SQEE 2017/2018

**Change-Visualization**

Andreas Löffler

## 1. Installation

Aktuell befindet sich der Quellcode im Vitruv-Branch „visualization“. Dort liegt der wesentliche Code im Plugin „tools.vitruv.extension.changevisualization“. Alle Änderungen in Klassen ausserhalb meines Plugins habe ich mit ################-Zeilen eingerahmt und am Anfang der geänderten Klassen zusammengefasst. Dies ist nicht als Dauerzustand geplant, sondern zur leichteren Auffindung fürs Review.

Zusätzliche Änderungen innerhalb von Vitruv befinden sich hier:

- Plugin tools.vitruv.testutils
    - Klasse VitruviusAplicationTest
        - Hier werden alle Änderungen abgefangen die innerhalb von JUnit Tests entstehen. Der zusätzliche Code befindet sich am Anfang.
        - Darüber hinaus befindet sich eine zusätzliche Codezeile innerhalb von propagateChanges()
- Plugin tools.vitruv.framework.vsum
    - Interface ChangeListener
        - Über dessen Methode wird mit der Visualisierung kommuniziert
    - Klasse VirtualModelImpl.xtend
        - Hier werden alle Änderungen abgefangen die innerhalb von Live-Modellen entstehen. Der zusätzliche Code befindet sich auch hier am Anfang.
        - Darüber hinaus befindet sich eine zusätzliche Codezeile innerhalb von propagateChange(VitruviusChange change)

Es sind keine zusätzlichen Abhängigkeiten oder ähnliches an bereits bestehenden Plugins notwendig. Mein Plugin hat seine Abhängigkeiten bereits konfiguriert.

## 2. Verwendung

Mit Live Modellen:

Hier ist nichts weiter zu beachten oder zu konfigurieren. Einzig nach dem Start der Eclipse Instanz in der man schließlich das Modell bearbeitet muss die Überwachung der Änderungen noch durch Klick auf den „Change Visualization“ Menüeintrag gestartet werden (bevor man etwas am Modell ändert). Hier ist am Ende ein Tastaturkürzel geplant und für weitere Einstellungen am Plugin die Integration in die Eclipse Preferences. Eventuell ist dies bis zum Review schon umgesetzt.

Mit JUnit Tests:

Hier sind Änderungen am Code der zu überwachenden Tests erforderlich. Zuerst muss man mein Plugin in die Abhängigkeiten des Plugins in dem der Test sich befindet hinzufügen um Zugriff auf die Klassen zu erhalten.

Danach könnte man entweder einzelne Tests innerhalb einer Klasse überwachen bzw. ganze Testdateien oder ganze Testserien. Es ist nur sicherzustellen dass vor dem ersten Test die Überwachung aktiviert wird, und nach dem letzten Test eine Wartemethode aufgerufen wird.

Exemplarisch hier der Code der dies zur Überwachung einer einzelnen JUnit Testdatei sicherstellt:

```
@BeforeClass
public static void doYourOneTimeSetup() {
	addChangeListener(ChangeVisualization.getChangeListener());
}

@AfterClass
public static void doYourOneTimeTeardown() {
	ChangeVisualization.showUi(); //Visualization could be made visible after addChangeListener
	//but as changes get populated so fast, this does not make much sense
	ChangeVisualization.waitForFrameClosing();//To prevent vm from closing during junit tests
}
```

Insbesondere der Aufruf von ChangeVisualization.waitForFrameClosing() ist extrem wichtig, da sonst Eclipse die VM sofort nach Ende des letzten Tests beendet, man also keine Gelegenheit hat, die Änderungen auch tatsächlich zu sehen. Sobald man dann letztlich die Visualisierung schließt, wird die VM endgültig heruntergefahren.

## 3. Kurze Zusammenfassung der Konzepte

Zum leichteren Verständnis des Aufbaus der Pakete und Klassen innerhalb meines Plugins hier noch eine grobe Beschreibung deren Funktionalitäten und Zwecke. Ich beschränke mich hierbei auf die Visualisierung als Baum, da die Visualisierung als Tabelle von mir entweder gar nicht mehr eingecheckt wird oder nur als Vorbild für eventuell irgendwann mal zusätzliche umgesetzte Visualisierungen.

Als erstes beschreibe ich kurz die UI, da deren Kenntnis das Verständnis des Quellcodes erleichtert. Ein Screenshot findet sich weiter unten.

