierten Dateien angegeben werden.

4.3.10. Export to XML Dialog

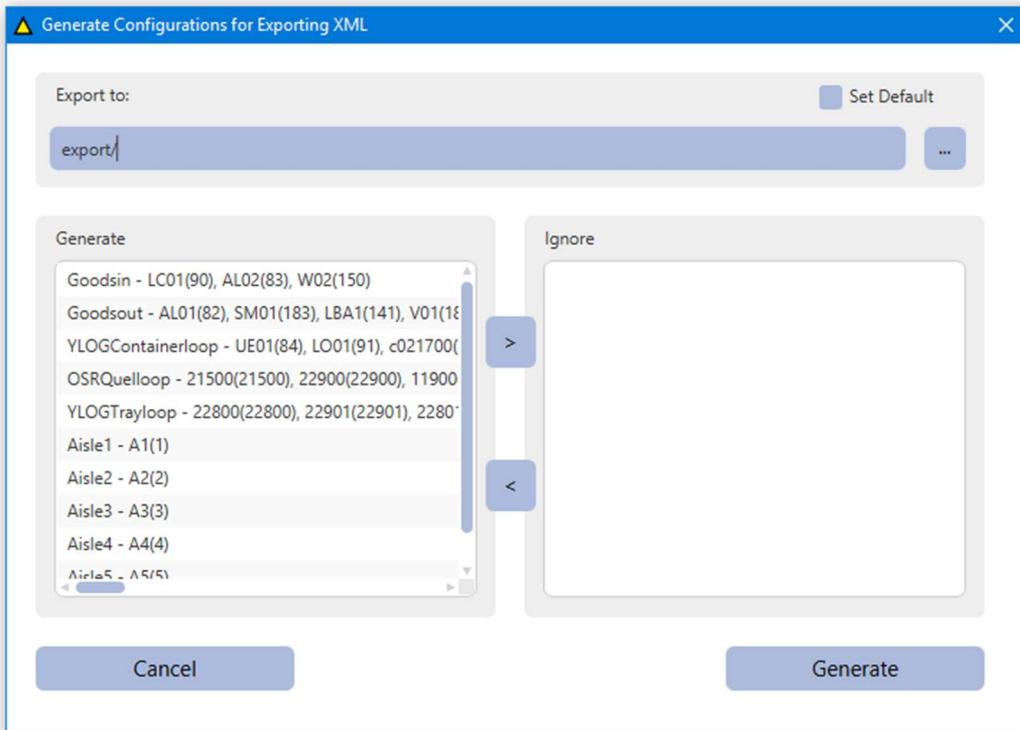


Abbildung 22: Export to XML Dialog

Die zweite Generierungsoption ist das Generieren von XML-Dateien.

Im Dialog befinden sich zwei Listen in denen sich die Workflows, welche generiert werden sollen, befinden. Mithilfe der Pfeil-Buttons können die Elemente zwischen den zwei Listen hin- und her bewegt werden.

Workflows, die sich in der linken Liste "Generate" befinden, werden, wenn man auf den Button "Generate" klickt, generiert.

Workflows, die sich in der rechten Liste "Ignore" befinden, werden, wenn man auf den Button "Generate" klickt, nicht generiert.

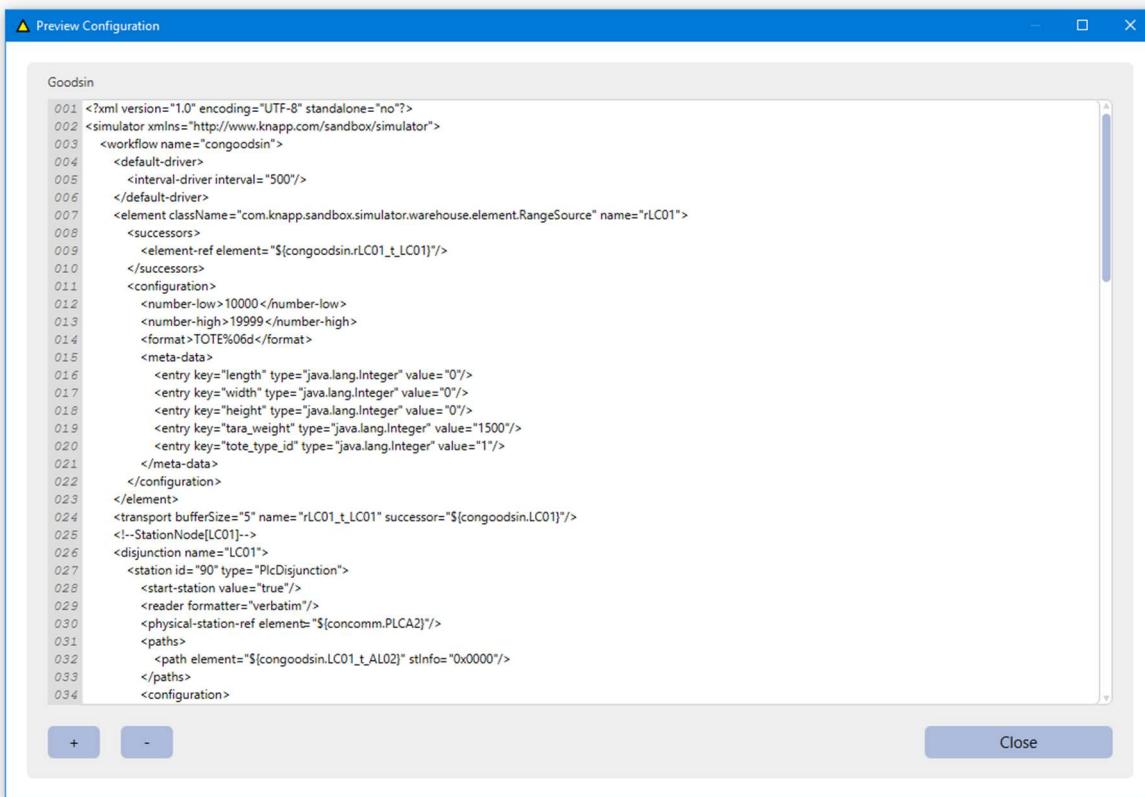
Mithilfe eines File-Choosers kann der Pfad für die generierten Dateien angegeben werden. Die Checkbox "Set Default" legt, wenn sie zum Zeitpunkt des Generierens ausgewählt ist,

den in der Textbox stehenden Pfad in die *workflow-generator.xml*-Konfigurations-Datei als Standard fest (siehe „General Settings Dialog“).

Beim Klicken auf den “Generate” Button werden folgende Dateien generiert:

- die ausgewählten Workflows
- *simulator.xml*
- *WorkflowMain.xml*
- Sechs Konfigurationsdateien für den Simulator

4.3.11. Preview Configuration



```

Goodsin
001 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
002 <simulator xmlns="http://www.knapp.com/sandbox/simulator">
003   <workflow name="congoodsin">
004     <default-driver>
005       <interval-driver interval="500"/>
006     </default-driver>
007     <element className="com.knapp.sandbox.simulator.warehouse.element.RangeSource" name="rLC01">
008       <successors>
009         <element-ref element="${congoodsin.rLC01_t_LC01}" />
010       </successors>
011       <configuration>
012         <number-low>10000</number-low>
013         <number-high>19999</number-high>
014         <format>TOTE%06d</format>
015         <meta-data>
016           <entry key="length" type="java.lang.Integer" value="0"/>
017           <entry key="width" type="java.lang.Integer" value="0"/>
018           <entry key="height" type="java.lang.Integer" value="0"/>
019           <entry key="tara_weight" type="java.lang.Integer" value="1500"/>
020           <entry key="tote_type_id" type="java.lang.Integer" value="1"/>
021         </meta-data>
022       </configuration>
023     </element>
024     <transport bufferSize="5" name="rLC01_t_LC01" successor="${congoodsin.LC01}" />
025   <!--StationNode[LC01]-->
026   <disjunction name="LC01">
027     <station id="90" type="PcDisjunction">
028       <start-station value="true"/>
029       <reader formatter="verbatim"/>
030       <physical-station-ref element="${concomm.PLCA2}" />
031       <paths>
032         <path element="${congoodsin.LC01_t_AL02}" stlInfo="0x0000"/>
033       </paths>
034     </configuration>

```

Abbildung 23: Preview Configuration

Im “Preview Configuration” Fenster kann sich der Benutzer ansehen, wie der im Fenster “Excel To ...” ausgewählte Workflow innerhalb einer generierten XML-Datei aussehen würde. Dieses beinhaltet alle Stationen und *Conveyor-Elemente* eines Workflows, sowie einige XML-spezifische Elemente.

Es dient auch dazu, die Stationen auf einem größeren Fenster angezeigt zu bekommen.

Am oberen Rand des Fensters befindet sich der Name des momentan angesehenen Workflows.

Wie auch schon im “Excel to ...”-Fenster kann mithilfe der + und - Buttons die Schriftgröße vergrößert beziehungsweise verkleinert werden.

4.3.12. XML to ...

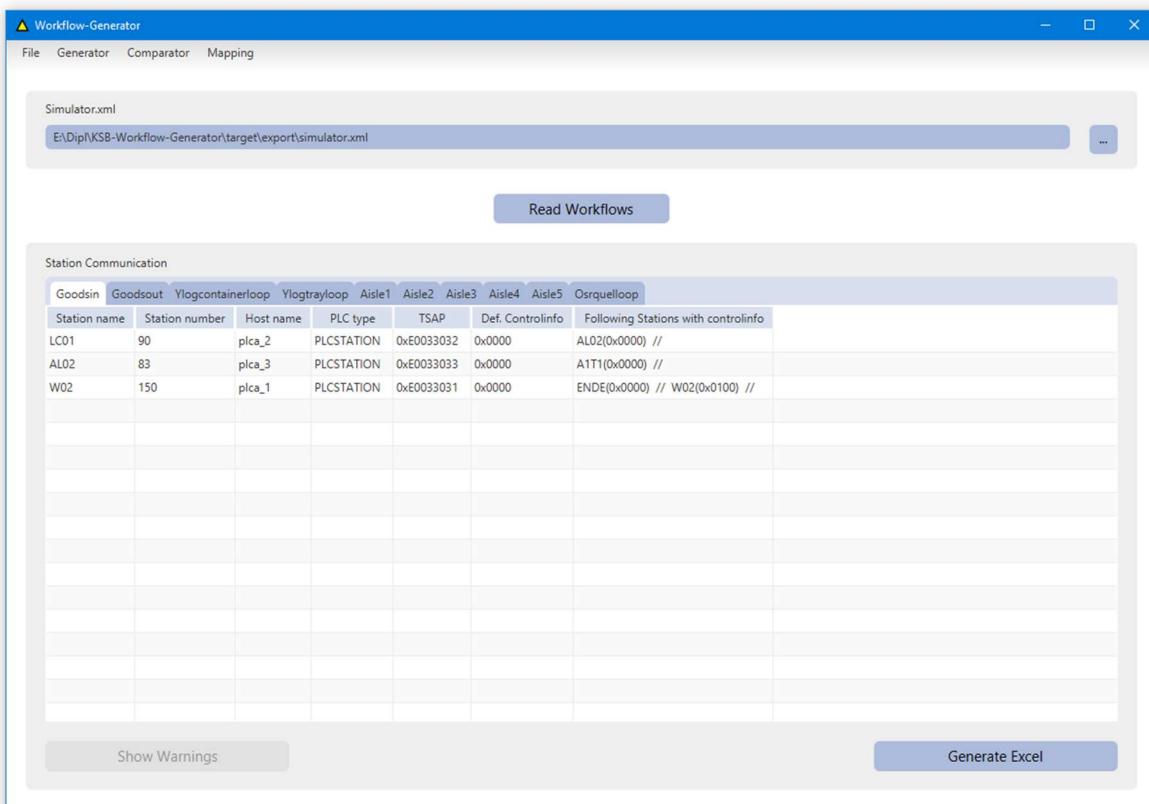


Abbildung 24: XML to ...

Im Menüpunkt “XML to ...” kann ein *simulator.xml* (Siehe „XML-Generierung“) in eine Excel-Datei konvertiert werden.

Bevor die Funktionen zum Konvertieren freigegeben werden, muss der Benutzer über einen File-Chooser eine *simulator.xml*-Datei auswählen und den Button „Read Workflows“ drücken.

Station Communication							
Goodsin	Goodsout	Ylogcontainerloop	Osrquelloop	Ylogtrayloop	Aisle1	Aisle2	Aisle3
Station name	Station number	Host name	PLC type	TSAP	Def. Controlinfo	Following Stations with controlinfo	
LC01	90	plca_2	PLCSTATION	0xE0033032	0x0000	AL02(0x0000) //	
AL02	83	plca_3	PLCSTATION	0xE0033033	0x0000	A1T1(0x0000) //	
W02	150	plca_1	PLCSTATION	0xE0033031	0x0000	ENDE(0x0000) // W02(0x0100) //	

Abbildung 25: XML to ... Vorschau Tabelle

Nachdem die Datei eingelesen wurde, werden die Daten in eine Tabelle geladen, welche eine Excel-Preview darstellt.

Durch Klicken des Generate Excel Buttons wird aus den eingelesenen Daten eine Excel-Datei erzeugt.

4.3.13. Compare Excel with XML

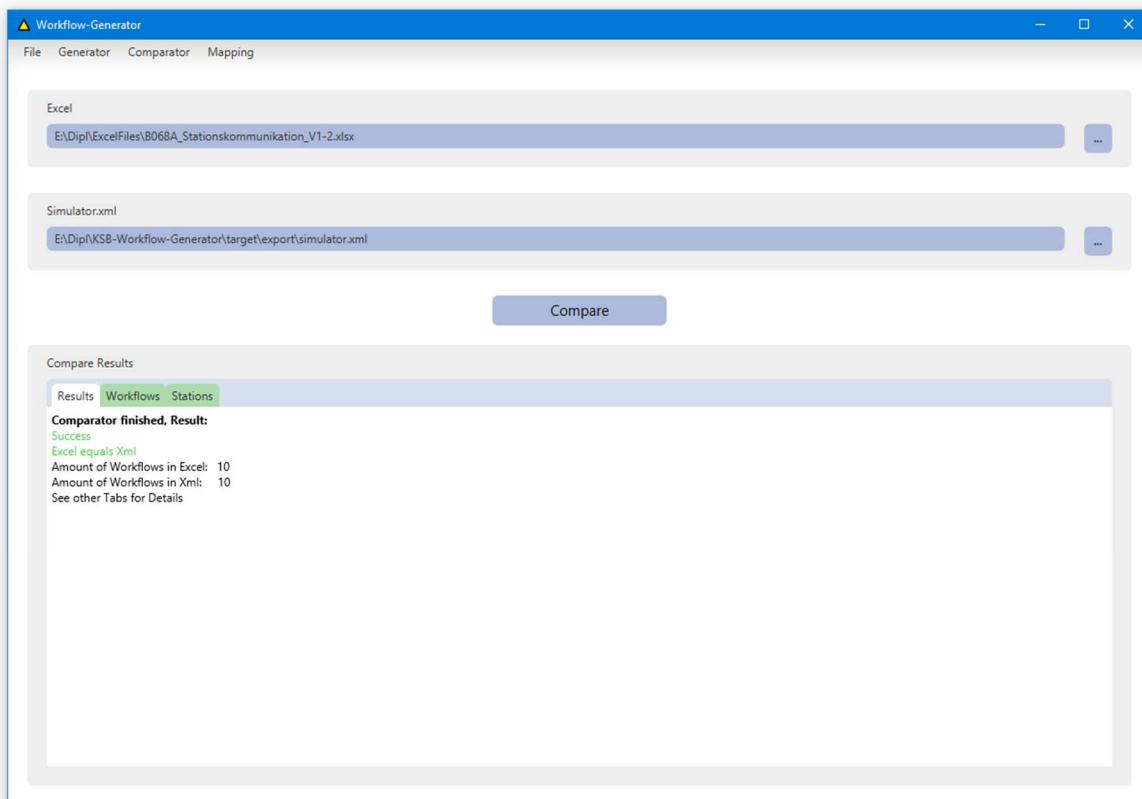


Abbildung 26: Compare Excel with XML

Im Menüpunkt “Compare Excel with XML” können die eingelesen Daten aus jeweils einer Excel-Datei und einer simulator.xml Datei, genannt “*Warehousemodel*” (siehe „Excel-Reader“), miteinander verglichen werden.

Durch Drücken des Buttons “Compare” wird der oben genannte Vergleich durchgeführt.

Nachdem dieser erfolgt ist, werden Details angezeigt. Sollten beide *Warehousemodels* ident sein, werden die erzeugten Tabs grün gefärbt.



Abbildung 27: Results Tab

Im Results-Tab finden sich generelle Informationen zum Vergleich.

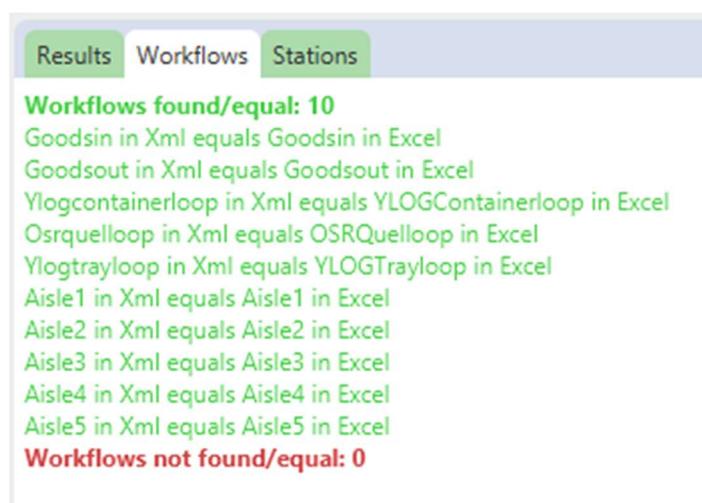


Abbildung 28: Results Workflows Tab

Im Workflows-Tab befinden sich alle Workflows aus beiden *Warehousemodels* und ob diese ident sind oder nicht. Tritt der Fall ein, dass beide *Warehousemodels* ident sind, werden die Tabs grün gefärbt, andernfalls werden die Tabs rot angezeigt.

Results	Workflows	Stations
Stations found/equal: 88		
LC01, goodstat was found and is equal in Excel and Xml		
AL02, goodstat was found and is equal in Excel and Xml		
WL03, goodstat was found and is equal in Excel and Xml		
AL01, goodout was found and is equal in Excel and Xml		
SM01, goodout was found and is equal in Excel and Xml		
LB01, goodout was found and is equal in Excel and Xml		
V01, goodout was found and is equal in Excel and Xml		
SV01ter, goodout was found and is equal in Excel and Xml		
SV01tramp, goodout was found and is equal in Excel and Xml		
UB01, ylogcontainerloop was found and is equal in Excel and Xml		
LC01, ylogcontainerloop was found and is equal in Excel and Xml		
c021700, ylogcontainerloop was found and is equal in Excel and Xml		
21700, ylogcontainerloop was found and is equal in Excel and Xml		
c021701, ylogcontainerloop was found and is equal in Excel and Xml		
21701, ylogcontainerloop was found and is equal in Excel and Xml		
c021702, ylogcontainerloop was found and is equal in Excel and Xml		
21702, ylogcontainerloop was found and is equal in Excel and Xml		
c021703, ylogcontainerloop was found and is equal in Excel and Xml		
21703, ylogcontainerloop was found and is equal in Excel and Xml		
c2400, ylogcontainerloop was found and is equal in Excel and Xml		
21400, ylogcontainerloop was found and is equal in Excel and Xml		
c021402, ylogcontainerloop was found and is equal in Excel and Xml		
21402, ylogcontainerloop was found and is equal in Excel and Xml		
c11501, ylogcontainerloop was found and is equal in Excel and Xml		
d5501, ylogcontainerloop was found and is equal in Excel and Xml		
5601, ylogcontainerloop was found and is equal in Excel and Xml		
LC02, ylogcontainerloop was found and is equal in Excel and Xml		
s5501, ylogcontainerloop was found and is equal in Excel and Xml		
21803, ylogcontainerloop was found and is equal in Excel and Xml		
21401, ylogcontainerloop was found and is equal in Excel and Xml		
21802, ylogcontainerloop was found and is equal in Excel and Xml		
21801, ylogcontainerloop was found and is equal in Excel and Xml		
21800, ylogcontainerloop was found and is equal in Excel and Xml		
c2000, ylogcontainerloop was found and is equal in Excel and Xml		
21500, esqueleloop was found and is equal in Excel and Xml		
c021500, esqueleloop was found and is equal in Excel and Xml		
22800, vlootrailloop was found and is equal in Excel and Xml		

Abbildung 29: Result Staions Tab

Im Workflows-Tab befinden sich alle Stationen aus allen Workflows beider *Warehousemodels* und ob der Inhalt und der Aufbau dieser ident ist. Stationen, welche sowohl im Excel- als auch im XML-*Warehousemodel* gefunden werden und ident sind, werden im Bereich “Stations found/equal” mit der Farbe Grün gekennzeichnet. Stationen welche nicht gefunden werden bzw. Stationen welche zwar in beiden *Warehousemodels* gefunden werden, sich aber nicht ähneln, werden im Bereich “Stations not found/equal” angezeigt und rot markiert.

4.3.14. Compare with Stationloader

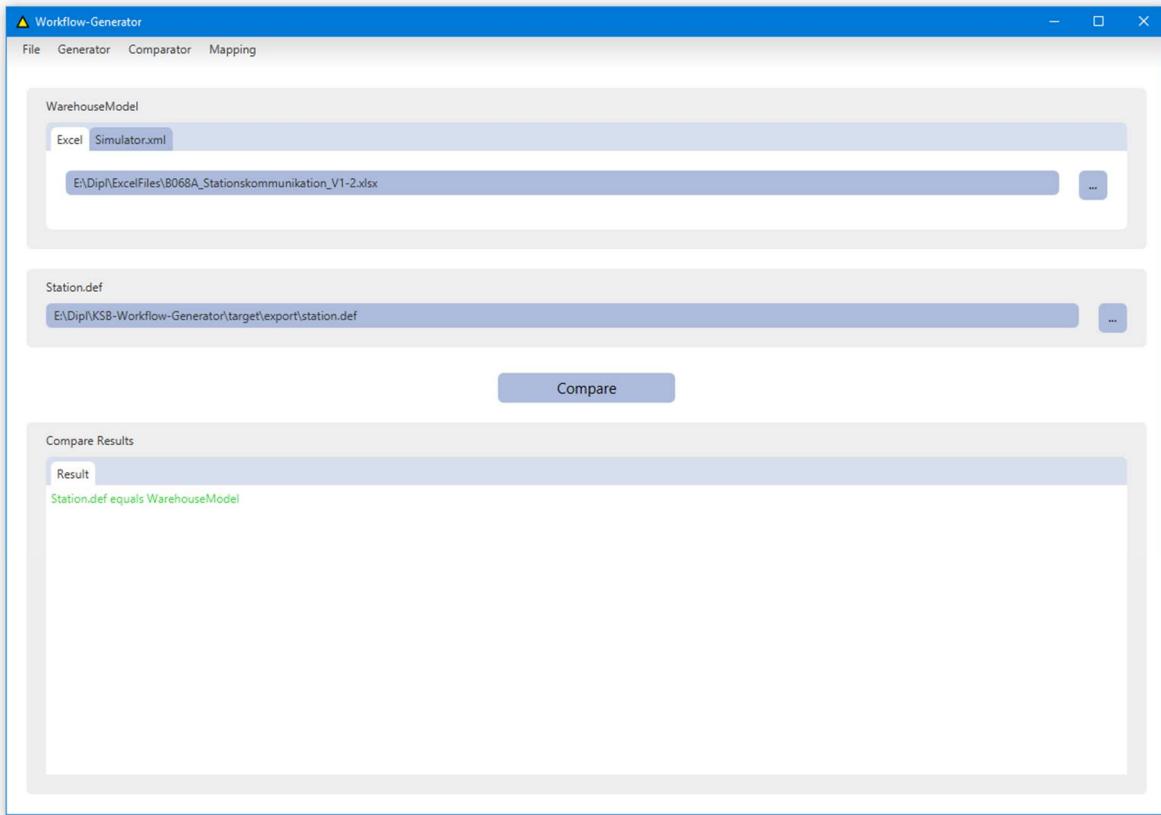


Abbildung 30: Compare with Stationloader

Im Menüpunkt “Compare with Stationloader” kann wahlweise eine Excel- oder simulator.xml Datei mit einer *Station-Definition* verglichen werden.

Durch Klicken auf “Compare” wird festgestellt, ob die angegebene Excel- oder simulator.xml Datei den Definitionen der *station.def* entspricht oder nicht.



Abbildung 31: Compare with Stationloader Result

Im Results-Tab findet sich das Ergebnis des Vergleichs.

4.3.15. Stationtype Mapping

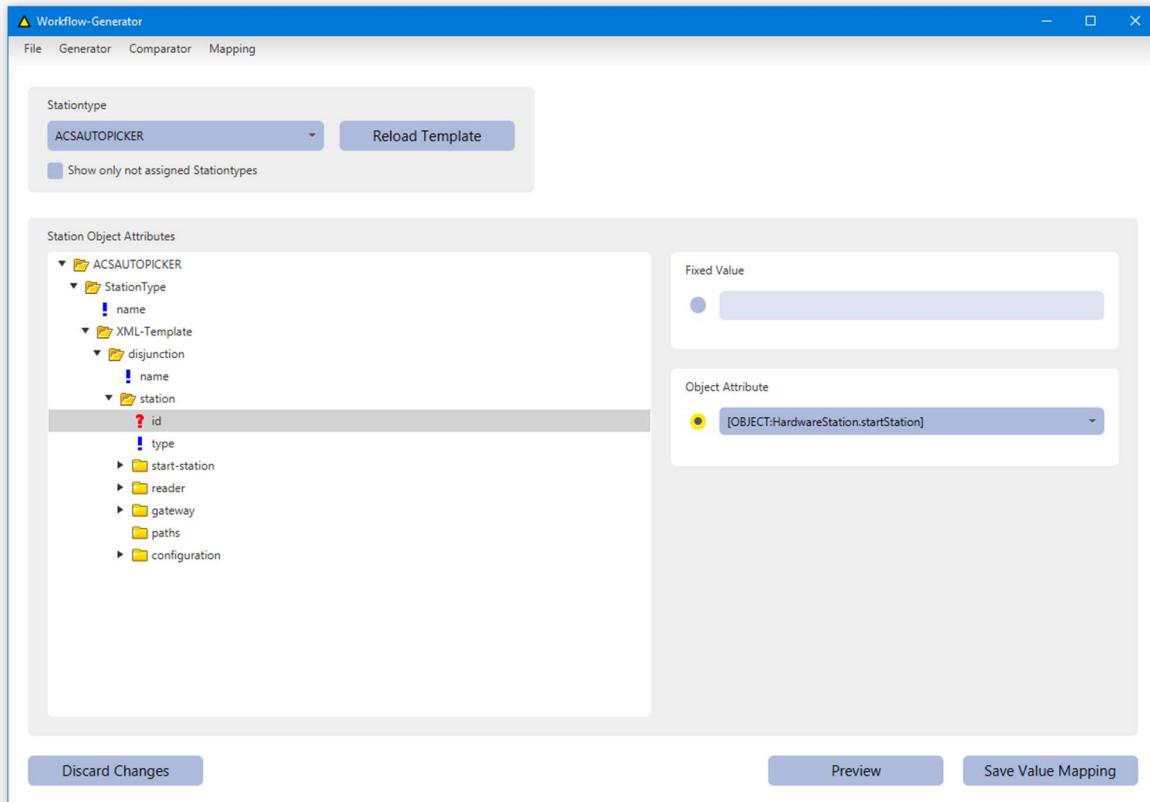


Abbildung 32: Stationtype Mapping

Im Menüpunkt „*Stationtype Mapping*“ kann der Benutzer das zur Erstellung der XML-Dateien notwendige Template bearbeiten.

Eben dieses Template ist in einer XML-Datei gespeichert. Wenn man ein Element in der Combobox auswählt, wird automatisch dessen Aufbau in Form einer Baumstruktur in die darunter liegende Ansicht geladen.



Abbildung 33: Icon Ordner

Sollte ein XML-Tag ein Unterelement besitzen, wird vor dem Namen ein Ordnersymbol angezeigt.

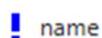


Abbildung 34: Icon Blaues Rufzeichen

Sollte das XML-Tag kein Unterelement besitzen, wird anstatt eines Ordnersymbols ein blaues Rufzeichen angezeigt, welches symbolisiert, dass der Wert dieses Tags nicht verändert wurde.



Abbildung 35: Icon rotes Fragezeichen

Für den Fall, dass ein XML-Tag verändert wird, wird dessen Icon auf ein rotes Fragezeichen geändert.

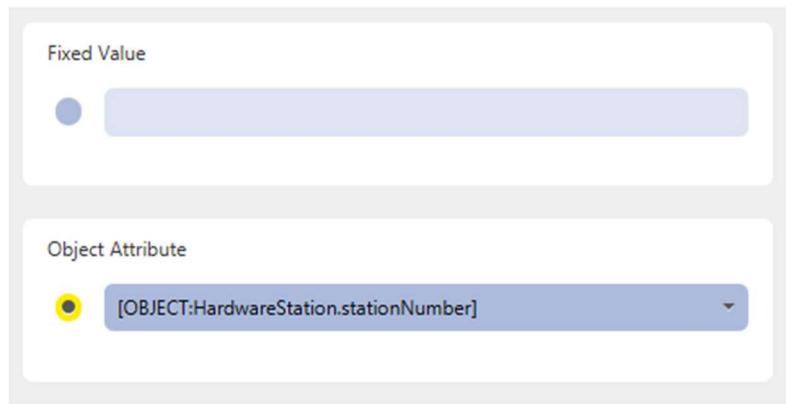


Abbildung 36: Fixed Value, Object Attribute

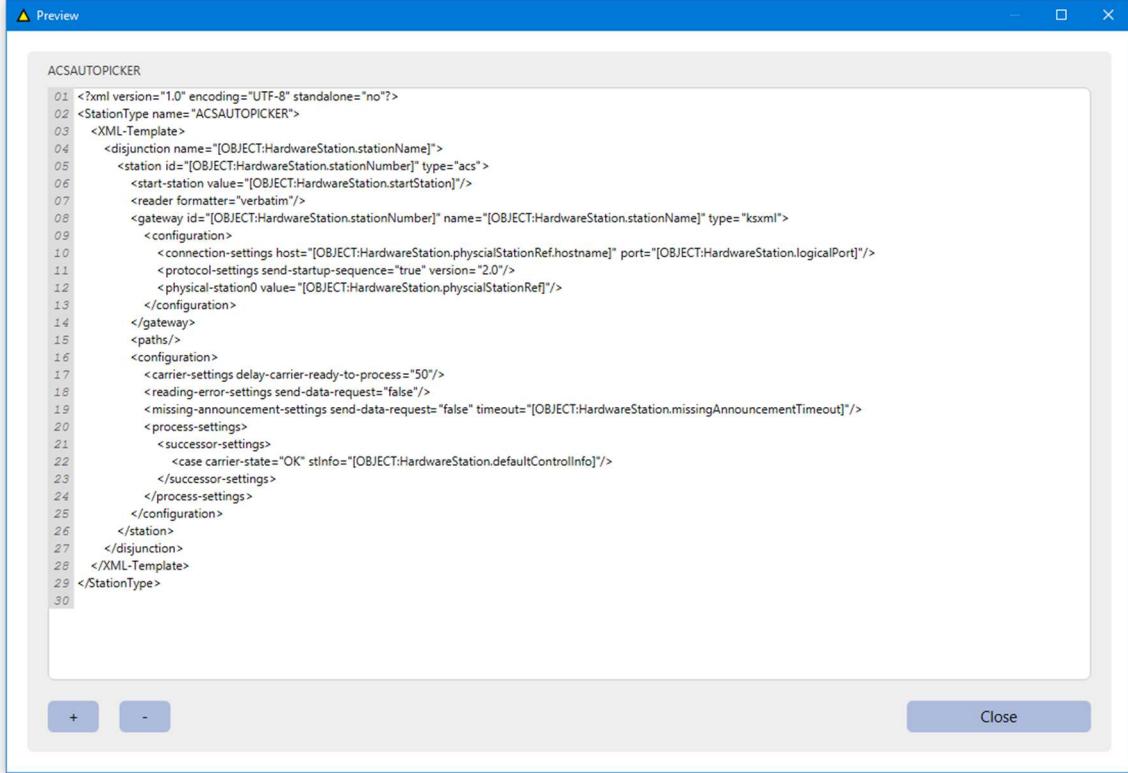
Der Benutzer kann sich entscheiden, ob er einen vorgefertigten Wert aus der Combobox verwendet. In der Combobox befinden sich alle Object-Attribute, die sich in dem *Stationtype-Mapping*-Template befinden.

Sollte sich das gewünschte Attribut nicht in der Combobox befinden, kann dieses auch über Freitext eingefügt werden.

Mithilfe von “Discard Changes” können alle vorgenommenen Änderungen wieder rückgängig gemacht werden.

Bei Klick auf den “Save Value Mapping” Button ist es möglich, die durchgeführten Änderungen zu übernehmen.

4.3.16. Preview Stationtype-Mapping



The screenshot shows a 'Preview' window with the title 'ACSAUTOPICKER'. The main area contains the following XML code:

```
01 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
02 <StationType name="ACSAUTOPICKER">
03   <XML-Template>
04     <disjunction name="[{OBJECT:HardwareStation.stationName}]">
05       <station id="[{OBJECT:HardwareStation.stationNumber}]" type="acs">
06         <start-station value="[{OBJECT:HardwareStation.startStation}]"/>
07         <reader formatter="verbatim"/>
08         <gateway id="[{OBJECT:HardwareStation.stationNumber}]" name="[{OBJECT:HardwareStation.stationName}]" type="ksxml">
09           <configuration>
10             <connection-settings host="[{OBJECT:HardwareStation.physicalStationRef.hostname}]" port="[{OBJECT:HardwareStation.logicalPort}]" />
11             <protocol-settings send-startup-sequence="true" version="2.0"/>
12             <physical-station0 value="[{OBJECT:HardwareStation.physicalStationRef}]" />
13           </configuration>
14         </gateway>
15         <paths/>
16         <configuration>
17           <carrier-settings delay-carrier-ready-to-process="50"/>
18           <reading-error-settings send-data-request="false"/>
19           <missing-announcement-settings send-data-request="false" timeout="[{OBJECT:HardwareStation.missingAnnouncementTimeout}]" />
20           <process-settings>
21             <successor-settings>
22               <case carrier-state="OK" stInfo="[{OBJECT:HardwareStation.defaultControlInfo}]" />
23             </successor-settings>
24           </process-settings>
25         </configuration>
26       </station>
27     </disjunction>
28   </XML-Template>
29 </StationType>
30
```

At the bottom left are two buttons: a blue '+' button and a grey '-' button. At the bottom right is a blue 'Close' button.

Abbildung 37: Previews Stationtype-Mapping

Im „Preview Stationtype-Mapping“ Fenster kann sich der Benutzer die von ihm im „Stationtype Mapping“ Fenster (siehe „Stationtype Mapping“) vorgenommenen Änderungen in Form der XML-Struktur anzeigen lassen. (siehe „Preview Configuration“)

4.4. JavaFX

4.4.1. Was ist JavaFX?

JavaFX ist ein von Oracle entwickeltes Framework für das Erstellen, Entwerfen und Debuggen von plattformübergreifenden Java Anwendungen.

Es ist eine Java Bibliothek, welche aus, in Java geschriebenen, Klassen und Interfaces besteht.

Somit kann JavaFX direkte Aufrufe von der Java Standard API durchführen.

(Vgl.: [8])

4.4.2. Lebenszyklus

JavaFX Anwendungen erweitern die JavaFX-Klasse Applikation, dadurch werden die Methoden der Application-Klasse, welche den Lebenszyklus definieren, vererbt.

Die Methoden die den Lebenszyklus definieren sind:

- *public void init()*
- *public abstract void start (Stage primaryStage)*
- *public void stop()*

Direkt zu Beginn der Anwendung wird eine Instanz der Application-Klasse von der JavaFX-Runtime instanziert.

(Vgl.: [7])

4.4.3. public void init()

Diese Methode wird direkt, nachdem die JavaFX Runtime ein Application-Objekt instanziert hat, ausgeführt.

Standardmäßig ist diese Methode leer.

Diese Methode wird üblicherweise dazu verwendet, anwendungsspezifische Instanziierungen durchzuführen. GUI-spezifische Funktionen wie das Erstellen und

Verwalten von Stages oder Scenes sind innerhalb dieser Methode noch nicht möglich, da es dem JavaFX Framework zu diesem Zeitpunkt noch nicht möglich ist eine grafische Benutzeroberfläche zu erzeugen.

(Vgl.: [7])

4.4.4. public abstract void start(Stage primaryStage)

Diese Methode wird, direkt nachdem das JavaFX Framework den Thread, welcher die grafischen Komponenten verwaltet, erzeugt hat, ausgeführt. Sie wird innerhalb dieses Threads ausgeführt.

Eine primäre Stage, am besten vergleichbar mit einem Startfenster, wird dieser Methode als Argument mitgeliefert.

Innerhalb der Application Klasse können in dieser Methode nun grafische Komponenten erzeugt, der primären Stage kann eine Scene zugeordnet und die Stage sichtbar gemacht werden.

Da diese Methode abstrakt ist, muss jede JavaFX-Anwendung diese Methode überschreiben.

(Vgl.: [7])

4.4.5. public void stop()

Wenn eine Anwendung beendet wird, wird diese Methode ausgeführt.

Dies kann durch das manuelle Schließen des Fensters (sofern implicitExit auf true gesetzt ist) und Plattform.exit() geschehen. Standardmäßig ist diese Methode leer.

Diese Methode wird üblicherweise dazu verwendet, um von der Anwendung verwendete Ressourcen wieder freizugeben.

(Vgl.: [7])

```
public class Lebenszyklus extends Application {  
    public void main(String[] args) {  
        System.out.println("Starten der JavaFX-Applikation");  
        launch(args);  
    }  
    public void init() {  
        System.out.println("init() Methode");  
    }  
    public void start(Stage primaryStage) {  
        System.out.println("start() Methode");  
    }  
    public void stop() {  
        System.out.println("stop() Methode - Beenden der Anwendung");  
    }  
}
```

Dieser Code liefert folgenden Output:

Starten der JavaFX-Applikation

init() Methode

start() Methode

stop() Methode - Beenden der Anwendung

(Vgl.: [7])

4.4.6. Scene-Graph

Scene-Graphs verwalten die einzelnen Bestandteile einer JavaFX GUI und wissen zu jeder Zeit, welche Objekte angezeigt werden müssen, welche Bereiche des Bildschirms neu gezeichnet werden müssen und wie alles möglichst Effizient gerendert werden soll.

Der Aufbau eines Scene-Graphs gleicht dem einer Baumstruktur, welche alle grafischen Komponenten (z.B. Pane) der GUI beinhaltet.

Die einzelnen Elemente dieser Struktur werden Node genannt.

(Vgl.: [9], [10])

4.4.7. Node

Nodes werden je nach ihrer Position unterteilt in:

- Root Node

Sie haben keinen Parent-Node und sind in jedem Scene-Graph jeweils nur einmal vorhanden. Die Applikation erkennt das Root-Node anhand der Root-Property.

- Branch Node

Sie haben einen Parent-Node und mindestens einen Child-Node

- Leaf Node

Sie haben einen Parent Node und keinen Child-Node. Sie stellen die sichtbaren Elemente der GUI dar.

(Vgl.: [9], [10])

4.4.8. Stage

Eine Stage kann grundsätzlich als Programmfenster verstanden werden.

Sie wird dazu verwendet, eine oder mehrere Scenes darzustellen (siehe „Scene“). Die primäre Stage wird von jeder JavaFX-Anwendung bei Start generiert. Optional können auch noch andere Stages erstellt werden.

Wenn Stage-Objekte erstellt beziehungsweise verändert werden sollen, muss das innerhalb eines JavaFX Application-Threads geschehen.

Über das StageStyle-Attribut kann festgelegt werden wie das Fenster aussehen soll (DECORATED z.B. sorgt dafür dass sich das Aussehen des Fensters am Betriebssystem orientiert).

(Vgl.: [12])



Abbildung 38: Stages unter verschiedenen Betriebssystemen (Quelle: [12])

4.4.9. Scene

Die JavaFX-Scene ist ein Container für alle sich im Scene-Graph befindenden grafischen Komponenten. Es kann als Bindeglied zwischen dem vom Betriebssystem bereitgestellten Fenster und dem Scene-Graph verstanden werden.

In einer Scene werden alle Leaf-Nodes angezeigt, da diese die grafischen Komponenten der GUI darstellen.

Scene-Objekte können nur über den JavaFX Application Thread erstellt und bearbeitet werden.