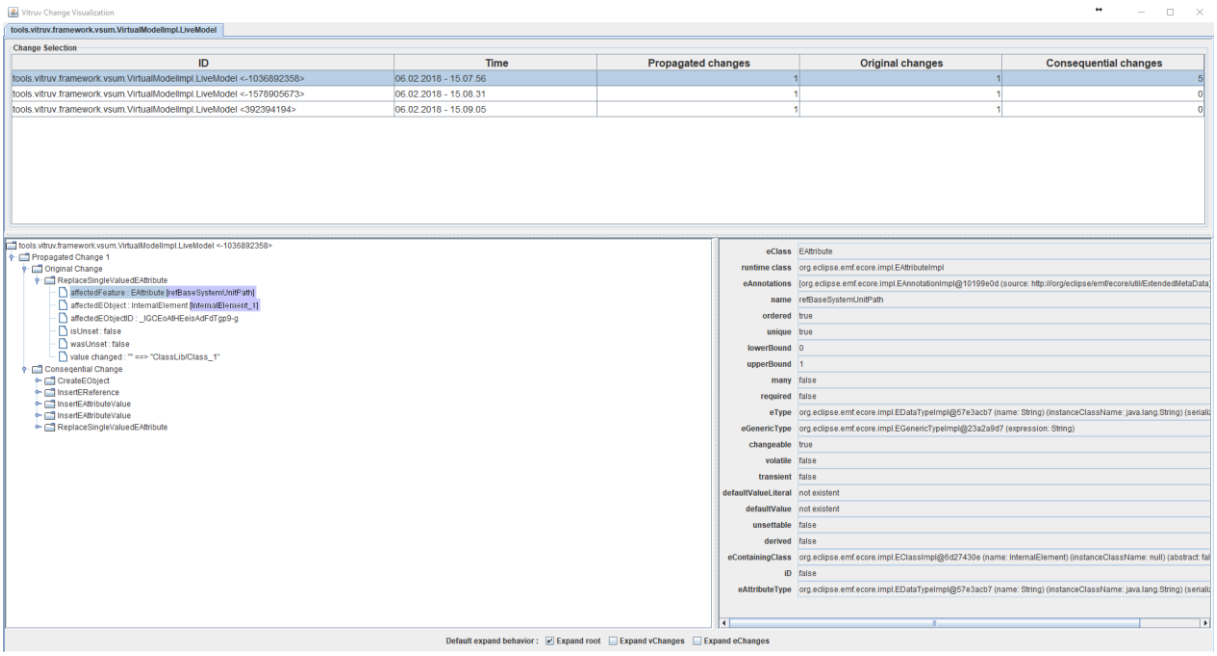
- Innerhalb des Fensters gibt es Tabs die jeweils alle Änderungen eines Modells, die im Laufe der Zeit erfasst werden, gruppiert. Im Beispiel ist ein Live-Modell, es wird nur ein Tab gezeigt. Der Name „Live Modell“ wird noch sinnvoller ausfallen, ist aktuell noch hardcodiert. Bei JUnit Tests wird der Name der Testmethode verwendet.
- Jeder Tab hat einen oberen Bereich, die „Change Selection“. Jedes registrierte Ereignis (ein Aufruf der propagateChange(s) Methode) wird hier zeitlich sortiert in einer Tabelle dargestellt. Aktuell mit groben Informationen wie der Anzahl der Propagated Changes,

Original Changes und Consequential Changes. Hier könnte nach Rücksprache alles dargestellt werden, was auf der Ebene eines propagation results Sinn macht.

Der untere Bereich ist zweigeteilt:

- Links ein Baum der die einzelnen Changes innerhalb des oben gewählten propagation results auflistet. Wenn im Zukunft vom Changes-Tree oder Baum gesprochen wird, ist dieser Bereich gemeint
- Rechts wird individuell für das gewählte Element im Baum eine ausführlichere Darstellung des gewählten Elements gezeigt. In Zukunft kurz Detailansicht genannt.

Ganz unten befindet sich noch eine Leiste mit allgemeinen Einstellungen. Aktuell nur die Default Aufklappverhalten für neue Results im Baum. Geplant ist hier noch eine Legende der Visualisierungs-Symbole, eine Suche und noch eine Reihe weiterer Gimmicks.