(Vgl.: [11])

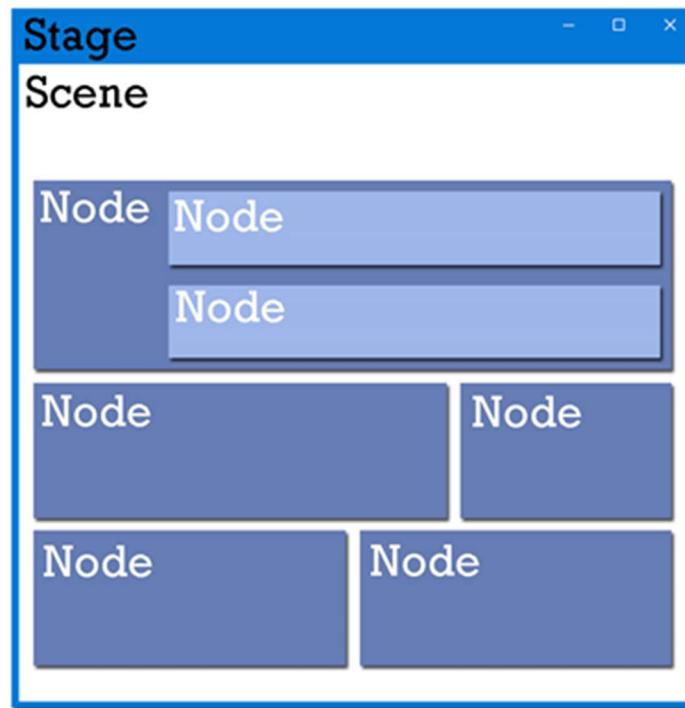


Abbildung 39: Stage und Scene

4.4.10. FXML

FXML ist eine XML-basierte Auszeichnungssprache für das Erstellen von Java Objektgraphen - im weiteren Sinne GUIs - und bietet somit eine gute Alternative zum prozeduralen Erstellen der UI-Elemente im Programmcode.

Es eignet sich besonders für JavaFX-Anwendungen, da sich die Struktur von XML-Dateien und die eines JavaFX Scene-Graphs sehr ähnlich sind.

(Vgl.: [8])

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.control.CheckBox?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.layout.Pane?>

<Pane maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-Infinity" minWidth="-Infinity" prefHeight="145.0" prefWidth="500.0"
       stylesheets="@../WorkflowGenerator.css" xmlns="http://javafx.com/javafx/8.0.65" xmlns:fx="http://javafx.com/fxml/1">
    <children>
        <Button fx:id="btnCloseApplication" layoutX="290.0" layoutY="100.0" maxWidth="294.0" mnemonicParsing="false"
               onAction="#btnCloseApplicationClicked" prefWidth="294.0" text="Close Application" />
        <Button fx:id="btnCancel" layoutX="30.0" layoutY="100.0" maxWidth="294.0" mnemonicParsing="false"
               onAction="#btnCancelClicked" prefWidth="294.0" text="Cancel" />
        <Label layoutX="30.0" layoutY="14.0" text="Do you want to close the Application &quot;Workflow Generator V1.0&quot;?" />
        <CheckBox fx:id="ckbxDontAskAgain" layoutX="30.0" layoutY="52.0" mnemonicParsing="false" text="Don't ask me again" />
    </children>
</Pane>
```

Abbildung 40: FXML

4.4.11. CSS

JavaFX CSS bietet die Möglichkeit, das Aussehen von GUI Elementen nach Belieben anzupassen. Funktionalität und Styling sind strikt getrennt.

Um den JavaFX CSS Code von herkömmlichen CSS zu unterscheiden, ist jeder Eigenschaft “-fx-” vorangestellt.

Wie schon erwähnt, wurde uns von der Knapp AG ein Styleguide zur Verfügung gestellt, was uns ein einheitliches GUI Design ermöglichte.

```
1  Button {  
2      -fx-background-color: #ACBBDB;  
3      -fx-text-fill: #000000;  
4      -fx-background-radius: 6 6 6 6;  
5      -fx-font-size: 15;  
6      -fx-pref-width: 180;  
7      -fx-pref-height: 30;  
8      -fx-max-width: 180;  
9      -fx-max-height: 30;  
10     -fx-min-width: 180;  
11     -fx-min-height: 30;  
12  }  
13  
14  Button:hover {  
15      -fx-border-color: black;  
16      -fx-border-radius: 6 6 6 6;  
17  }  
18  
19  Button:pressed {  
20      -fx-background-color: #FFEC00;  
21      -fx-border-color: black;  
22      -fx-border-radius: 6 6 6 6;  
23  }  
24  
25  Button:disabled {  
26      -fx-background-color: #CCCCCC;  
27  }  
28
```

Abbildung 41: CSS

4.4.12. RichTextFX

RichTextFX ist eine Softwareprojekt auf GitHub, welches von Tomas Mikula ins Leben gerufen wurde.

RichTextFX stellt ein Textfeld zur Verfügung, welches verschiedene Inhalte farblich voneinander unterscheidet, um so eine bessere Lesbarkeit für den Anwender zu gewährleisten. Der Text wird mit Regular-Expressions auf bestimmte Zeichenfolgen untersucht, die anschließend mit CSS entsprechend farblich hervorgehoben werden. In unserem Fall wurde RichTextFX für die Vorschau der zu generierenden XML Datei in der Benutzeroberfläche benötigt, die, wie von unserem Auftraggeber gewünscht, auch Code-Highlighting enthalten sollte. (Vgl.: [13])

Beispiel: Regular Expression für XML-Tags (ohne Attribute):

```
(?<ELEMENT>(</?\\h*>)(\\w+[-*\\w*]*)([^</?>]*)(\\h*/?>))" + "|(?<COMMENT><!--[^>]+-->)
```

```

01 <disjunction name="V01">
02   <station id="182" type="PlcShipping">
03     <start-station value="false"/>
04     <reader formatter="verbatim"/>
05     <physical-station-ref element="${concomm.PLCA1}" />
06     <paths>
07       <path element="${congoodsout.V01.station.ramps.error}" stInfo="0x0000"/>
08       <path element="${congoodsout.V01.station.ramps}" stInfo="0x0001"/>
09     </paths>
10     <configuration>
11       <announcement-storage-policy max-announcements="100"/>
12       <missing-announcement-settings send-data-request="true" timeout="5000"/>
13       <ramps>
14         <ramp number="1" successor="${congoodsout.sV01ramp}" />
15         <ramp number="2" successor="${congoodsout.sV01ramp}" />
16         <ramp number="3" successor="${congoodsout.sV01ramp}" />
17         <ramp number="4" successor="${congoodsout.sV01ramp}" />
18         <error successor="${congoodsout.sV01err}" />
19       </ramps>
20       <process-settings>
21         <successor-settings>
22           <case carrier-state="OK" fromAnnouncement="true" stInfo="0x0000"/>
23           <case carrier-state="Misread" stInfo="0x0000"/>
24             <default stInfo="0x0000"/>
25           </successor-settings>
26         </process-settings>
27       </configuration>
28     </station>
29   </disjunction>
30 
```

Abbildung 42: RichtextFX CodeArea

4.4.13. Warum JavaFX?

Dass das Programm mit JavaFX bzw. Java realisiert werden sollte, wurde von der Knapp AG bereits im Lastenheft festgelegt. Welche Vor- und Nachteile die Entscheidung nach sich zieht und welche Vor- bzw. Nachteile die Alternative, C# bzw. WPF bietet, wird im Folgenden beschrieben.

4.4.14. Vorteile von JavaFX im Vergleich zu WPF

Da JavaFX auf Java basiert, sind Anwendungen welche in JavaFX realisiert werden plattformunabhängig.

Dadurch, dass alle am Projekt Beteiligten bereits sehr gut mit JavaFX vertraut waren, bot sich Java an, wogegen WPF zwar ein Begriff war, aber die Beteiligten nicht ansatzweise so viel Erfahrung mit WPF gesammelt hatten, wie mit JavaFX.

JavaFX Anwendungen können mit jeder beliebigen Java Entwicklungsumgebung wie z.B. Eclipse oder IntelliJ geschrieben werden. Diese Entwicklungsumgebungen sind im Vergleich zur Alternative kostenlos. Visual Studio, welches benötigt wird um C# Anwendungen zu erstellen, ist nicht kostenlos. Es existiert zwar eine kostenlose Community Edition von Visual Studio, diese darf jedoch nur von Einzelpersonen und kleinen Unternehmen bis 5 Mitarbeiter kommerziell genutzt werden.

(Vgl.: [14])

Ein weiterer großer Vorteil ist, dass Java mit CSS gestyled werden kann. Dies war deswegen ein Vorteil, da der firmeninterne Styleguide “easy use” bereits teilweise mit CSS realisiert wurde.

4.4.15. Nachteile von JavaFX im Vergleich zu WPF

JavaFX bot für das Projekt nicht viele Nachteile im Vergleich zu WPF. Der einzige nennenswerte Nachteil war, dass die Erstellung von GUI-Komponenten mittels FXML nicht nativ in Java Entwicklungsumgebungen eingebunden ist. Es ist möglich, diese Dateien in z.B.

Eclipse textbasiert zu erstellen. Für die Erstellung von JavaFX-GUIs existiert ein externes Programm, der JavaFX Scene Builder.

Problematisch ist dies aus dem Grund, dass die FXML-Dateien von Hand mittels Pfaden in das Projekt integriert werden muss. Dies führte innerhalb unseres Projektes mehrmals zu Problemen.

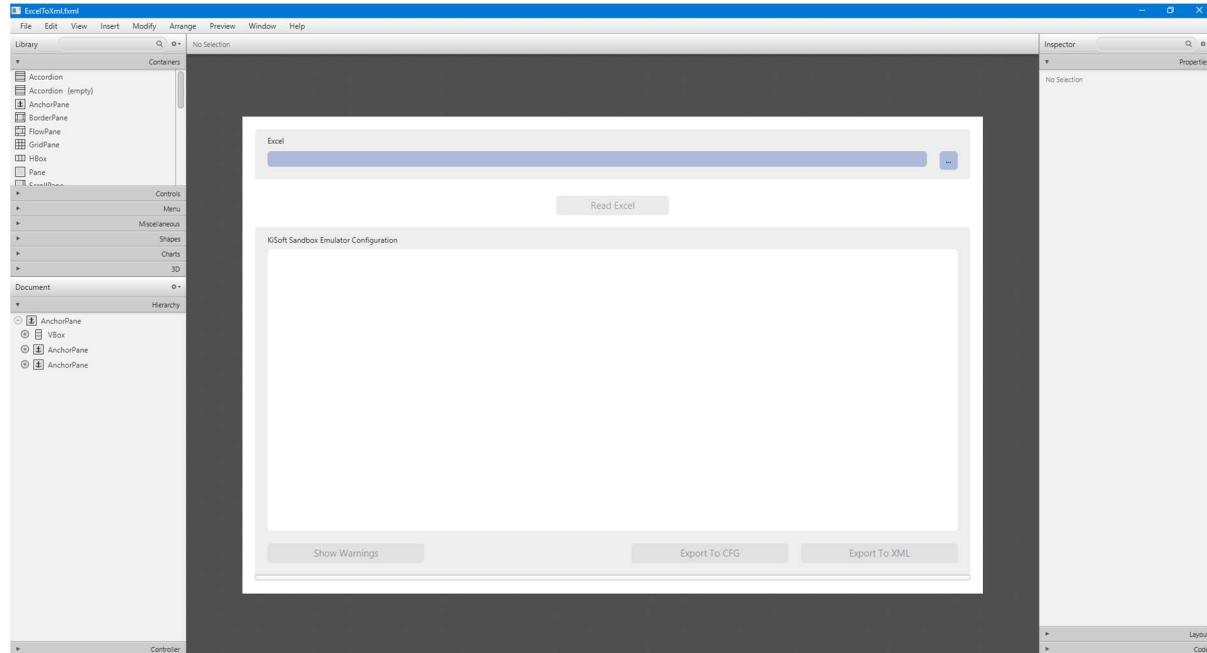


Abbildung 43: JavaFX SceneBuilder

In WPF hingegen können die GUI Komponenten (XAML) innerhalb der Entwicklungsumgebung (Visual Studio) erstellt werden.

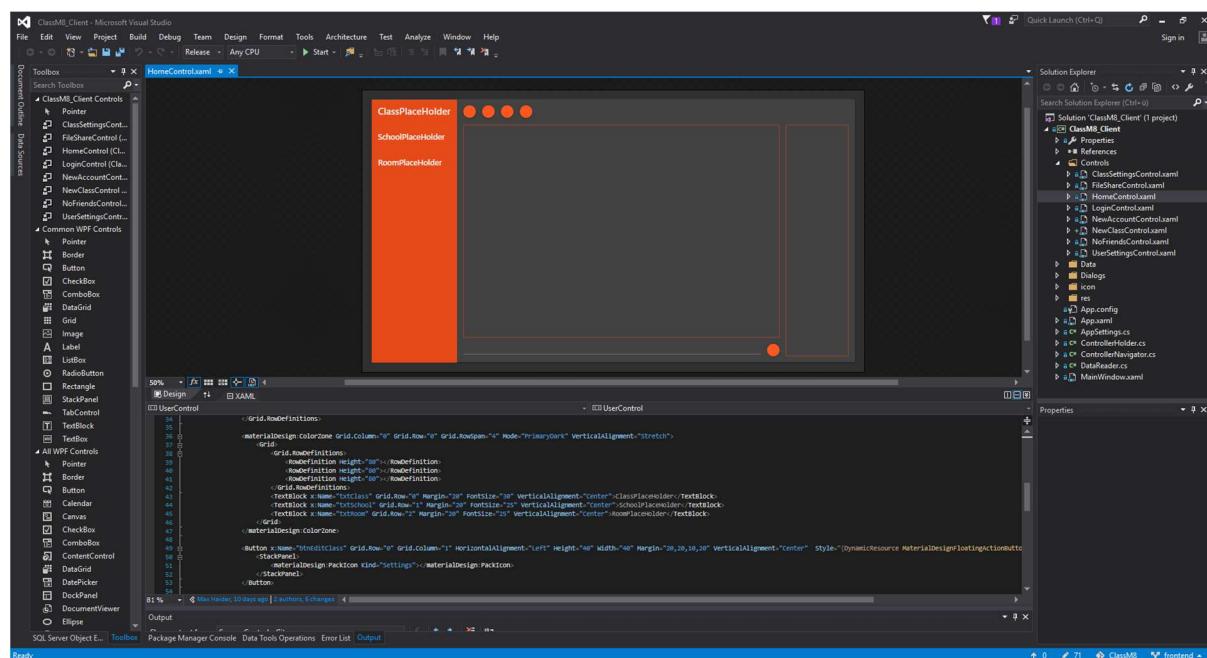


Abbildung 44: WPF GUI Builder

5. Excel-Reader

Der *Excel-Reader* dient dem Einlesen und Konvertieren der Excel-Dateien in das XML-Dateiformat.

Die dafür wichtigste Klasse ist die *ExcelReader*-Klasse. Die eingelesenen Daten werden im *WarehouseModelManager* gespeichert.

5.1. ParseExcel

Zum Konvertieren wird die *parseExcel*-Methode der *ExcelReader*-Klasse aufgerufen. Dieser muss man das zu konvertierende Excel-Datei und die Namen der Excel-Tabellen, die zu konvertieren sind, innerhalb einer *HashMap* mitgegeben.

Diese Excel-Tabellen tragen bei der Knapp AG einheitlich die Namen

- *WCSPLC*
- *HARDWARE*
- *OSRPLC*

Sie halfen auch bei der Zuordnung der Ordner, in welche die generierten XML-Dateien gespeichert werden.

In der *parseExcel*-Methode wird die Excel-Datei auf ihre Existenz überprüft.

Nach der Überprüfung zum Status der Excel-Datei wird jede einzeln angegebene Excel-Tabelle eingelesen.

Mithilfe der *ExcelReaderValidator*-Klasse wird überprüft, ob die Excel-Tabelle gültig ist.

Anschließend wird für jede einzelne Reihe der Excel-Tabelle mit Hilfe des *ExcelReaderValidators* überprüft, ob ein Layout, eine Station, ein *Stationpath*, ein Workflow, oder ein Lift eingelesen werden soll. Abhängig davon müssen verschiedene Methoden zum Einlesen der Daten verwendet werden.

Sollte eine Station eingelesen werden müssen, wird mithilfe der Namen der Excel-Tabellen überprüft, ob eine *Hardware-Station*, eine *OSRPLC-Station* oder eine *WCSPLC-Station* eingelesen werden muss. Abhängig davon müssen wieder verschiedene Methoden zum Einlesen verwendet werden, da die Stationen unterschiedlich aufgebaut sind.

Diese Methoden werden von der Klasse *ExcelParser* zur Verfügung gestellt. Beim Erstellen der einzelnen Stationen werden auch die Gateways erstellt und als Attribute der Stationen gespeichert.

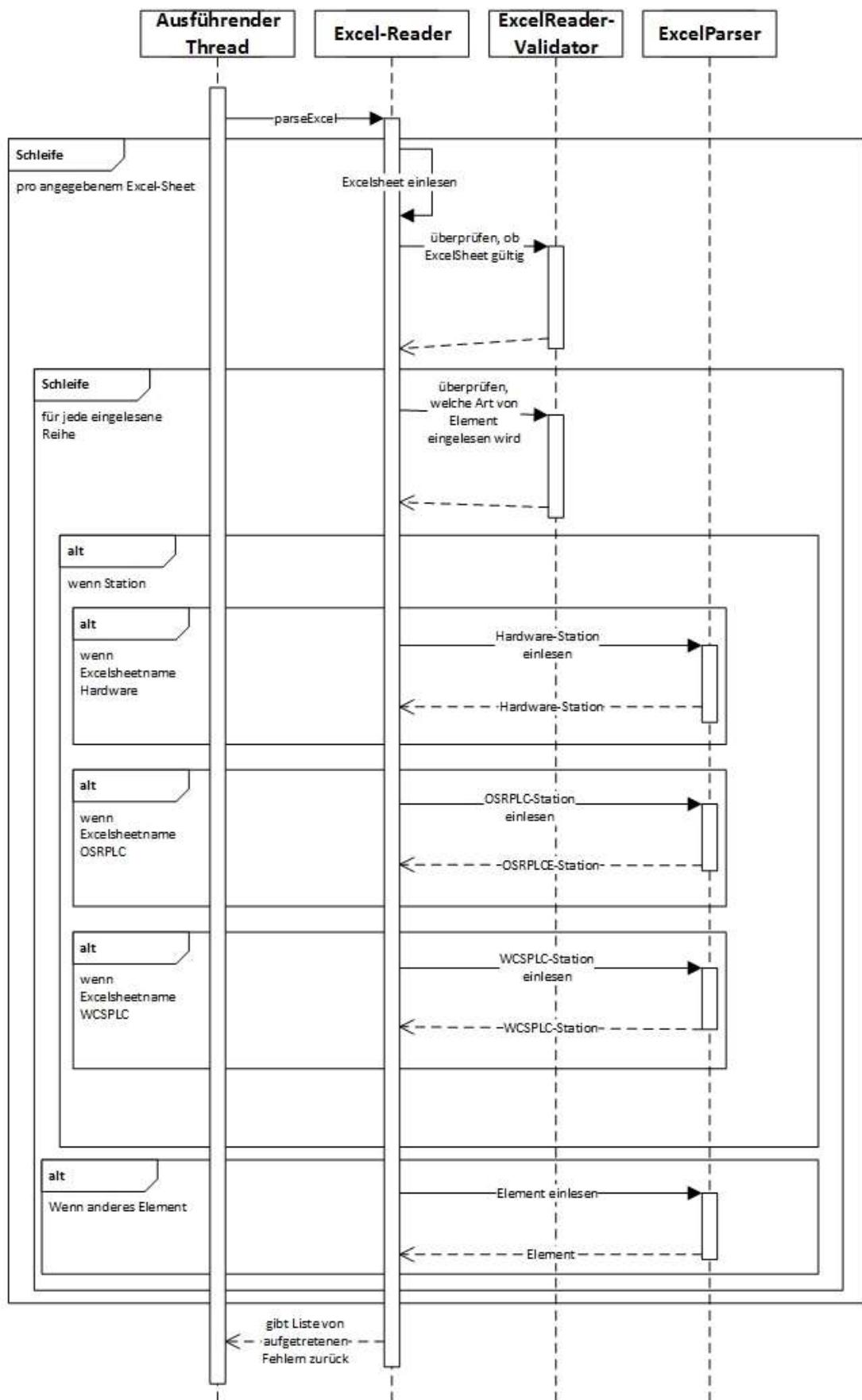


Abbildung 45 Vereinfachtes Ablaufdiagramm des Einlesens einer Excel-Datei

5.1.1. Einlesen der Stationen

Die Excel-Tabellen werden mithilfe der Apache POI API eingelesen (siehe „Apache POI“).

Dabei hatte jede Station ein so genanntes Gateway, welche je nach Typ unterschieden werden musste.

Die verschiedenen Gateway-Varianten waren

- *HardwarePhysicalGateway*
- *OsrPlcGateway*
- *WcsPlcGateway*

Nach dem internen Speichern der Workflows war es notwendig, diese in das gewünschte XML-Format zu übertragen. Dazu wird die Instanz des *WarehouseModelManagers*, der die Daten beinhaltet, an den sogenannten *SimulatorXMLGenerator* übergeben (siehe „XML-Generierung“).

5.2. Apache POI

Apache POI ist eine Java API, die es ermöglicht, Microsoft Office-Dateien mithilfe von Java Programmen zu öffnen, zu modifizieren und anzuzeigen.

Apache POI unterliegt der Open Source Lizenierung und wird von der Apache Software Foundation herausgegeben.

(Vgl.: [15])

Während der Erstellung des *KSB-Workflow-Generators* wurde Apache POI dazu verwendet, die zu konvertierenden Excel-Dateien einzulesen.

5.2.1. Öffnen von Workbooks

Workbook ist eine Klasse von Apache POI und bezieht sich auf Dateien vom Typ Excel. Dazu zählen auch .xlsx-Dateien.

Bevor man Workbooks lesen oder ändern will, muss man sie zuerst öffnen. Dies ist durch eine einfache Mitgabe einer Datei im Konstruktor eines Workbooks möglich oder der Mitgabe zur `create`-Funktion einer WorkbookFactory möglich. (Vgl.: [16])

5.2.2. Einlesen von Spreadsheets

Es gibt zwei Möglichkeiten, Tabellen (Spreadsheets) aus einem Workbook zu lesen.

Einerseits kann man mithilfe eines Index, andererseits mithilfe des Namens der Tabellen, die Tabelle aus einem Array lesen.

```
XSSFSheet spreadsheet = workbook.getSheetAt(0);
mySheet = myExcel.getSheet(sheetName);
```

Mithilfe eines Iterators ist es möglich, durch jede einzelne Zeile, Row genannt, einer Tabelle durchzugehen.

```
Iterator <Row> rowIterator = spreadsheet.iterator();
while (rowIterator.hasNext()) {
    row = (XSSFRow) rowIterator.next();
}
```

Jede Zeile, und damit jede Tabelle, besteht aus einzelnen Zellen, genannt Cells, welchen man Eigenschaften und Werte zuteilen kann (siehe „Cells“).

Die einzelnen Zellen bekommt man wieder durch Durchiterieren der Zellen einer Reihe, oder über einen Index. (Vgl.: [18])

```
row.getCell(0);
```

5.2.3. Cell

Cells spiegeln Zellen einer Excel-Tabelle wider. Sie können numerische, formel-basierte und Texteingaben als Wert besitzen. Die Art des Wertes wird durch das Attribut (Field) type spezifiziert. Diese sind durch das Enum CellType einfach abrufbar.

Es gibt folgende CellTypes:

- BLANK
- BOOLEAN
- ERROR
- FORMULA

- NUMERIC
- STRING

(Vgl.: [17])

5.2.4. FORMULA

Beim CellType “Formula” kann eine Formel angegeben werden. Zu diesen zählt beispielsweise die manuelle Eingabe der Summen-Funktion in Excel. Es können Formeln angegeben werden, die nur eine einzige Zelle betreffen, Formeln, die mehrere Zellen betreffen und Formeln, die von mehreren Zellen verwendet werden.

(Vgl.: [19], [20])

6. XML–Generierung

Die Konvertierung der intern gespeicherten Workflows ins XML-Format wird von der Klasse *SimulatorXMLGenerator* und ihrer Methode *createXMLDocumentsForWarehouseModel* übernommen. Dabei wird für jeden einzelnen Workflow ein eigenes DOM-Dokument (siehe „Java DOM“) erstellt.

Die eigentliche Konvertierung übernimmt die *generateWorkflowXMLDocument*-Funktion der *SimulatorFileFactory*-Klasse. Diese erstellt ein DOM-Dokument mithilfe eines DocumentBuilders.

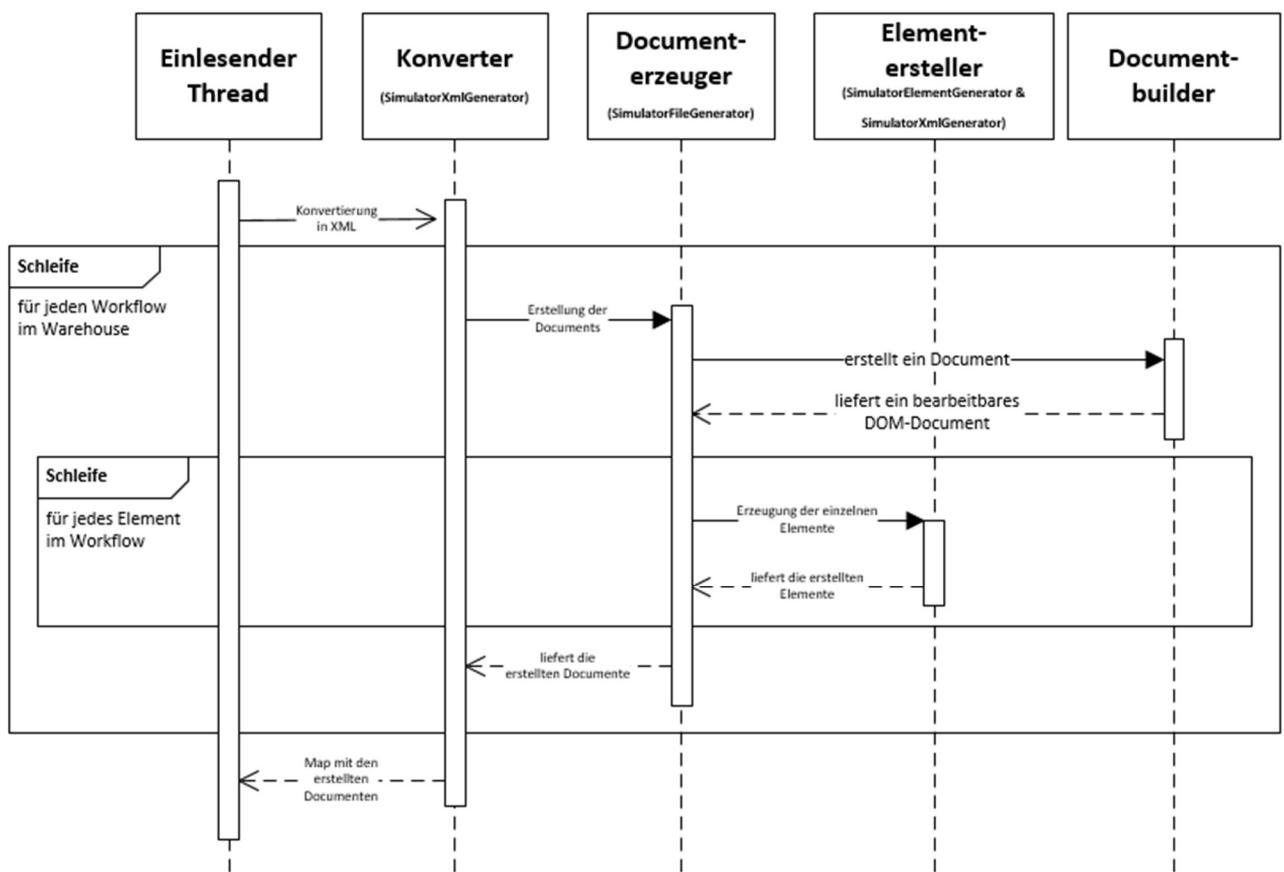


Abbildung 46 Vereinfachtes Ablaufdiagramm der Konvertierung einer eingelesenen Excel-Datei

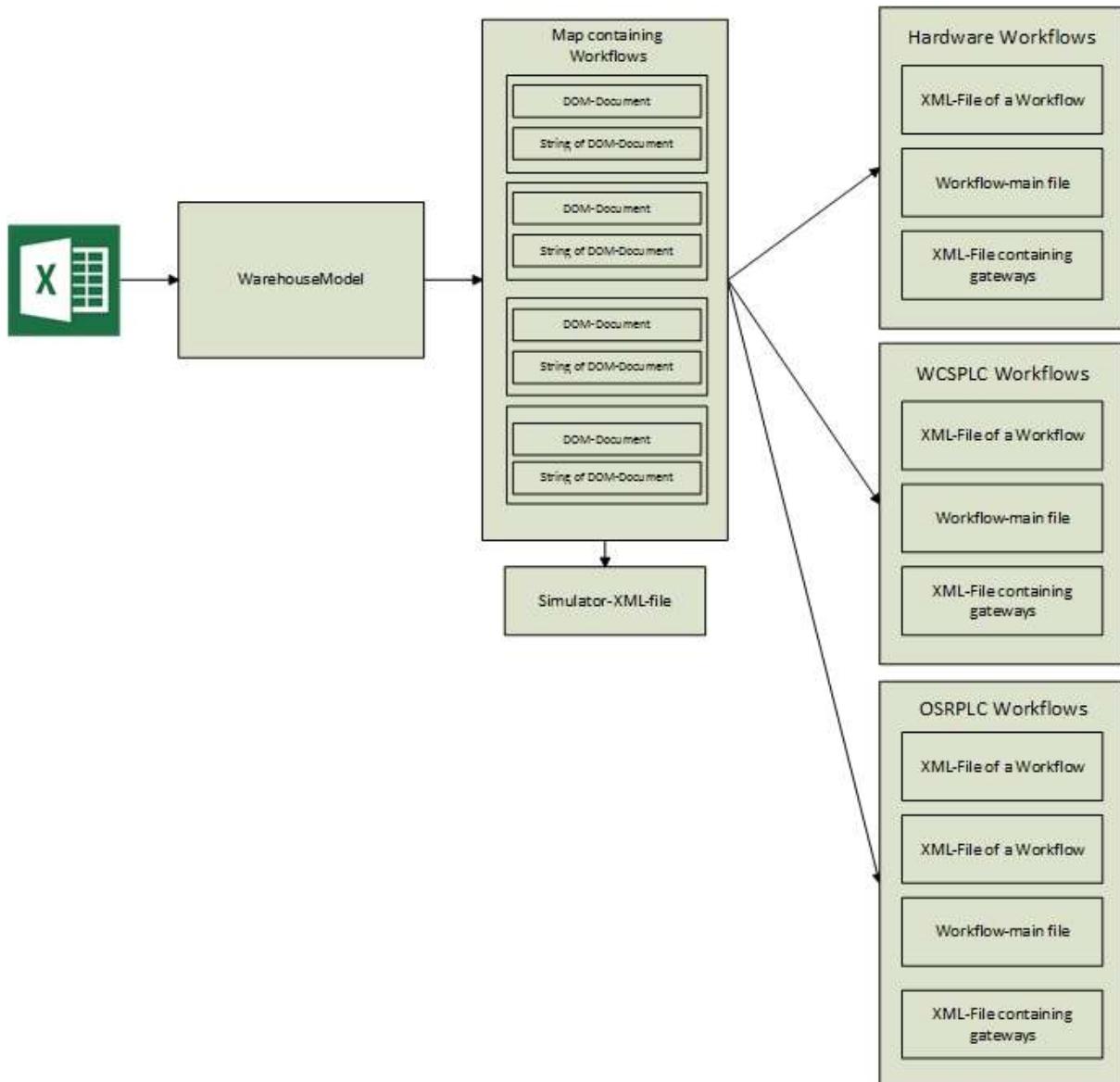


Abbildung 47 Lebenszyklus eines DOM-Dokuments (Quelle Excel-Logo: [35])

Diese DOM-Dokumente mussten in das String-Format konvertiert werden, um auf der GUI angezeigt werden zu können. Aufgrund einiger auftretender Bugs beim Konvertieren nur durch den so genannten LSSerializer, wurden die Dokumente mithilfe eines Transformers doppelt überprüft.

Beim Erstellen der eigentlichen XML-Dateien mussten zusätzlich zu den Stationen auch die Gateways in das XML-Format übertragen werden. Diese werden nicht auf der GUI angezeigt und werden beim Speichern in eigenen Dateien innerhalb der Ordner abgelegt.

Da die Dokumente für Benutzer immer noch nicht ausreichend lesbar waren, da Leerzeilen vor, beziehungsweise nach einigen Elementen fehlten, mussten diese auf Wunsch des Kunden manuell eingefügt werden.

```

<!--StationNode[SK01]-->
<disjunction name="SK01">
    <station id="81" type="PlcDisjunction">
        <start-station value="false"/>
        <reader formatter="verbatim"/>
        <physical-station-ref element="${concomm.PLCA1}"/>
        <paths>
            <path element="${conwarehouse.SK01_t_K01}" stInfo="0x0000"/>
            <path element="${conwarehouse.SK01_t_sSK01}" stInfo="0x0100"/>
        </paths>
        <configuration>
            <announcement-storage-policy max-announcements="100"/>
            <missing-announcement-settings send-data-request="true" timeout="WaitForever"/>
            <process-settings>
                <successor-settings>
                    <case carrier-state="OK" fromAnnouncement="true" stInfo="0x0100"/>
                    <case carrier-state="Misread" stInfo="0x0100"/>
                    <default stInfo="0x0100"/>
                </successor-settings>
            </process-settings>
        </configuration>
    </station>
</disjunction>

<transport bufferSize="5" name="SK01_t_K01" successor="${conwarehouse.K01}"/>
<transport bufferSize="5" name="SK01_t_sSK01" successor="${conwarehouse.sSK01}"/>

```

Abbildung 48 Beispiel einer Station mit manuell eingefügten Leerzeilen

```

<!--StationNode[SK01]-->
<disjunction name="SK01">
    <station id="81" type="PlcDisjunction">
        <start-station value="false"/>
        <reader formatter="verbatim"/>
        <physical-station-ref element="${concomm.PLCA1}"/>
        <paths>
            <path element="${conwarehouse.SK01_t_K01}" stInfo="0x0000"/>
            <path element="${conwarehouse.SK01_t_sSK01}" stInfo="0x0100"/>
        </paths>
        <configuration>
            <announcement-storage-policy max-announcements="100"/>
            <missing-announcement-settings send-data-request="true" timeout="WaitForever"/>
            <process-settings>
                <successor-settings>
                    <case carrier-state="OK" fromAnnouncement="true" stInfo="0x0100"/>
                    <case carrier-state="Misread" stInfo="0x0100"/>
                    <default stInfo="0x0100"/>
                </successor-settings>
            </process-settings>
        </configuration>
    </station>
</disjunction>
<transport bufferSize="5" name="SK01_t_K01" successor="${conwarehouse.K01}"/>
<transport bufferSize="5" name="SK01_t_sSK01" successor="${conwarehouse.sSK01}"/>

```

Abbildung 49 Beispiel einer Station ohne manuell eingefügten Leerzeilen

Auch der Copyright-Text der Firma Knapp AG musste manuell eingefügt werden, da der Transformer Kommentare unterhalb des Root-Elements versetzte, und nicht oberhalb, wie es vom Kunden gewünscht war.

Beim Speichern eines *WarehouseModels* wird für jeden Workflow eine XML-Datei erstellt.

Diese werden abhängig von ihren Typen (*WCSPLC*, *OSRPLC*, *Hardware*) in verschiedenen Ordnern gespeichert. Die Namen dieser Ordner und die Zuteilung dieser zu den Typen der Workflows wird aus der *workflow-generator.xml* Datei ausgelesen.

Name
workflow-con-comm.xml
workflow-con-goodsin.xml
workflow-con-goodsout.xml
workflow-con-ylogcontainerloop.xml
workflow-main.xml

Abbildung 50 Beispiel von generierten Workflow-Dateien

```
<areas>
  <area exp-directory="workflow-conveyor" name="WCSPLC" short-name="con"/>
  <area exp-directory="workflow-conveyor" name="HARDWARE" short-name="con"/>
  <area exp-directory="workflow-osr" name="OSRPLC" short-name="osr"/>
</areas>
```

Abbildung 51 Teil der workflow-generator Datei, die für die Zuteilung der Workflows in die Ordner zuständig ist

Zusätzlich musste jeder Workflow eines Ordners in einer *workflow-main*-Datei referenziert werden.

```
<simulator xmlns="http://www.knapp.com/sandbox/simulator"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.knapp.com/sandbox/simulator file://sandbox-simulator.xsd">

  <property name="interval" value="500"/>
  <workflow name="conmain">

    <default-driver>
      <interval-driver interval="${interval}"/>
    </default-driver>

  </workflow>

  <import file="workflow-con-comm.xml"/>

  <import file="workflow-con-warehouse.xml"/>

</simulator>
```

Abbildung 52 Beispiel einer main-workflow.xml Datei

Die *workflow-main*-Dateien der verschiedenen Ordner werden wiederum in der *simulator*-Datei referenziert, damit der Simulator diese auch findet.

```
<simulator xmlns="http://www.knapp.com/sandbox/simulator"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.knapp.com/sandbox/simulator file://sandbox-simulator.xsd">
  ...
  <property name="ROOT" value="/users/sandbox/cfg/simulator/" />
  <property name="SANDBOX_CON_CFG_DIR" value="${ROOT+'warehouse-conveyor/'}"/>
  <property name="SANDBOX_OSR_CFG_DIR" value="${ROOT+'warehouse-osr/'}"/>
  <property name="SANDBOX_DFT_CFG_DIR" value="${ROOT+'warehouse-dft/'}"/>
  <property name="SANDBOX_PROP_DIR" value="${ROOT+'properties/'}/>
  <property name="SANDBOX_SCRIPT_DIR" value="${ROOT+'scripts/'}/>
  <properties file="${SANDBOX_PROP_DIR+'station-controlinfo.properties'}" format="properties" prefix="sci"/>
  <properties file="${SANDBOX_PROP_DIR+'tote-types.properties'}" format="properties" prefix="tote_type"/>
  <property name="interval" value="500"/>
  <import file="workflow-conveyor/workflow-main.xml"/>
</simulator>
```

Abbildung 53 Beispiel einer simulator.xml Datei

Zusätzlich werden für den Simulator sechs, sich nicht verändernde, Property Dateien erstellt. Auf Wunsch des Product Owners werden diese Dateien nicht einfach kopiert, sondern jedes Mal neu erstellt.

Diese Property Dateien sind

- JavaPolicy
- *LoggingModulesProperties*
- *StationControlInfoProperties*
- *ToteTypesProperties*
- *SimulatorProperties*
- Log4J

Diese werden bis auf die *SimulatorProperties*-Datei alle in einem eigenen „properties“ Ordner gespeichert.

6.1. Java DOM

Das Document Object Model (DOM) ist eine Schnittstelle für XML oder HTML-Dokumente. Es beschreibt die logische Struktur von Dokumenten und wie diese bearbeitet werden. DOM wird dazu genutzt, um XML-Dateien handzuhaben.

DOM repräsentiert Dokumente in einer Baum-Struktur.

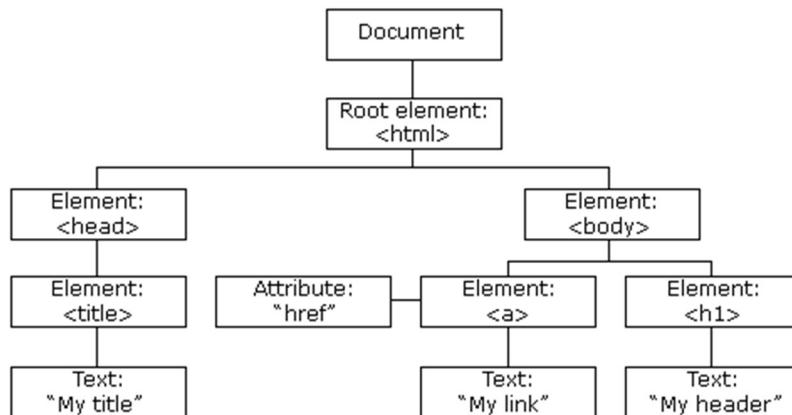


Abbildung 54 Visualisierung einer Baum-Struktur nach dem Document Object Model (Quelle: [30])

Mit Hilfe von DOM ist es möglich, XML-Dokumente zu erzeugen, durch ihre Struktur zu navigieren, Elemente hinzuzufügen, zu bearbeiten und zu löschen.

6.1.1. Elemente in DOM

Im *KSB-Workflow-Generator* stellten sich folgende Interfaces als besonders wichtig heraus.

6.1.1.1. Node

Node ist der Haupt-Datentyp eines jeden der folgenden Interfaces. Da das Node Interface Methoden zum Hinzufügen und Bearbeiten von Unterelementen (Child-Elements) zur Verfügung stellt, stellen auch Klassen beziehungsweise Interfaces, die keine "Kinderelemente" annehmen, wie beispielsweise das "Comment"-Interface, diese Methoden zur Verfügung. Sollte versucht werden, eine solche Methode zu benutzen, wird eine "DOM-Exception" ausgelöst. (Vgl.: [23])

6.1.1.2. Document

Ein DOM-Document spiegelt ein komplettes HTML- oder XML-Dokument wider. Es besteht aus Elementen. Um ein Document zu erstellen, ist ein DocumentBuilder zu benützen. (Vgl.: [24])

6.1.1.3. Element

Elemente sind Teile eines ganzen Dokuments. Elemente können wieder Elemente beinhalten. Das äußerste Element eines Dokuments wird als Root-Element bezeichnet (siehe „XML-Root-Element“). Elementen können Attribute zugewiesen werden. (Vgl.: [26])

6.1.1.4. Comment

Mithilfe von Comments können in den DOM-Baum Kommentare eingefügt werden. Diese starten im DOM-Dokument mit '<!--' und enden mit '-->'. (Vgl.: [25])

6.1.1.5. Attr

Attr ist das Interface für Attribute. Elemente können aus Attributen bestehen. Attribut-Objekte erben zwar vom Node-Interface, sind aber nicht Teil des Dokumenten-Baums. Mit ihnen könne Elemente genauer beschrieben werden. (Vgl.: [27])

6.1.1.6. Text

Das Text-Interface spiegelt den eigentlichen Inhalt eines Elementes oder Attributes wider. (Vgl.: [28])

6.1.2. Suche von Elementen im DOM

Um gezielt Elemente aus einem DOM-Baum zu suchen ist es notwendig, diesen in eine Liste zu speichern und durch diese zu iterieren. Durch Vergleichen der Properties der einzelnen Nodes in der Liste mit den Eigenschaften des gesuchten Elements kann das gesuchte Element gefunden werden.

Beispiel:

```
final NodeList settings = getXMLObjects(xmlConfigFile, "process-settings");

for (int y = 0; y < settings.getLength(); y++) {

    final Node settingNode = settings.item(y);
    if (settingNode.getNodeType() == Node.ELEMENT_NODE) {
        exportDirectory = ((Element)settingNode)
            .getAttribute("xml-export-directory");
    }
}
```

6.1.3. Konvertierung

6.1.3.1. Umwandlung von DOM in STRING

Es gibt zwei Möglichkeiten zur Konvertierung eines DOM-Objektes in einen String. Im *KSB-Workflow-Generator* werden diese Strings zum Anzeigen der Stationen auf der GUI, aber auch zum Speichern der XML-Dokumente als Dateien benötigt.

- Transformer
- LSSerializer

6.1.3.2. Transformer

Instanzen der Transformer-Klasse ermöglichen es, einen XML-Source-Baum in ein Ergebnis des Types “Result” umzuwandeln.

6.1.3.3. Source-Baum

Im Fall des *KSB-Workflow-Generators* ist dieser XML-Source-Baum immer ein Dokument, welches in ein so genanntes DOMSource-Objekt umgewandelt wird.

6.1.3.4. Result

Das Ergebnis kann mithilfe eines StreamReaders gelesen und in einen String umgewandelt werden.

6.1.3.5. OutputProperties

Der Transformer-Instanz können Eigenschaften zur Umwandlung zugewiesen werden. Dies ist mithilfe der setOutputProperties-Funktion möglich.

Zu diesen Eigenschaften zählen:

- ENCODING
- INDENT
- METHOD
- OMIT_XML_DECLARATION

(Vgl.: [29])

ENCODING	Mit dem ENCODING-Parameter wird sichergestellt, dass die UTF-8-Encodierung beim Transformieren verwendet wird.
INDENT	Der INDENT-Parameter sorgt dafür, dass tiefer liegende Elemente weiter eingerückt werden.
METHOD	Mit diesem Parameter wird sichergestellt, dass die Umwandlung von DOM in XML erfolgt, und nicht von DOM in HTML.
OMIT_XML_DECLARATION	Mithilfe von OMIT_XML_DECLARATION wird sichergestellt, dass die XML-Deklaration eingefügt wird.

6.1.3.6. LsSerializer

Zur Konvertierung wird ein DOMImplementationLS-Objekt benötigt, welches dazu verwendet wird, einen LSserializer und LSOoutput herzustellen. Dem LSOoutputObjekt muss ein StringWriter mitgegeben werden, und der write-Funktion des LSserializers das LSOoutput-Objekt und das zu konvertierende Dokument.

Nun beinhaltet der StringWriter das Dom-Dokument und dieses ist über eine toString-Methode abrufbar.

```
DOMImplementationLS impls = (DOMImplementationLS) DOMImplementationRegistry.newInstance().getDOMImplementation("LS");
Writer stringWriter = new StringWriter();
LSSerializer domWriter = impls.createLSSerializer();
LSOutput lsOutput = impls.createLSOutput();
lsOutput.setEncoding("UTF-8");
lsOutput.setCharacterStream(stringWriter);
domWriter.write(node, lsOutput);
s = stringWriter.toString();
```

6.1.3.7. Konvertierung von String zu DOM

Um einen String in ein DOM-Dokument zu konvertieren, ist es nötig, eine DocumentBuilderFactory herzustellen. Mit dieser ist es möglich, ein DocumentBuilder-Objekt herzustellen.

Um einen String zu konvertieren, ist die parse-Funktion dieses DocumentBuilders voraussetzt. Diese nimmt allerdings nur ein Objekt des Typs InputSource an. Um in diese zu schreiben ist die Hilfe eines StringReaders in Anspruch zu nehmen, welcher den String in einen CharacterStream umwandelt.

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
DocumentBuilder db = dbf.newDocumentBuilder();
InputSource is = new InputSource();
is.setCharacterStream(new StringReader(s));
doc = db.parse(is);
```

6.1.3.8. Einlesen einer Datei in ein DOM-Objekt

Es gibt 2 Möglichkeiten, Objekte aus einer schon vorhandenen XML-Struktur einzulesen. Dies wurde benötigt, um Einstellungen aus der Konfigurations-Datei zu erfassen und anzuwenden.

Diese beiden Möglichkeiten sind:

1. Sax Parser
2. DOM Parser

6.1.3.8.1. Sax Parser

Der Sax Parser ist eine event-basierte API. Er erzeugt keine interne Struktur, sondern gibt dem Programm lediglich auftretende Teile als Events zurück. Er gibt der Anwendung also immer nur Teile eines ganzen Dokuments.

Da man DOM-Dokumente durchsuchen und verändern musste, war der SaxParser für das Einlesen der Einstellungen ungeeignet.

6.1.3.8.2. DOM Parser

Der DOM Parser liest eine Datei ein und wandelt es in DOM-Dokumente um. Somit wird eine DOM -Baumstruktur erstellt, welche aus einem Dokument und mehreren Nodes besteht.

Daher ist der DOM Parser deutlich aufwändiger, aber auch langsamer, als der Sax Parser. Des Weiteren muss der DOM Parser immer ganze Dokumente einlesen, da ansonsten Teile der Baumstruktur verloren gehen könnten. Auf Grund dessen ist er für kleine XML-Strukturen besser geeignet als für größere.

(Vgl.: [31], [32], [33])

Vorteile DOM	Vorteile Sax
<ul style="list-style-type: none">• Konvertiert das ganze Dokument.• Das Ergebnis der Konvertierung befindet sich in einer Baumstruktur.	<ul style="list-style-type: none">• Konvertiert solange, bis ihm gesagt wird, dass er aufhören soll.• Es ist möglich, nur kleine Teile eines großen Dokumentes zu lesen.

6.1.3.9. Validierung von DOM-Objekten

Mithilfe von XSD ist es möglich, DOM-Objekte auf ihre Gültigkeit und Korrektheit zu überprüfen (siehe „XSD-Validierung“).

7. Validierung

7.1. XSD

Die XML-Schema-Definition ist eine Empfehlung vom World-Wide-Web-Consortium (W3C).

Mithilfe von XSD ist es möglich, den Aufbau und die Struktur eines Elementes eines XML-Dokumentes auf Übereinstimmung und Korrektheit zu validieren.

Die Überprüfung von XML-Strukturen mithilfe von XSD wurde während der Arbeit notwendig, um Fehler bei der Konvertierung von Excel in XML schnell zu bemerken.

Obwohl schon eine Datei mit einem fertigen XML-Schema vorhanden war, war es notwendig, sich mit XSD zu beschäftigen, da während der Entwicklung einige Einstellungen verändert beziehungsweise hinzugefügt werden mussten.

(Vgl.: [34], [38])

7.1.1. Namespace

Mithilfe von Namespaces können XSD-Schemen auf mehrere Dateien verteilt werden. Diese "Sub-Schemen" können in einem "Eltern-Schema" zusammengefasst und ihre Elemente weiterverwendet werden.

Um Konflikte zwischen mehreren Kind-Schemen zu vermeiden, können Präfixes verwendet werden.

Beispiel eines Elementes mit Präfix:

```
<xs:attribute>
```

W3C bietet einen Namespace an, um die Einhaltung des XSD-Standards zu vereinfachen.

(Vgl.: [37])

7.1.2. Aufbau

Der Aufbau einer XSD-Datei entspricht der einer XML-Datei. Das bedeutet, dass Elemente in so genannten Tags gespeichert werden. Diese Tags können mehrere Attribute beinhalten, die die Eigenschaften der Elemente beschreibt. (Vgl.: [34])

7.1.2.1. Elemente

7.1.2.1.1. Complex Type

Complex Type-Elemente sollen helfen, die Strukturierung einer XSD-Datei zu vereinfachen. Sie dienen quasi als Container für weitere Elemente, wodurch es möglich wird, dass Kind-Elemente definiert werden können. (Vgl.: [34])

7.1.2.1.2. Global Types

Complex Types können global definiert werden. Das heißt, ein Complex Type kann mehrmals wiederverwendet werden. Dazu muss diesem ein einzigartiger Name zugewiesen werden und er muss unabhängig von einem Element definiert werden.

Der global definierte Complex Type kann mithilfe des Attributes Type verwendet werden. (Vgl.: [34])

7.1.2.1.3. Compositors

Compositors regeln das Auftreten von Elementen. (Vgl.: [34])

7.1.2.1.4. Sequence

Die Kind-Elemente eines Sequence-Elementes müssen in genau der gleichen Reihenfolge in der XML-Datei auftreten, wie sie in der XSD-Datei angegeben wurden. (Vgl.: [34])

7.1.2.1.5. Choice

Bei Kind-Elementen eines Choice-Elementes ist die Reihenfolge ihres Auftretens nicht von Belang. (Vgl.: [34])

```

<xs:element name="Element">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Unterelement1" type="xs:string" />
      <xs:element name="Unterelement2" type="xs:string" />
      <xs:element name="Unterelement3" type="xs:string" />
    </xs:sequence>
  </xs:complexType>
</xs:element>

```

7.1.2.1.6. Attribute

Attribute-Elemente dienen der Definition von Attributen innerhalb eines Elementes in XML. Den Attributen können in XSD Attribute wie Name und Typ zugewiesen werden. (Vgl.: [34])

7.1.2.1.7. Use

Mithilfe des Attributes „use“ kann angegeben werden, ob ein Attribut angegeben werden muss, oder nicht. Dies geschieht mithilfe der Werte Required oder Optional. Per Default ist use auf den Wert Optional gestellt. (Vgl.: [34])

min-Occurs	<ul style="list-style-type: none"> Min-Occurs gibt die Mindestanzahl, wie oft ein Element vorkommen darf, an. Es darf nicht negativ sein.
max-Occurs	<ul style="list-style-type: none"> Max-Occurs gibt die maximale Anzahl an erlaubten Elementen an. Es darf nicht negativ sein und erlaubt zusätzlich den String „unbounded“ als Wert. Sollte dieser Wert als „unbounded“ angegeben werden, so ist dies mit keiner Obergrenze an Elementen gleichzusetzen.

Beispiele:

XSD-Schema Beispiel	Passendes XML-Element
<pre><xs:element name="Element"> <xs:complexType> <xs:attribute name="ElementID" type="xs:int" /> </xs:complexType> </xs:element></pre>	<pre><Element ElementID="3" /> ... - or no attribute - <Element /></pre>
<pre><xs:element name="Element"> <xs:complexType> <xs:attribute name="ElementID" type="xs:int" use="optional" /> </xs:complexType> </xs:element></pre>	<pre><Element ElementID="5" /> ... - or no attribute - <Element /></pre>
<pre><xs:element name="Element"> <xs:complexType> <xs:attribute name="ElementID" type="xs:int" use="required" /> </xs:complexType> </xs:element></pre>	<pre><Element ElementID="7" /></pre>

7.1.3. Attribute

7.1.3.1. Name

Das Attribut „name“ muss in jedem Element vorhanden sein. Es definiert den Namen jenes Elementes, welches an der Stelle des Elementes stehen soll. (Vgl.: [34])

Beispiel:

XSD-Schema Beispiel	Passendes XML-Element
<code><xs:element name="XML-Element" /></code>	<code><XML-Element></XML-Element></code>

7.1.3.2. Type

Das „Type“-Attribut gibt an, welchen Typ das Element, gemeint ist der Wert innerhalb der Spitzklammern, einnehmen soll. (Vgl.: [34])

Beispiele:

XSD-Schema Beispiel	Passendes XML-Element
<code><xs:element name="Element" type="xs:string" /></code>	<code><Element> ElementWert </Element></code>
<code><xs:element name="ElementID" type="xs:int"></xs:element></code>	<code><ElementID>1</ElementID></code>

7.1.3.3. Data Type

XSD unterstützt primitive sowie erweiterte Datentypen.

Zu den primitiven zählen

- string
- boolean
- decimal
- float

(Vgl.: [34])

7.1.3.4. Kardinalität

Mithilfe der Attribute "min-Occurs" und "max-Occurs" kann die Kardinalität, also die erlaubte Anzahl eines Elementes bestimmt werden. Die Default-Werte von min-Occurs und max-Occurs beträgt bei beiden 1. Jedes Element darf also nur genau einmal vorkommen, sollte keines der beiden Attribute angegeben werden. (Vgl.: [34])

min-Occurs	<ul style="list-style-type: none"> Min-Occurs gibt die Mindestanzahl, die ein Element vorkommen darf an. Es darf nicht negativ sein.
max-Occurs	<ul style="list-style-type: none"> Max-Occurs gibt die maximale Anzahl an erlaubten Elementen an. Es darf nicht negativ sein und erlaubt zusätzlich den String "unbounded" als Wert. Sollte dieser Wert als "unbounded" angegeben werden, so ist dies mit keiner Obergrenze an Elementen gleichzusetzen.

Beispiel:

XSD-Schema Beispiel	Beschreibung
<pre><xs:element name="Element" type="xs:integer" minOccurs="0" maxOccurs="unbounded"> </xs:element></pre>	Das Element kann beliebig oft vorkommen

<pre><xs:element name="Element" type="xs:string" minOccurs="2" maxOccurs="10"> </xs:element></pre>	Das Element muss mindestens zweimal, darf aber maximal zehnmal vorkommen.
--	--

7.1.4. Validierung einer XML-Datei mittels XSD in Java

XSD Validierung wird in Java seit Version 1.5 standardmäßig unterstützt. Dazu benötigt wird eine Instanz der Klasse Schema benötigt. Diese wird mit Hilfe einer SchemaFactory initialisiert. Bei der Initialisierung des Schema-Objektes muss die XSD-Datei an das Objekt gegeben werden.

Um eine Validierung zu ermöglichen ist es notwendig, ein DOM-Dokument (siehe „DOM“) in ein DOMSource-Objekt zu konvertieren.

Die Validierung selbst übernimmt ein mithilfe des Schema-Objektes erstelltes Validator-Objekt.

Sollte die Validierung fehlgeschlagen, also das DOM-Dokument nicht valide sein, wird eine SAXException ausgelöst.

Beispielcode:

```
try {
    SchemaFactory factory = SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
    Schema schema = factory.newSchema(new File(xsdPath));
    Validator validator = schema.newValidator();
    StreamSource ss = new StreamSource(new File(xmlPath));
    validator.validate(ss);
} catch (IOException | SAXException e) {
    Helper.Logger.error(e.getMessage(), e);
    throw new SimulatorGeneratorException(e.getMessage());
}
```

8. Lambda-Ausdrücke

Lambda-Ausdrücke sind ein Feature, das mit Java 8 integriert wurde. Sie sollen helfen, die Übersichtlichkeit von anonymen Klassen zu erhöhen. (Vgl.: [42]) Da es sich bei Anonymen Klassen oft um Funktionale Interfaces (siehe „Funktionale Interfaces“) handelt und darunter die Code-Qualität leidet, wird der Entwickler durch SonarQube (siehe „SonarQube“) darauf hingewiesen, diese durch einen Lambda-Ausdruck zu ersetzen. Des Weiteren wird auch die falsche Verwendung von SonarQube erkannt und bemängelt.

8.1. Was sind Lambda-Ausdrücke?

Lambda-Ausdrücke sind im Grunde genommen Methoden ohne Namen. (Vgl.: [42], [41])

Sie teilen sich mit Methoden zwei grundlegende Bausteine:

- Argument List
- Body

Die Argument List und der Body werden durch einen Pfeil voneinander getrennt. Dieser wird “Arrow Token” genannt und stellt lediglich eine Schnittstelle zwischen diesen beiden Teilen dar. Er hat keine weiteren Funktionen. (Vgl.: [42], [41])

8.1.1. Body

In den Body wird der auszuführende Code geschrieben.

Der Body kann sowohl ein einziger Befehl, aber auch ein ganzer Befehl-Block sein. (Vgl.: [42], [41])

8.1.2. Argument List

In der Argument-List werden die Variablen, die im Body verwendet werden deklariert.

Lambda-Ausdrücke akzeptieren 0,1 oder mehr Argumente. Argumente stellen die Parameter dar, die an die Lambda-Funktion mitgegeben werden. (Vgl.: [41], [42])

8.2. Rückgabetypen

Im Gegensatz zu üblichen Methoden müssen keine Rückgabetypen und kein Funktionsname angegeben werden. Auch das Hinzufügen einer *throws*-Klausel ist nicht möglich. Auch wenn sie nicht angegeben werden müssen, ist es möglich, Werte zurückzugeben, da die Typen automatisch vom Compiler erkannt und bestimmt werden. Dieses automatische Erkennen des Typs des Lambda-Ausdruckes nennt man *target typing*. (Vgl.: [41] Seite 17)

Beispiel für einen einfachen Lambda-Ausdruck:

Im folgenden Beispiel werden die Werte x und y mit Hilfe der “Argument List” an den Body weitergegeben. Dort werden sie dann addiert und das Ergebnis ausgegeben.

Argument List	Arrow Token	Body
(int x, int y)	->	{int z = x + y; System.out.println(z);}

8.3. Lambda-Ausdrücke oder anonyme Klassen?

Lambda-Ausdrücke werden “ad-hoc” implementiert, also immer dort, wo sie gerade gebraucht werden. In diesem Sinne sind sie den anonymen Klassen sehr ähnlich. Auch, dass sie üblicherweise als Parameter an eine Funktion übergeben werden, um die Funktionalität des Lambda-Ausdrucks bzw. anonymen Klasse innerhalb einer anderen zu verwenden, haben beide gemeinsam. Wird eine Funktionalität als Parameter an eine Funktion übergeben, wird das als “codeasdata” bezeichnet. Dadurch, dass bei anonymen Klassen ein Objekt gebildet und dieses an eine Funktion übergeben wird, ist dieser Begriff allerdings bei den anonymen Klassen nicht ganz zutreffend. Bei Lambdas hingegen wird keine ganze Klasse abgebildet, sondern nur eine einzige Methode. Dennoch wird die Lambda-Funktion vom Compiler zur Laufzeit instanziert. (Vgl.: [41] Seite 9ff)

8.4. Instanziierung während der Laufzeit

Die Instanziierung während der Laufzeit ist insofern ein Vorteil, da dadurch, aufgrund von Optimierungen seitens der Kompilierung, die Performance erhöht wird. So wird beispielsweise ein synthetischer Typ des Lambda-Ausdrucks, aber auch das Lambda-Objekt selbst, dynamisch und erst während der Laufzeit von der Java virtual machine erstellt.

Dadurch ist es möglich, die Erstellung dieses synthetischen Typs und des Objektes bis zum ersten Auftreten zu verhindern. Wird es nur definiert, aber nie benutzt, wird weder der Typ des Lambda-Ausdruckes, noch das Objekt erstellt. (Vgl.: [41] Seite 17f)

8.5. Vergleich zwischen Lambda-Ausdrücken und anonymen Klassen

8.5.1. Syntax

Anonyme Klasse	Lambda-Ausdruck
<pre>Runnable r = new Runnable(){ public void run(){ System.out.println ("count: " + count); } };</pre>	<pre>Runnable r = () -> { System.out.println ("count: " + cnt); };</pre>

8.5.2. Variablenbindung

Sowohl bei anonymen Klassen, als auch bei Lambda-Ausdrücken, müssen Variablen, die innerhalb von anonymen Klassen bzw. Lambda-Ausdrücken verwendet werden, final sein. Dies muss allerdings nicht explizit bei der Variablen Deklaration angegeben werden. Der Compiler erkennt automatisch, dass die Variable in einem Lambda-Ausdruck bzw. einer anonymen-Klasse verwendet wird und deklariert sie automatisch final. Bei anonymen Klassen war dies bis Java 8 nicht der Fall. (Vgl.: [41] Seite 22f)

Count wird nicht verändert	Count wird verändert
<pre>void method() { int count = 16; Runnable r = () -> { System.out.println ("count: " + count); }; Thread t = new Thread(r); t.start(); }</pre>	<pre>void method() { int count = 16; Runnable r = () -> { System.out.println ("count: " + count); }; Thread t = new Thread(r); t.start(); count++; }</pre>

8.5.3. Aufwand zur Laufzeit

Um anonyme Klassen zu erstellen, muss während der Laufzeit die Klasse geladen, Speicherplatz geschaffen, das Objekt initialisiert und es müssen zusätzliche, nicht statische, Methoden aufgerufen werden.

Bei Lambda-Ausdrücken muss zur Laufzeit das funktionale Interface konvertiert und eventuell aufgerufen werden. Die Zuweisung der Typen an den target type erfolgt schon beim Kompilieren und erfolgt aufgrund verschiedenster Optimierungen ressourcenschonend. (Vgl.: [41] Seite 25)

8.5.4. Sichtbarkeitsbereich (Scope) von Variablen

Bei Anonymen Klassen wird ein eigener Sichtbarkeitsbereich (Scope) angelegt, während sich der Lambda-Ausdruck an den ihn umgebenden Scope anpasst. Das bedeutet, dass man Variablen, die schon im sie umschließenden Scope definiert wurden, nicht noch einmal definieren kann. (Vgl.: [41] Seite 23f)

Anonyme Klasse	Lambda-Expression
<pre>void method(){ int count = 16; Runnable r = new Runnable(){ public void run() { int count = 0; //funktioniert System.out.println("cnt is: "+count); } System.out.println("cnt is: "+count); } }</pre>	<pre>void method(){ int cnt = 16; Runnable r = ()-> { int cnt = 0; System.out.println ("cnt is: "+cnt); }; }</pre>

8.6. Funktionale Interfaces

Wie bereits erwähnt, wird während der Laufzeit ein synthetischer Typ für den Lambda-Ausdruck erstellt, welcher einen Subtypen des target types darstellt. Der Compiler prüft schon beim Kompilieren den Kontext, in welchem der Lambda-Ausdruck definiert ist und passt den Typen an den Kontext an.

Java besitzt keine funktionalen Typen. Um Lambda-Ausdrücke in Java integrieren zu können, ohne funktionale Typen einzuführen, wurden so genannte functional interfaces verwendet. Diese sind Interfaces mit nur einer einzigen Methode. Einige solcher Interfaces gibt es schon seit der ersten Java-Version. Zu ihnen gehören die Interfaces Runnable, Iterable und auch Comparable.

Da sowohl functional interfaces, als auch Lambda-Ausdrücke, nur eine Methode benötigen, konvertiert der Compiler jeden Lambda-Ausdruck in einen passenden functional interface-Typ. Daher ist es möglich, functional interfaces durch Lambda-Ausdrücke zu ersetzen.

(Vgl.: [41] Seite 17-20)

Beispiel für Funktionale Interfaces

In diesem Beispiel wird mit Hilfe der Erstellung eines Threads veranschaulicht, wie Lambda-Ausdrücke dazu verwendet werden können, Interfaces, die nur eine Funktion benötigen, zu instanziieren.

Anonyme Klasse	Lambda
<pre>Thread t = new Thread (new Runnable(){ @Override public void run (){ //Some code };});</pre>	<pre>Thread t = new Thread(()->{ //Some Code });</pre>

8.7. Warum wurden Lambdas eingeführt?

Der Hauptgrund für die Einführung von Lambda-Ausdrücken war das Ziel, die Java-Sprache weiterzuentwickeln. Durch das originale API-Design von Java sind einige dringend benötigte Optimierungen unmöglich. (Vgl.: [41] Seite 24f)

8.7.1. Neue Collection-Funktionen

Während heutzutage nur ein paar wenige Prozessorkerne in Computern eingebaut werden, ist damit zu rechnen, dass sich die Anzahl dieser in den nächsten paar Jahren vervielfacht. Aufgrund dessen sollten einzelne Aufgaben auf mehrere Threads aufgeteilt werden. Zu aufzuteilenden Aufgaben zählt beispielsweise das Durchiterieren von Collections. Dieses wurde durch die Einführung von parallel bulk operations for collections ermöglicht. So muss beispielsweise bei einer Veränderung jedes einzelnen Elementes einer Collection, nicht mehr nur eine einzige Schleife, die von einem einzelnen Thread ausgeführt wird, durch die Collection durchgehen. Stattdessen wird der Auftrag in mehrere, kleinere Aufträge aufgeteilt, und diese kleineren Aufträge, werden von einzelnen Threads ausgeführt.

(Vgl.: [41] Seite 24f)

8.7.1.1. Iterieren von Collections mithilfe von Lambdas

In Java 8 wurde jeder Collection eine neue Methode hinzugefügt. Diese ist die `foreach`-Methode.

Der Unterschied zwischen dieser und der traditionellen `foreach`-Schleife lässt sich durch den Unterschied zwischen dem internen und externen Iterieren erklären. (Vgl.: [41] Seite 25ff)

8.7.1.1.1. Externes Iterieren

Beim externen Iterieren wird die Collection mithilfe eines sogenannten Iterators durchgegangen. Der Entwickler kann zwar selbst Optimierungen vornehmen, dies ist aber mit Aufwand verbunden. (Vgl.: [41] Seite 25ff)

Beispiel einer traditionellen foreach-Schleife

```
for( String listEntry: listOfStrings){
    System.out.println(listEntry);
}
```

Auflösung einer traditionellen Lambda-Expression

```
Iterator iter = listOfStrings.iterator();
while(iter.hasNext()){
    String listEntry = iter.next();
    System.out.println(listEntry);
}
```

8.7.1.1.2. Internes Iterieren

Beim internen Iterieren hat die Collection selbst Kontrolle darüber, was während der Iteration passiert. Der Entwickler gibt der Methode mit Hilfe einer Lambda-Expression nur die auszuführende Funktionalität mit. Durch das interne Exekutieren der foreach-Schleife können die Schleifendurchgänge durch Parallelisierung und andere Optimierungen verbessert werden. (Vgl.: [41] Seite 25ff)

```
listOfStrings.foreach(v-> System.out.println(v);
```

8.8. Throw-und Return-Statements in Lambda-Ausdrücken

Sowohl throw-, als auch return-Statements beenden zwar einen Lambda-Ausdruck, in dem sie aufgerufen werden, beenden aber niemals direkt die sie umschließende Methode.

Ein return-Statement beendet zwar die Lambda-Methode, aber nicht die umschließende Methode. Es können auch Rückgabewerte mitgegeben werden. Lambda-Ausdrücke funktionieren in diesem Sinne genauso wie eine reguläre Methode.

Auch ein throw-Statement beendet einen Lambda-Ausdruck, gibt aber anstatt eines Rückgabewertes eine Exception an die ausführende Methode zurück. Daher kann man auch Exceptions, die in einem Lambda-Ausdruck ausgelöst werden, außerhalb dieser mit Hilfe einer catch-Klausel abfangen. Daher funktioniert auch das throw-Statement, wie das return-Statement, genauso, wie in gewöhnlichen Methoden. (Vgl.: [40] Seite 47f)

8.9. Break-Statement

Ein Lambda-Ausdruck kann nicht von einem break-Statement beendet werden. Diese sind nur in Schleifen und switch-Statements erlaubt. (Vgl.: [40] Seite 46f)

9. Comparator

Der *Comparator* war ein von der Knapp AG gewünschtes Feature unserer Anwendung.

Die Daten einer Excel-Datei, sowie die einer XML-Datei, werden eingelesen und in eine einheitliche Struktur, genannt *WarehouseModelManager*, gebracht.

Diese werden anschließend verglichen. Das Resultat wird dann auf der GUI angezeigt.

Der *Comparator* kann, alternativ zum Start in der GUI, auch über die Kommandozeile ausgeführt werden.

9.1. Einlesen

Zuerst werden die angegebene Excel- und die XML-Datei eingelesen.

Beispiel:

```
wmExcel = new WarehouseModelManager();
wmXml = new WarehouseModelManager();
ComparatorHelper.readExcel(wmExcel, txtExcel.getText(),true);
ComparatorHelper.readXml(wmXml, txtSimulatorXml.getText(),true);
```

Durch die Methoden *readExcel()* und *readXml()* werden die Dateien eingelesen und in ein einheitliches Format, den *WarehouseModelManager*, gebracht. Dadurch können diese inhaltlich und strukturell unterschiedlichen Dateien verglichen werden.

Die Klasse *WarehouseModelManager* bildet ein ganzes Lagersystem ab und enthält eine Liste von Workflows.

Attribute der Klasse Workflow:

```
private final String workflowName;
private Set<AbstractStation> stations = new LinkedHashSet<>();
private final String workflowIdentifier;
private final StationTypeEnum stge;
```

Jeder Workflow wiederum enthält eine Liste der Klasse *AbstractStation*, welche als Elternklasse für die verschiedenen Stationstypen dient.

Diese drei Klassen bilden die Grundlage für den Vergleich.

9.2. Vergleichen

Nachdem die *WarehouseModelManager* erstellt wurden, werden sie mittels der Methode *compare()* verglichen. Für den Vergleich werden, bis auf wenige Ausnahmen, alle wichtigen Daten der zuvor genannten Klassen herangezogen. Anschließend wird das Ergebnis, das heißt, ob es sich um das gleiche Lagersystem handelt, auf der GUI angezeigt (siehe „Compare Excel with Xml“).

Beispielcode der Haupt-Compare-Methode:

```
private void compare() {  
    try {  
        boolean equals = false;  
  
        if (wmExcel.equals(wmXml)) {  
            equals = true;  
        }  
  
        addMessageToCompareResults();  
        removeRedundantEntriesFromFinalCompareResults();  
        ComparatorHelper.removeWorkflow();  
        addTabResult("Results", equals, wmXml, wmExcel);  
        addTabWorkflows("Workflows", equals);  
        addTabStations("Stations", equals);  
        workflowsExcel.clear();  
        workflowsXml.clear();  
  
    } catch (Exception e) {  
        Helper.logger.error(e.getMessage(), e);  
    }  
}
```

Beim einleitenden Aufruf werden der *WarehouseModelManager* der Excel-Datei (Abb: *wmExcel*) und jener der XML-Datei (Abb: *wmXml*) mittels der Methode *equals()*, die, um den Anforderungen gerecht zu werden, überschrieben wurde, verglichen.

9.2.1. Vergleich von WarehouseModelManagern

Zu Beginn wird überprüft, ob die beiden *WarehouseModelManager* die gleiche Anzahl an Workflows haben, anschließend werden iterativ alle Workflows der beiden *WarehouseModelManager* miteinander verglichen. Dieser Prozess wird auch weitergeführt, wenn bereits eine Abweichung in den beiden Dateien festgestellt wurde, da ein

umfangreicher Bericht, welche Workflows und somit auch *Stations* gleich sind und welche sich unterscheiden, gefordert war.

9.2.2. Vergleich von Workflows

Als erstes wird überprüft, ob die eingelesenen Workflows keine Daten enthalten.

Danach werden der Namen und die Anzahl an Stationen der beiden Workflows verglichen.

Letztendlich werden wieder iterativ alle Stationen der Workflows verglichen.

9.2.3. Vergleich von Stationen

Durch die große Anzahl an unterschiedlichen Stationstypen muss vor dem eigentlichen Vergleichen überprüft werden, ob die Stationen denselben Typ besitzen.

Sollte dies der Fall sein, werden sie mittels der geeigneten *equals*-Methode verglichen.

9.3. Anzeigen

Während des Vergleichens werden die gleichen sowie die unterschiedlichen Stationen in zwei separaten Listen gespeichert, um das Erstellen des Berichts zu erleichtern.

Die beiden Methoden *addMessageToCompareResults()* und *removeRedundantEntriesFromFinalCompareResults()* bereiten die Daten für den Compare-Bericht auf.

Dieser Bericht wird mit Hilfe der Methoden *addTabResult()*, *addTabWorkflows()* und *addTabStations()* auf der GUI in verschiedenen Tabs angezeigt (siehe „Compare Excel with Xml“).

10. Testing

Im Kapitel Testing wird das Verwendete Framework TestNG und die Unit Tests der Applikation beschrieben.

10.1. TestNG

TestNG ist ein Framework zum Testen von Java Applikationen. Es ist von JUnit und NUnit inspiriert, aber im Vergleich bietet TestNG viele neue Funktionalitäten. Das NG in Test NG steht für „Next Generation“.

TestNG ist ein Open Source Test-Framework, welches von Cedric Beust erschaffen wurde. Es wurde dazu entworfen, besser und simpler als JUnit zu sein.

TestNG hat das Ziel die Limitierungen alter Frameworks aufzuheben und soll dem Entwickler die Möglichkeit bieten, flexible Tests mithilfe von Annotationen, Gruppierung, Sequenzierung und Parametrieren zu schreiben.

(Vgl.: [45])

10.1.1. Vorteile von TestNG

Die größten Vorteile des TestNG Frameworks sind:

- HTML-Berichte über die Ausführung des Tests können erzeugt werden;
- Anhand von Annotationen kann die Programm/Funktion-Priorität leichter gesetzt werden;
- Prioritäten für Testfälle können gesetzt werden;
- Paralleles Testen mithilfe von Multithreading ist möglich;
- Testfälle können gruppiert werden;
- Logs werden auto-generiert;
- Daten können parametrisiert werden;
- Es ist sehr flexibel;

(Vgl.: [47])

10.1.2. TestNG im Vergleich zu JUnit

- Die Annotationen von TestNG sind verständlicher und simpler als JUnit
- Die Methodennamen in TestNG müssen dank Annotationen keiner gewissen Form oder Muster, wie in JUnit, entsprechen
- Testfälle können in TestNG gruppiert werden. Anhand dieser Gruppen ist es dann möglich, nur Testfälle einer gewissen Gruppe durchzuführen.
- TestNG hat im Vergleich zu JUnit 3 zusätzliche setUp bzw. tearDown Level:
@Before/AfterSuite, @Before/AfterTest, @Before/AfterGroup
- In TestNG müssen die Test Methoden keine Klasse erweitern.
- Das Erstellen von Testfällen, welche voneinander abhängig sind, ist in JUnit nicht möglich, in TestNG jedoch schon.
- TestNG ermöglicht es Testfälle Parallel durchzuführen.

(Vgl.: [45])

10.1.3. Annotationen

Annotationen in TestNG sind Codezeilen, welche kontrollieren, wie die Methode zu der sie gehören, ausgeführt werden soll.

@BeforeSuite	Die annotierte Methode wird vor allen Tests in der Test-Suite ausgeführt.
@AfterSuite	Die annotierte Methode wird nach allen Tests in der Test-Suite ausgeführt.
@BeforeTest	Die annotierte Methode wird vor allen Test-Methoden, die zu den Klassen innerhalb des <test> Tag gehören, ausgeführt.
@AfterTest	Die annotierte Methode wird nach allen Test-Methoden, die zu den Klassen innerhalb des <test> Tag gehören, ausgeführt.

@BeforeGroups	Der Test-Case wird vor einer oder mehreren Gruppe von Tests ausgeführt.
@AfterGroups	Der Test-Case wird nach einer oder mehreren Gruppe von Tests ausgeführt.
@BeforeClass	Die annotierte Methode wird vor dem Aufruf der ersten Test-Methode in der momentanen Klasse ausgeführt
@AfterClass	Die annotierte Methode wird, nachdem alle Methoden in der momentanen Klasse fertig sind, ausgeführt.
@BeforeMethod	Die annotierte Methode wird vor jeder Test-Methode ausgeführt.
@AfterMethod	Die annotierte Methode wird vor jeder Test-Methode ausgeführt.
<p>Die oben beschriebenen Annotationen werden vererbt, wenn die TestNG-Superklasse entsprechend annotiert wurde.</p> <p>Beispiel:</p> <pre>@BeforeSuite public void beforeSuite() { ... }</pre>	
@DataProvider	Legt fest, ob eine Methode für die Test-Methode Daten liefert. Die annotierte Methode muss ein Object[][] zurückliefern, in dem jedes Object[] einer Parameter-Liste einer Test-Methode zugewiesen werden kann.

@Factory	Kennzeichnet eine Methode als eine Factory, die Objekte zurückliefert, welche von TestNG als Test-Klasse benutzt wird. Die Methode muss ein Object[] zurückliefern.
@Listeners	Definiert die Listener einer Test-Klasse
@Parameters	Beschreibt, wie Parameter einer @Test-Method übergeben werden sollen.
@Test	Kennzeichnet eine Klasse oder Methode als Teil eines Tests.

Beispiel:

```
@Test(description = "This is an example", timeOut = 25000, priority = 10, dependsOnMethods = "test3", groups ="example")
public void test4() {
    ...
}
```

(Vgl.: [46])

10.1.3.1. Vorteile von Annotationen

TestNG-Tests müssen keine Klasse erweitern (Bei z.B. JUnit 3 muss die Test-Klasse Test-Case erweitern).

Es ist möglich zusätzliche Parameter an Annotationen zu übergeben.

Annotationen sind stark typisiert, also erkennt der Compiler etwaige Fehler sofort.

TestNG identifiziert die benötigten Methoden anhand der Annotation. Methodennamen müssen daher nicht einem bestimmten Muster oder Format entsprechen.

(Vgl.: [45])

10.1.4. Generierte Dateien

TestNG generiert nach dem Testen einen Ordner names ut-report:

Name	Änderungsdatum	Typ	Größe
📁 Ant suite	14.09.2016 08:07	Dateiordner	
chrome_emailable-report.html	31.08.2016 15:25	Chrome HTML Do...	52 KB
index.html	31.08.2016 15:25	Chrome HTML Do...	1 KB
testng.css	31.08.2016 15:25	CSS-Datei	1 KB
testng-results.xml	31.08.2016 15:25	XML-Dokument	53 KB

In diesem befindet sich das Ergebnis des Tests in Form einer XML Datei und als HTML Bericht (index.html bzw. emailable-report.html). Es wird für jede Test-Suite ein Ordner mit dem gleichen Namen erstellt. In diesem Fall, da es bei unserem Projekt nur eine Suite gab, gibt es nur den Ordner „Ant suite“. Innerhalb dieses Ordners befinden sich Informationen zur Test-Suite.

Name	Änderungsdatum	Typ	Größe
chrome_Ant test.html	31.08.2016 15:25	Chrome HTML Do...	46 KB
Ant test.properties	31.08.2016 15:25	PROPERTIES-Datei	1 KB
Ant test.xml	31.08.2016 15:25	XML-Dokument	20 KB
classes.html	31.08.2016 15:25	Chrome HTML Do...	29 KB
groups.html	31.08.2016 15:25	Chrome HTML Do...	1 KB
index.html	31.08.2016 15:25	Chrome HTML Do...	1 KB
main.html	31.08.2016 15:25	Chrome HTML Do...	1 KB
methods.html	31.08.2016 15:25	Chrome HTML Do...	46 KB
methods-alphabetical.html	31.08.2016 15:25	Chrome HTML Do...	46 KB
methods-not-run.html	31.08.2016 15:25	Chrome HTML Do...	1 KB
reporter-output.html	31.08.2016 15:25	Chrome HTML Do...	1 KB
testng.xml.html	31.08.2016 15:25	Chrome HTML Do...	8 KB
toc.html	31.08.2016 15:25	Chrome HTML Do...	2 KB

Abbildung 55: Test Suite "Ant Suite" Output

10.1.4.1. HTML Bericht (index.html)

Die gezeigten HTML Berichte sind aus unserem Projekt entnommen.

Test results

Suite	Passed	Failed	Skipped	testng.xml
Total	141	0	0	
Ant suite	141	0	0	Link

Abbildung 56: TestNG HTML Bericht Overview

In diesem Fall sind alle 141 Testfälle positiv verlaufen, wäre dies nicht der Fall, wäre die zweite Zeile „Ant suite“ rot gefärbt.

Ant test		
Tests passed/Failed/Skipped:	141/0/0	
Started on:	Wed Aug 31 15:25:12 CEST 2016	
Total time:	24 seconds (24724 ms)	
Included groups:		
Excluded groups:		
(Hover the method name to see the test class name)		
testAcsAutopickerFilledSimulatorXml	0	
testAddLift	0	
testAddWorkflow	0	
testConfigFileInvalid	0	<pre>com.knapp.sandbox.workflowgenerator.exceptions.ApplicationException: Gen004;.\doesntExist.xml at com.knapp.sandbox.workflowgenerator.helper.Helper.getXMLObjects(Helper.java:231) at com.knapp.sandbox.workflowgenerator.gui.controller.ControllerWorkflowGenerator pa... ... Removed 22 stack frames Click to show all stack frames</pre>
testConfigFileNull	0	<pre>java.lang.IllegalArgumentException: File cannot be null at javax.xml.parsers.DocumentBuilder.parse(DocumentBuilder.java:198) at com.knapp.sandbox.workflowgenerator.helper.Helper.getXMLObjects(Helper.java:221) at com.knapp.sandbox.workflowgenerator.gui.controller.ControllerWorkflowGenerator pa... ... Removed 22 stack frames Click to show all stack frames</pre>

Abbildung 57: TestNG HTML Bericht für einzelne Suite

Wenn man im oben gezeigten Screenshot in der Spalte Suite auf „Ant suite“ klickt, kommt man zum detaillierten Bericht der Test-Suite. In dieser Ansicht finden sich Informationen bezüglich der einzelnen Tests. In der Spalte „Time (seconds)“ ist zu sehen, wie lange gebraucht wurde, um den Test auszuführen. Diese Zeit ist in Sekunden angegeben. Sollte sich die Dauer eines Tests im Millisekunden-Bereich befinden, wird 0 angezeigt. Die Spalte „Exception“ zeigt den StackTrace der Exception, falls es diesen gibt.

10.1.5. testng.xml & ant

In TestNG wird eine Test-Suite nicht mit Source-Code geschrieben bzw. definiert, sondern mithilfe einer testng.xml-Datei. Es ist jedoch auch möglich, dies mit Ant oder der Kommandozeile zu realisieren. Dadurch ist die Konfiguration der Test-Suites sehr flexibel. Ein testng.xml kann beispielsweise so aussehen:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >

<suite name="Suite1">

    <test name="exapletest1">
        <classes>
            <class name="Test1" >
                <methods>
                    <exclude name="m1" />
                    <include name="m2" />
                </methods>
            </class>
        </classes>

        <groups>
            <run>
                <exclude name="brokenTests" />
                <include name="checkinTests" />
            </run>
        </groups>
    </test>

    <test name="exapletest2">
        <packages>
            <package name="sample" />
        </packages >
    </test>
</suite>
```

Die Suite wird mit einem `<suite>` Tag, die einzelnen Tests mit einem `<test>` Tag und die Klassen, in denen sich der Test befinden, wird, mit einem `<classes>` Tag definiert. Die sich darin befindenden Klassen werden mit einem `<class>` Tag bestimmt. Anstelle eines `<classes>` bzw. `<class>` Tags ist es auch möglich, Packages innerhalb eines `<packages>` bzw. `<package>` Tags anzugeben. Es ist ebenso möglich einzelne Methoden oder Gruppen eines Tests mit einem `<include>` Tag zu inkludieren und mit einem `<exclude>` Tag zu exkludieren. Einzelne Methoden lassen sich jedoch wie oben beschrieben mit Annotationen exkludieren.

Da wir während des Projektes mit Ant gearbeitet haben und somit schon eine build.xml vorhanden war, wurde der Start von TestNG auch mithilfe der build.xml geregelt.

```
<!-- Define task 'testng' to run unit-tests with lib -->
<taskdef name="testng" classname="org.testng.TestNGAntTask">
    <classpath location="${lib.testng}" />
</taskdef>

<!-- Run unit tests. classpathref="cptestng" -->
<testng
    classpath="${dir.deploy-test-path}:${dir.deploy-libs-path}"
    outputDir="${dir.testng-report-path}"
    haltOnFailure="true">

    <!-- Add test and src compiled classes to classpath which are required for unit tests -->
    <classpath>
        <fileset dir="${dir.deploy-libs-path}">
            <include name="**/*" />
        </fileset>

        <fileset dir="${dir.deploy-test-path}">
            <include name="**/*" />
        </fileset>
    </classpath>

    <!-- Execute now all compiled test classes -->
    <classfileset dir="${dir.deploy-test-path}" includes="**/Test*.class" />
</testng>
```

(Vgl.: [43], [44])

10.1.6. Beispielhafter Aufbau eines TestNG-Testfalles

In diesem Beispiel wird die Methode „`testEqualsTrue`“ der Klasse „`testWarehouseModelEquals`“ beschrieben. Der Sinn dieser Methode war es den gesamten Comparator zu testen. Dieses Beispiel wurde gewählt, da die Klasse `WareHouseModelManager` die komplexeste Klasse der Anwendung darstellt.

```

@Test
public void testEqualsTrue() {
    try {
        //add Workflows
        Workflow wf1 = new Workflow("WorkflowName", "WorkflowIdentifier", StationTypeEnum.WCSPLC);
        Workflow wf2 = new Workflow("WorkflowName", "WorkflowIdentifier", StationTypeEnum.WCSPLC);

        //add EndeStation
        EndeStation es1 = new EndeStation("Unit-Test", "E01", wf1);
        EndeStation es2 = new EndeStation("Unit-Test", "E01", wf2);

        //add HardwareStation
        String stationName = "HardwareStationTest";
        Integer stationNumber = 1;
        StationTypeEnum stationType = StationTypeEnum.ACSAUTOPICKER;
        HardwarePhysicalGateway physicalStationRef = new HardwarePhysicalGateway("testGateway", "testGateway", 2);
        String defaultControlInfo = "-";
        final HardwareStation hs1 = new HardwareStation("TestSource", stationName, wf1, stationNumber, stationType,
            1, physicalStationRef, defaultControlInfo);

        final HardwareStation hs2 = new HardwareStation("TestSource", stationName, wf2, stationNumber, stationType,
            1, physicalStationRef, defaultControlInfo);

        //add InternalStation
        final InternalStation is1 = new InternalStation("testSource", stationName, wf1, stationNumber);
        final InternalStation is2 = new InternalStation("testSource", stationName, wf2, stationNumber);

        wf1.addStationToWorkflow(es1);
        wf1.addStationToWorkflow(hs1);
        wf1.addStationToWorkflow(is1);

        wf2.addStationToWorkflow(es2);
        wf2.addStationToWorkflow(hs2);
        wf2.addStationToWorkflow(is2);

        WarehouseModelManager wml = new WarehouseModelManager();
        WarehouseModelManager wm2 = new WarehouseModelManager();

        wml.addWorkflow(wf1);
        wml.addWorkflow(wf2);

        wm2.addWorkflow(wf1);
        wm2.addWorkflow(wf2);

        Assert.assertTrue(wml.equals(wm2));

    } catch (Exception e) {
        Assert.fail(e.getMessage(), e);
    }
}

```

Wie bereits im Kapitel Software-Architektur – `WarehouseModel` beschrieben, besteht ein `WarehouseModel` aus `Workflows`, welche wiederum Stationen beinhalten.

Da der Ausgang dieses Testfalls sein soll, dass 2 `WarehouseModels` ident sind, werden am Anfang 2 identische `Workflows` `w1` und `w2` erstellt. Diese `Workflows` müssen nach der Erstellung mit Stationen gefüllt werden. Die ersten Stationen, die erstellt werden, sind

EndeStations, welche das Ende eines Workflows darstellen, das heißt, diese Stationen haben unter keinen Umständen einen Nachfolger. Als nächstes werden *HardwareStations* und *InternalStations* erstellt. Anschließend werden die Stationen es1, hs1 und is1 zum ersten und es2, hs2 und is2 zum zweiten Workflow hinzugefügt.

Zum Schluss wird das Herzstück dieses Testfalls erstellt: die beiden WarehouseModelManager bzw. *WarehouseModels* wm1 und wm2, an welche auch nach der Erstellung die beiden Workflows angehängt werden. Das `Assert.assertTrue(wm1.equals(wm2))` liefert in diesem Fall True.

11. Verwendete Tools

11.1. SonarQube

11.1.1. Allgemein

SonarQube ist eine open-source-Plattform für statische Code-Analyse der Qualität von Sourcecode. Der Sourcecode eines Programms wird hinsichtlich verschiedener Qualitätsbereiche analysiert, die Ergebnisse werden auf einer Website dargestellt. SonarQube wurde in Java programmiert, unterstützt aber neben der Analyse von Java-Programmen mit entsprechenden Plugins unter anderem auch die Programmiersprachen C#, C/C++, PHP, PL/SQL.

Wie auch in unserem Fall kann eine Codeanalyse mittels SonarQube von einem laufenden Jenkins-Server gestartet werden.

(Vgl.: [48])

Issues			
<input checked="" type="checkbox"/> Severity			
Blocker	0	Minor	0
Critical	0	Info	1
Major	18		
<input checked="" type="checkbox"/> Resolution			
Unresolved	19	Fixed	5,221
False Positive	0	Won't fix	6
Removed	49		
<input type="checkbox"/> Status			
<input type="checkbox"/> New Issues			
<input type="checkbox"/> Rule			
<input type="checkbox"/> Tag			
<input type="checkbox"/> Module			
<input type="checkbox"/> Directory			
<input type="checkbox"/> File			
<input type="checkbox"/> Assignee			
<input type="checkbox"/> Reporter			
<input type="checkbox"/> Author			
<input type="checkbox"/> Language			
<input type="checkbox"/> Action Plan			

Abbildung 58: SonarQube

SonarQube besteht aus drei Komponenten:

- Einem Modul für Build-Management-Tools wie Apache Ant
Dieses analysiert den Sourcecode hinsichtlich der Qualitätsmerkmale.
- Einer Datenbank, in der die Testergebnisse der Analyse gespeichert werden.
- Einer Website für einfaches Management und Auswertung der Testergebnisse.

11.1.2. Qualitätsmerkmale

SonarQube analysiert den Sourcecode hinsichtlich folgender Qualitätsbereiche:

- Softwarearchitektur & Softwaredesign
- Doppelter Code
- Modultests
- Komplexität
- Potenzielle Fehler
- Kodierrichtlinien
- Kommentare

Wird ein möglicher Verstoß gegen eines dieser Qualitätsmerkmale erkannt, legt Sonar einen sogenannten "Issue" an. Alle anfallenden Issues können auf der Website eingesehen werden, zudem wird auf die Stelle im Code verwiesen, wo der Fehler aufgetreten ist und ein Lösungsansatz angeboten.

Die Issues werden wie folgt eingeteilt:

1. BLOCKER

Ein Bug, der mit hoher Wahrscheinlichkeit ein Fehlverhalten des Programms herbeiführt, wie zum Beispiel nicht geschlossene Datenbankverbindungen.
Blocker sollten schnellstmöglich wieder behoben werden.

2. CRITICAL

Ist entweder ein kleinerer Bug oder ein Sicherheitsmangel, zum Beispiel eine nicht behandelte Fehlermeldung.
Auch Criticals sollten zeitnah behoben werden.

3. MAJOR

Gröbere Qualitätsmängel, die sowohl die Produktivität der Entwickler, als auch das Voranschreiten des Entwicklungsprozesses deutlich behindern können.

Diese können sein: Doppelter Code, zu viele Bedingungen in einer Abfrage.

4. MINOR

Kleinere Qualitätsmängel, die Entwickler möglicherweise stören könnten.

Solche wären: Zu lange Codezeilen, nicht gebrauchte Parameter.

5. INFO

Minimaler Makel, wie zum Beispiel unnötige Imports.

Die Regeln für ein Projekt können je nach Bedarf angepasst werden, für den Fall, dass Verstöße bewusst gemacht werden sollten. Dadurch scheinen diese bei der Auswertung nicht mehr auf. In unserem Fall wurden unter anderem auch If-Anweisungen mit mehr als drei Bedingungen toleriert.

(Vgl.: [48])

11.1.3. Quality-Gates

Quality Gates geben an, ob der derzeitige Stand eines Projekts den vorgegebenen Qualitätsvorgaben entspricht. Zusätzlich zu den Issues wird die “Technical Dept” angezeigt, welche eine ungefähre Schätzung der Zeit darstellt, die zum Beheben aller derzeit im Programm befindlichen Issues gebraucht wird.

Diese Vorgaben lassen sich für jedes Projekt einzeln festlegen und können, bezogen auf dieses Projekt, sein: keine Blocker/Critical Issues, weniger als 2% duplizierter Code.

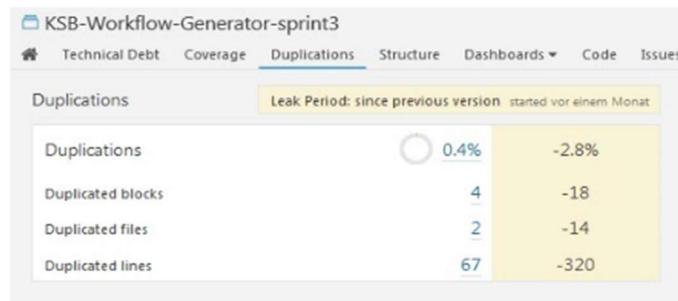


Abbildung 59: SonarQube Quality Gates



Abbildung 60: SonarQube Issues

11.1.4. Statische Code-Analyse

Bei der statischen Code-Analyse wird eine Software überprüft, ohne diese dabei auszuführen. Das bedeutet, der Quelltext wird auf formale Fehler überprüft, gefundene Fehler können somit noch vor dem Ausführen der Software behoben werden.

Ziel ist es, mögliche Schwachstellen, Fehler und Sicherheitslücken zu finden und aufzuzeigen. Dies wird meist mit Hilfe von Tools automatisiert durchgeführt.

Ein weiterer Vorteil von Tools ist, dass sie im Vergleich zum manuellen Code-Refactoring and Debugging um einiges schneller und ausgedehnter arbeiten.

Dies gewährleistet von Beginn eines Projekts an eine hohe Qualität und somit auch leichte Wartbarkeit des Codes.

Zudem macht sie Programmierer auf "unsauberen" Code oder schlechten Programmierstil aufmerksam und hilft somit, Code-Richtlinien und Best Practices innerhalb eines Unternehmens einzuführen und durchzusetzen.

(Vgl.: [49], [53])

11.1.4.1. Software Metriken

Software Metriken dienen dazu, auf die Qualität und Struktur einer Software bezogene Daten, zu messen und somit vergleichbar zu machen und verständlicher darzustellen. Hierfür können die folgenden verschiedenen Metriken angewandt werden:

11.1.4.1.1. Objektorientierte Metriken

Als Grundlage dafür dienen die Strukturmerkmale der Klassen einer Anwendung.

Maße können sein:

Response for a Class (RFC)	Ist die Anzahl aller möglichen ausführbaren Methoden einer Klasse. Auch über Assoziationen erreichbare Methoden werden mitgezählt.
Number of Children (NOC)	Gibt die Anzahl der direkten Spezialisierungen einer Klasse an. Damit lässt sich das Maß der Auswirkung, bei Vererbung von Fehlern der Superklasse auf die Unterklassen, bewerten.
Coupling between objects (CBO)	Gibt an, von wie vielen anderen Klassen die betrachtete Klasse Instanzvariablen und Methoden verwendet.

11.1.4.1.2. Umfangs Metriken

Dienen zum Messen die Größe eines Programms.

Lines of Code (LOC)	Ist die gesamte Anzahl der effektiven Codezeilen einer Anwendung. Diese ist jedoch stark von der verwendeten Programmiersprache und vom Programmieren selbst abhängig.
Number of Statements (NOS)	Hierbei werden die ausführbaren Anweisungen gezählt. Diese Methode ist objektiver und somit aussagekräftiger als LOC.

11.1.4.1.3. Metriken zur Komplexität

Bezieht sich auf Merkmale von Methoden, die Aufschluss über die Komplexität geben.

Halstead Metriken	Beruht auf der Anzahl und dem Zusammenhang der Operatoren (Rechenoperatoren, Schleifen, usw.) und Operanden (Variablen, Konstanten) einer Methode. Daraus ergeben sich Kennzahlen wie der Programmieraufwand, oder die Schwierigkeit einen Algorithmus zu implementieren oder zu verstehen.
McCabe-Metrik	Versucht eine Software auf Basis ihrer logischen Struktur zu analysieren. Die Komplexität wird anhand der Häufigkeit von Anweisungen und der sich daraus ergebenden möglichen Abläufe berechnet. Wird häufig für die Ermittlung von Test- und Wartungsaufwänden benutzt.

(Vgl.: [50], [51])

11.1.5. Dynamische Code-Analyse

Im Gegensatz zur statischen Code-Analyse wird bei der dynamischen eine Anwendung während des Ausführens überprüft. Dabei werden immer dieselben Schritte durchlaufen: Die Software wird ausgeführt, Daten werden eingegeben, das Verhalten wird beobachtet und schließlich wird überprüft, ob die verarbeiteten Daten dem gewünschten Output entsprechen. Um dieses Analyseverfahren durchführen zu können, muss jedoch eine lauffähige Version der Anwendung vorhanden sein.

Diese Art der Analyse eignet sich besonders, um die Auswirkung der Anwendung auf Systemressourcen wie CPU- und RAM-Auslastung zu messen und Sicherheitslücken zu finden. Jedoch beschränkt sich die Analyse auf die ausgeführten Programmteile und kann somit keine allgemeine Aussage über die Codequalität liefern.

Eine dynamische Code-Analyse kann entweder:

- durch eine Person

(Der Entwickler selbst, oder eine andere Person, testet, ob die Software den Ansprüchen entspricht und einwandfrei funktioniert.

Neben dem manuellen Testen und Debuggen des Codes sind Unit-Tests das häufigste Verfahren.)

- oder mittels Tools erfolgen.

(Viele Entwicklungsumgebungen beinhalten integrierte Tools zur Code-Analyse.

Externe Anwendungen wären zum Beispiel der JProfiler für Java.)

(Vgl.: [52], [53])

11.1.6. SonarQube im Vergleich

Im Folgenden werden alternative Programme zu SonarQube aufgezählt und deren Stärken kurz beschrieben.

11.1.6.1. Checkstyle

Checkstyle eignet sich besonders für die Einhaltung von Code-Conventions und Best Practices.

11.1.6.2. PMD

PMD legt den Fokus mehr auf die Vermeidung von potenziellen Defekten und unsicheren, ineffizienten Code-Teilen, im allgemeinen „Bad Practices“.

11.1.6.3. FindBugs

Im Gegensatz zu den vorherigen Tools beschränkt sich FindBugs großteils auf Bugs und möglichem unperformantem Code. Da es auch schwer zu findende Bugs erkennt, die von anderen Tools übersehen werden könnten, ist es eine gute Erweiterung.

(Vgl.: [54])

11.1.7. Warum SonarQube

SonarQube spezialisiert sich nicht auf einen Gesichtspunkt des Codes, es versucht möglichst viele Aspekte (siehe „SonarQube - Qualitätsmerkmale“) mit einzubeziehen und bietet zudem mögliche Lösungsvorschläge.

Des Weiteren zeigt es zusätzlich zum Stand des letzten Commit auch Trends im Verlauf des Projekts.

Der wohl größte Vorteil jedoch ist, dass diese Code-bezogenen Kennzahlen in wirtschaftlich relevante Werte, wie z.B. geschätzter Zeitaufwand zum Beheben von Bugs oder Risiko für das Projekt, umgerechnet werden und somit auch für das Management interessante Daten darstellen.

(Vgl.: [54])

11.2. Jenkins

11.2.1. Allgemein

Jenkins ist eine webbasierte Software zur kontinuierlichen Integration von Programmteilen in ein Endprodukt.

Jenkins wurde in Java geschrieben und kann auf einem Application-Server mit JRE installiert werden, ohne zusätzliche Erweiterungen zu benötigen.

Jenkins kann zum Builden, Deployen oder auch zum Testen genutzt werden.

Die Ergebnisse können mittels eines Browsers auf dem Dashboard von Jenkins angezeigt und überwacht werden. Des Weiteren finden sich dort auch die Ergebnisse von älteren Versionen der Anwendung und Links zu etwaigen Plugins.

In unserem Fall wurde nach jedem Checkin in das Git Repository automatisch durch Jenkins ein Build des aktuellen Programms gestartet und die angelegten Testfälle überprüft.

Die Verwendung von Jenkins wurde von der Knapp AG vorgegeben.

(Vgl.: [55])

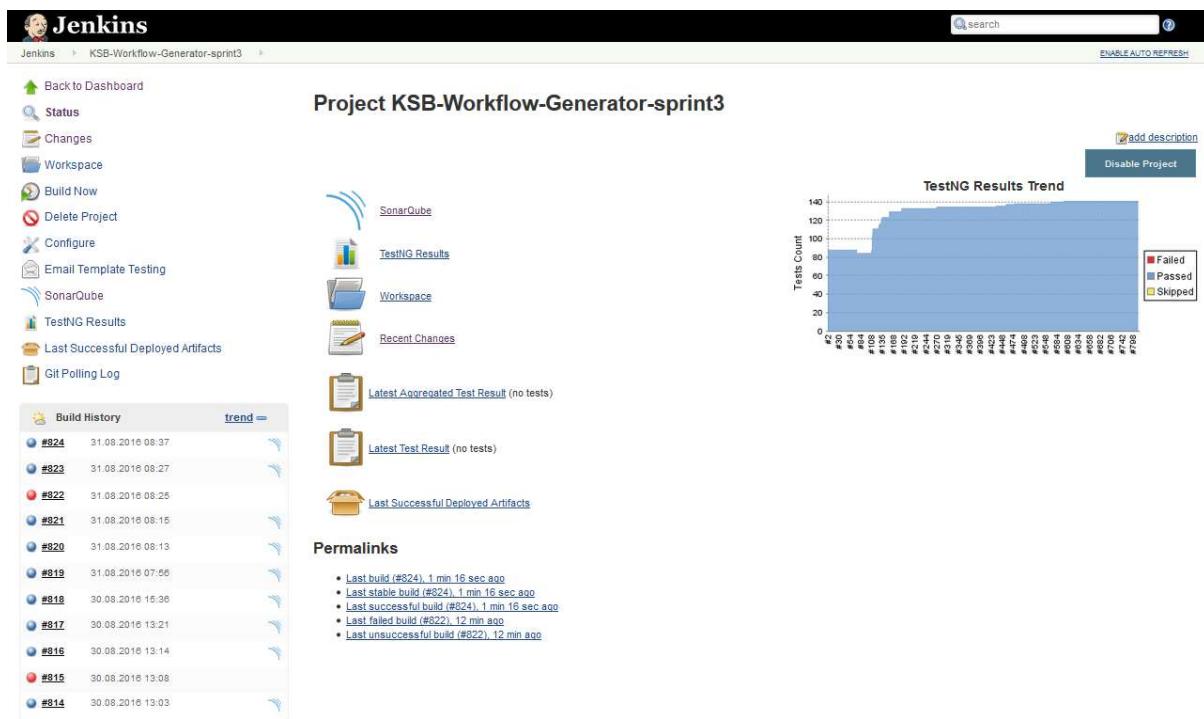


Abbildung 61: Jenkins Dashboard

11.2.2. Kontinuierliche Integration

Kontinuierliche oder fortlaufende Integration wird für das Zusammenfügen von Komponenten zu einer Anwendung benutzt. Das Ziel der kontinuierlichen Integration ist eine Steigerung der Softwarequalität und das frühe Erkennen von möglichen Fehlern.

Üblicherweise wird dafür nicht nur die gesamte Anwendung neu gebaut (build), sondern es werden auch automatisierte Tests durchgeführt.

Dieser Vorgang wird meist automatisch ausgelöst durch Einchecken in die Versionsverwaltung, in unserem Fall Git.

Vorteile

- Integrations-Probleme werden laufend entdeckt und gefixt;
- Frühe Warnungen bei nicht zusammenpassenden Bestandteilen;
- Sofortige Unitests;
- Ständige Verfügbarkeit eines lauffähigen Standes für Demo-, Test- oder Vertriebszwecke;

- Die sofortige Reaktion des Programms auf das Einchecken eines fehlerhaften oder unvollständigen Codes „erzieht“ die Entwickler zu einem verantwortlicheren Umgang und kürzeren Checkin-Intervallen;

(Vgl.: [56])

11.2.3. Plugin: SonarQube

Mit dem SonarQube-Plugin für Jenkins lässt sich Sonar in einem bestehenden Jenkins-Server integrieren und auch konfigurieren.

Dadurch kann von Jenkins aus, nach einem neuen Checkin oder auch manuell, eine SonarQube-Analyse gestartet werden.

Nach der Beendigung der Analyse wird auf dem Dashboard von Jenkins der Qualitygate-Status und ein Link zur Sonar-Seite eingeblendet, auf der ein detaillierter Bericht eingesehen werden kann.

(Vgl.: [55])

11.3. Log4J

Log4J ist ein Logging-Framework für Java und besteht aus 3 Hauptbestandteilen:

- **Logger**
Logger sind verantwortlich für das Aufnehmen von Informationen.
- **Appender**
Appender sind dafür verantwortlich, die Nachrichten an die verschiedenen Ausgabeziele auszugeben (z.B. ConsoleAppender, FileAppender).
- **Layouts**
Layouts sind verantwortlich für die Formatierung der Nachrichten. (z.B. Präfix)

Es kann der Logging-Level umgeschaltet werden, um zusätzliche Meldungen zur Fehlersuche einzuschalten. Der Logging-Level kann als die Wichtigkeit der Nachricht verstanden werden, nachdem die Ausgabe gefiltert wird. Folgende Logging-Level bietet Log4J.

1. ALL

Alle Meldungen werden ausgegeben;

2. TRACE

Sehr spezifische Meldungen über das Debugging;

3. DEBUG

Allgemeine Meldungen für das Debuggen der Anwendung;

4. INFO

Allgemeine Meldungen wie z.B. Programm startet, usw.;

5. WARN

Potenziell schädliche Meldungen oder das Auftreten einer unerwarteten Situation;

6. ERROR

Fehlerfälle, welche die Anwendung nicht zwingend abstürzen lassen;

7. FATAL

Schwerwiegende Fehlerfälle, welche mit einer hohen Wahrscheinlichkeit zum Absturz des Programms führen;

8. OFF

Logging ist ausgeschalten;

(Vgl.: [57], [58], [59])

VI. Resümee

Abschließend kann gesagt werden, dass fast alle Ziele erfüllt wurden und deswegen auch alle am Projekt beteiligten sehr zufrieden mit dem Ergebnis sind.

Die im Kapitel „Einleitung und Überblick“ beschriebene Problemstellung konnte vollkommen gelöst werden. Das Hauptziel, der produktive Einsatz der Applikation, innerhalb der Knapp AG konnte erreicht werden und somit ist die gewünschte Zeitersparnis in Zukunft gewährleistet.

In einem Praxistest war es möglich 10 *WarehouseModels* innerhalb von 20 Minuten in das gewünschte Format zu bringen (entweder Excel oder XML). Bisher dauerte das händische Umschreiben einer Datei ca. 4 Stunden.

Durch die Größe unseres Teams von 8 Entwicklern und 3 Vorgesetzten war es uns möglich viel Erfahrung in Sachen Teamarbeit zu sammeln. Darüber hinaus war die strikte Anwendung von Scrum sehr hilfreich, um nachvollziehen zu können, wie Software-Projekte in Betrieben ablaufen.

Da das Projekt ein Erfolg war, kann davon ausgegangen werden, dass die Anwendung in Zukunft intern noch weiterentwickelt werden wird.

VII. Glossar

Ant	Apache Ant ist ein Tool zur Ausführung von Prozessen, die in einer so genannten “Build-Datei” beschrieben werden. Dabei werden meist Java-Applikationen gebaut. Dazu bietet Ant eine Reihe von eingebauten Aufgaben zum Kompilieren, Testen und Ausführen. (Vgl.: [60])
Git	Das Supply-Chain-Management-Tool Git wurde während der Entwicklung zur Versionierung und zum Datenaustausch benutzt. Abseits dieser Funktionen bietet Git noch weitere Vorteile, wie die Aufteilung in mehrere Entwicklungszweige. Des Weiteren unterliegt Git der GNU GENERAL PUBLIC License und ist somit Open Source und auch für Unternehmen geeignet. (Vgl.: [61])
Eclipse IDE	Bei unserem Projekt wurde diese Entwicklungsumgebung gewählt, da von der Knapp AG eine Eclipse Version mit allen benötigten Plugins und Einstellungen bereits vorhanden war. So konnten wir sofort innerhalb der ersten Woche mit der Entwicklung des Programms starten.

VIII. Abbildungsverzeichnis

Abbildung 1: Beispiel Excel Tabelle (Quelle Excel-Logo: [35])	12
Abbildung 2: Beispiel XML Datei (Quelle: XML Icon: [60]).....	13
Abbildung 3: XML to Excel.....	13
Abbildung 4: Aufbau WarehouseModel.....	17
Abbildung 5:Ladevorgang einer Excel-Datei	19
Abbildung 6: Aufbau ExcelReader	20
Abbildung 7:Aufbau XML-Generator.....	21
Abbildung 8: Aufbau Comparator	22
Abbildung 9: Ablauf von Scrum (Quelle: [2])	24
Abbildung 10: Hauptfenster	32
Abbildung 11: General Settings Dialog.....	33
Abbildung 12: Log Dialog.....	34
Abbildung 13: Close Dialog.....	35
Abbildung 14: About Dialog	35
Abbildung 15: Excel to	36
Abbildung 16: TabBar	37
Abbildung 17: Workflows in Liste.....	37
Abbildung 18: Show Warnings, Conveyor Elements, Zoom Buttons	37
Abbildung 19: Export to CFG bzw. XML und Preview Configuration Button	38
Abbildung 20: Show Warnings	38
Abbildung 21: Export to CFG Dialog	39
Abbildung 22: Export to XML Dialog	40
Abbildung 23: Preview Configuration	41
Abbildung 24: XML to	42
Abbildung 25: XML to ... Vorschau Tabelle	43
Abbildung 26: Compare Excel with XML	43
Abbildung 27: Results Tab	44
Abbildung 28: Results Workflows Tab	44
Abbildung 29: Result Staions Tab.....	45
Abbildung 30: Compare with Stationloader.....	46
Abbildung 31: Compare with Stationloader Result.....	46
Abbildung 32: Stationtype Mapping	47
Abbildung 33: Icon Ordner	47
Abbildung 34: Icon Blaues Rufzeichen	47
Abbildung 35: Icon rotes Fragezeichen.....	48
Abbildung 36: Fixed Value, Object Attribute	48
Abbildung 37: Previes Stationtype-Mapping	49
Abbildung 38: Stages unter verschiedenen Betriebssystemen (Quelle: [12])	54
Abbildung 39: Stage und Scene.....	55

Abbildung 40: FXML	55
Abbildung 41: CSS.....	56
Abbildung 42: RichtextFX CodeArea	57
Abbildung 43: JavaFX SceneBuilder.....	59
Abbildung 44: WPF GUI Builder	59
Abbildung 45 Vereinfachtes Ablaufdiagramm des Einlesens einer Excel-Datei	62
Abbildung 46 Vereinfachtes Ablaufdiagramm der Konvertierung einer eingelesenen Excel-Datei	66
Abbildung 47 Lebenszyklus eines DOM-Dokuments (Quelle Excel-Logo: [35])	67
Abbildung 48 Beispiel einer Station mit manuell eingefügten Leerzeilen	68
Abbildung 49 Beispiel einer Station ohne manuell eingefügten Leerzeilen	68
Abbildung 50 Beispiel von generierten Workflow-Dateien	68
Abbildung 51 Teil der workflow-generator Datei, die für die Zuteilung der Workflows in die Ordner zuständig ist	69
Abbildung 52 Beispiel einer main-workflow.xml Datei.....	69
Abbildung 53 Beispiel einer simulator.xml Datei.....	70
Abbildung 54 Visualisierung einer Baum-Struktur nach dem Document Object Model (Quelle: [30])	71
Abbildung 55: Test Suite "Ant Suite" Output	99
Abbildung 56: TestNG HTML Bericht Overview	99
Abbildung 57: TestNG HTML Bericht für einzelne Suite	100
Abbildung 58: SonarQube	105
Abbildung 59: SonarQube Quality Gates	107
Abbildung 60: SonarQube Issues.....	108
Abbildung 61: Jenkins Dashboard	113

IX. Literaturverzeichnis

Kapitel Scrum

- [1] Oroox: Agile-Entwicklung
<http://www.oroox.com/de/dev/Agile-Entwicklung>
05.01.2017
- [2] Wikipedia: Scrum
<https://de.wikipedia.org/wiki/Scrum>
27.03.2017
- [3] Agiles-Projektmanagement: Vorteile und Nachteile von Scrum
<http://agiles-projektmanagement.org/scrum-vorteile-nachteile/>
03.04.2017
- [4] Scrum-Master: Scrum Rollen
<http://scrum-master.de/Scrum-Rollen>
03.04.2017
- [5] Scrum-Master: Product Backlog
http://scrum-master.de/Scrum-Glossar/Product_Backlog
03.04.2017
- [6] Scrum-Master: Sprint Backlog
http://scrum-master.de/Scrum-Glossar/Sprint_Backlog
03.04.2017

Kapitel Frontend

- [7] Java Forums: A JavaFX Blog
<http://www.java-forums.org/blogs/javafx/1596-life-cycle-methods-javafx-application.html>
27.12.2016
- [8] Wikipedia: JavaFX
<https://de.wikipedia.org/wiki/JavaFX>
27.12.2016
- [9] Oracle Docs: JavaFX Scenegraph
<https://docs.oracle.com/javase/8/javafx/scene-graph-tutorial/scenegraph.htm>
27.12.2016
- [10] Oracle Docs: Working on the Scene Graph
<http://docs.oracle.com/javafx/2/scenegraph/jfxpub-scenegraph.htm>
27.12.2016
- [11] Oracle Docs: Scene
<https://docs.oracle.com/javase/8/javafx/api/javafx/scene/Scene.html>

27.12.2016

- [12] Oracle Docs: Stage

<https://docs.oracle.com/javase/8/javafx/api/javafx/stage/Stage.html>

27.12.2016

- [13] Github Thomas Mikula: RichTextFX

<https://github.com/TomasMikula/RichTextFX>

27.12.2016

- [14] Softwareengineering: Understanding Visual Studio Community Edition

<http://softwareengineering.stackexchange.com/questions/262916/understanding-visual-studio-community-edition-license>

03.04.2017

Kapitel Excel-Reader

- [15] Tutorialspoint: Apache POI Overview

https://www.tutorialspoint.com/apache_poi/apache_poi_overview.htm

28.12.2016

- [16] Tutorialspoint: Apache POI – Workbooks

https://www.tutorialspoint.com/apache_poi/apache_poi_workbooks.htm

28.12.2016

- [17] Tutorialspoint: Apache POI – Cells

https://www.tutorialspoint.com/apache_poi/apache_poi_cells.htm

28.12.2016

- [18] Tutorialspoint: Apache POI

https://www.tutorialspoint.com/apache_poi/apache_poi_spreadsheets.htm

28.12.2016

- [19] Apache POI: Formula Support

18.02.2017

<https://poi.apache.org/spreadsheet/formula.html>

- [20] Tutorialspoint: Formula

18.02.2017

<https://poi.apache.org/spreadsheet/formula.html>

Kapitel XML-Generierung

- [21] W3: DOM & Java DOM Parser Einführung

<https://www.w3.org/TR/REC-DOM-Level-1/introduction.html>

27.12.2016

- [22] Tutorialspoint: DOM & Java DOM Parser Einführung

https://www.tutorialspoint.com/java_xml/java_dom_parser.htm

27.12.2016

-
- [23] Oracle Docs: Node
<https://docs.oracle.com/javase/7/docs/api/org/w3c/dom/Node.html>
27.12.2016
 - [24] Oracle Docs: Document
<https://docs.oracle.com/javase/7/docs/api/org/w3c/dom/Document.html>
27.12.2016
 - [25] Oracle Docs: Comment
<https://docs.oracle.com/javase/7/docs/api/org/w3c/dom/Comment.html>
27.12.2016
 - [26] Oracle Docs: Element
<https://docs.oracle.com/javase/7/docs/api/org/w3c/dom/Element.html>
27.12.2016
 - [27] Oracle Docs: Attr
<https://docs.oracle.com/javase/7/docs/api/org/w3c/dom/Attr.html>
27.12.2016
 - [28] Oracle Docs: Text
<https://docs.oracle.com/javase/7/docs/api/org/w3c/dom/Text.html>
27.12.2016
 - [29] Oracle Docs: OutputKeys
<http://docs.oracle.com/javase/1.5.0/docs/api/javax/xml/transform/OutputKeys.html>
27.12.2016
 - [30] Visualisierung eines DOM-Baumes
https://www.w3schools.com/js/js_htmldom.asp
27.12.2016
 - [31] DOM-Parser VS SAX-Parser
<http://howtodoinjava.com/xml/dom-vs-sax-parser-in-java/>
04.3.2017
 - [32] Unterschied zwischen DOM-Parser und SAX-Parser
<http://javarevisited.blogspot.co.at/2011/12/difference-between-dom-and-sax-parsers.html>
04.3.2017
 - [33] XML-Parser Grundlegendes
http://www.uzi-web.de/parser/parser_grundlegendes.htm
04.3.2017
 - [34] XML (XSD): Übersicht
<https://www.liquid-technologies.com/xml-schema-tutorial/xsd-elements-attributes>
13.02.2017
 - [35] XML(XSD): Class Schema
<https://docs.oracle.com/javase/7/docs/api/javax/xml/validation/Schema.html>
18.02.2017
 - [36] Excel-Bild

https://commons.wikimedia.org/wiki/File:Microsoft_Excel_2013_logo_with_background.png

1.04.2017

- [37] XML (XSD): Namespaces

<https://www.liquid-technologies.com/xml-schema-tutorial/xsd-namespaces>

13.02.2017

- [38] W3C XML Schema Definition Language W3C Recommendation 5 April 2012

<https://www.w3.org/TR/xmlschema11-1/>

13.02.2017

Kapitel Lambda Ausdrücke

- [39] Mithil Shah: Java 8 – Lambdas, Method References and Composition

http://www.studytrails.com/java/java8/java8_lambdas_functionalprogramming/

04.03.2017

- [40] Lambda Expressions in Java-Reference

<http://www.angelikalanger.com/Lambdas/LambdaReference.pre-release.pdf>

Stand: 27.03.2017

Quelle: <http://www.angelikalanger.com/Lambdas/Lambdas.html>

Autor: Angelika Langer & Klaus Kreft

- [41] Lambda Expressions in Java-Tutorial

<http://www.angelikalanger.com/Lambdas/Lambdas.pdf>

Stand: 21.3.2017

Quelle: <http://www.angelikalanger.com/Lambdas/Lambdas.html>

Autor: Angelika Langer & Klaus Kreft

- [42] Lambda Expression: Einführung und Erklärung

<http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html>

28.12.2016

Kapitel Testing

- [43] Tutorialspoint: TestNG Test Suite

https://www.tutorialspoint.com/testng/testng_suite_test.htm

20.03.2017

- [44] Tutorialspoint: TestNG Group Test

https://www.tutorialspoint.com/testng/testng_group_test.htm

20.03.2017

- [45] Toolsqa TestNG Introduction

<http://toolsqa.com/selenium-webdriver/testng-introduction/>

20.03.2017

- [46] TestNG: Documentation

<http://testng.org/doc/documentation-main.html>

20.03.2017

- [47] Softwaretestingclass: Advantages of TestNG over Junit Framework
<http://www.softwaretestingclass.com/introduction-of-testng-framework-advantages-of-testng-over-junit-framework/>
03.04.2017

Kapitel Verwendete Tools

- [48] SonarQube: Dokumentation
<https://docs.sonarqube.org/display/SONAR/Documentation>
18.02.2017
- [49] VerifySoft: Statische Code Analyse
http://www.verifysoft.com/de_static_code_analysis.html
25.03.2017
- [50] ITWissen: Software Metrik Übersicht und genauere Erklärungen
<http://www.itwissen.info/Software-Metrik-software-metric.html>
31.03.2017
- [51] HS-Wismar: Statische Code-Analyse: Software Metriken
http://www.wi.hs-wismar.de/~cleve/vorl/projects/formal/ws11fern/04/Praesentation_Fuell_Gutsche_software-Metriken_FINAL.pdf
31.03.2017
- [52] Viva64: Dynamic Code Analysis
<https://www.viva64.com/en/t/0070/>
25.03.2017
- [53] Ba-horb: Codeanalyse
<http://www.ba-horb.de/fileadmin/media/it/studienprojekte/seminararbeiten/SWE3-071204/Codeanalyse.pdf>
25.03.2017
- [54] DZone: Why SonarQube?
<https://dzone.com/articles/why-sonarqube-1>
25.03.2017
- [55] Jenkins: Dokumentation
<https://jenkins.io/doc/>
13.02.2017
- [56] MartinFowler: Continuous Integration Erklärung
<https://www.martinfowler.com/articles/continuousIntegration.html>
31.03.2017
- [57] Wikipedia: Log4J
<https://de.wikipedia.org/wiki/Log4j>
18.02.2017
- [58] Thorsten-Horn: Logging mit Log4J
<http://www.thorsten-horn.de/techdocs/java-log4j.htm>
18.02.2017

-
- [59] Tutorialspoint: Log4J logging levels
https://www.tutorialspoint.com/log4j/log4j_logging_levels.htm
18.02.2017
 - [60] Apache: Ant
<http://ant.apache.org>
18.02.2017
 - [61] Git: Dokumentation
<https://git-scm.com>
18.02.2017

Sonstige

- [62] XML Icon
https://image.freepik.com/free-icon/xml-file-format-symbol_318-45423.jpg
03.04.2017