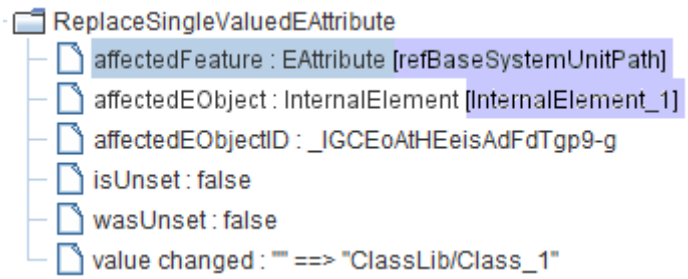

Grundlegendes Konzept ist die Visualisierung so generisch wie möglich zu halten, um alle aktuellen sowie alle zukünftig eventuell noch umgesetzten Change-Typen ohne Änderungen darstellen zu können. Dazu wird ein propagation result (Liste von PropagatedChanges) der Reihe nach durchlaufen und jeweils in Original Changes und Consequential Changes zerlegt. Jeder Change-Typ wird dann als Knoten im Baum dargestellt, standardmäßig mit seinem Klassennamen. (z.b. ReplaceSingleValuedEAttribute oder CreateEObject usw.).

Danach werden alle Informationen die zu einem solchen Change aus dem Modell ausgelesen werden können als Blätter unterhalb des Changes dargestellt. Genauer werden alle StructuralFeatures ausgelesen. Standardmäßig in der Form „Name des SF : toString() des zugehörigen Objekts“.

Sobald nun ein einzelnes SF eines Changes (ein Blatt) ausgewählt wird, wird je nach Klasse des Objektes eine Detailansicht erzeugt. Die ist für simple Features wie Strings oder Booleans unsinnig und wird aktuell nur für Instanzen von EObject erzeugt. Für diese werden dann wiederrum alle SF ausgelesen und dargestellt. Im ersten Screenshot für ein gewähltes EAttribute.

Dieses generische Vorgehen funktioniert potentiell für alles, kann aber an mehreren Stellen erweitert werden um spezifische, bessere Visualisierungen umzusetzen. Ich werde für einige existente Changes

und Objekte sinnvolle Visualisierungen implementieren, falls zukünftig mal Bedarf für individuelle Visualisierungen neuer Changes oder Objekte existiert kann dies sehr einfach nachgerüstet werden.




Die Stellen für individuelle Visualisierungen sind folgende:

- Change-Knoten (im Beispiel ReplaceSingleValuedEAttribute)
    - Aktuell noch nicht vollständig implementiert, aber hier soll es möglich sein die wesentlichen Infos eines Changes zu kodieren ohne ihn aufklappen zu müssen. In dem Beispiel ca. so: ReplaceSingleValuedEAttribute / EAttribute 'refBaseSystemUnitPath' : „“ ➔ „ClassLib/Class_1”
- Einzelnes Structural Feature
    - Je nach Klasse des Objekt verschieden. Im Beispiel für Boolean einfach „false“ oder „true“, für ein EObject jedoch der Klassenname sowie in eckigen Klammern und farbig hervorgehoben der Name/EntityName des betroffenen EObjects.
    - Wenn für die Klasse des Objekts ein spezifischer Detail-Renderer vorliegt, werden dann Details dazu rechts dargestellt. Diese sind nicht auf Text beschränkt, sondern können alles sein, was man in einen JPanel packen kann. Im Extremfall sogar Bilder oder Videos, falls das mal irgendwie Sinn ergibt.
- Mehrere StructuralFeatures in Kombination
    - Es können individuelle Visualisierungen für Kombinationen von SF registriert werden, im Beispiel für die oldValue/newValue Kombination. Daraus wird eine Zeile mit value changed : „“ ➔ „ClassLib/Class_1“

Beim ersten Commit fürs Review wird es noch nicht vorliegen, aber bis zum Review selbst habe ich noch vor die verschiedenen Klassen von Changes (ExistenceChanges, RootChanges, ReferenceChanges...) durch individuelle Symbole im Baum farblich hervorzuheben.

Nun zu den Paketen:

- tools.vitruv.extensions.changevisualization
    - Hier liegt der Einstiegpunkt (ChangeVisualization) sowie die Basisklasse für Informationen zu einem propagation result (ChangeDataSet). Es wird beim Abfangen von propagation results immer eine Kopie aller wesentlichen Informationen erstellt, so dass die Original Objekte nicht mehr notwendig sind. Einerseits um die GarbageCollection nicht zu behindern da so deutlich weniger gespeichert werden muss, andererseits um jegliche Seiteneffekte auszuschließen. Es müssen ja alle Werte die genau zum Ereignis vorlagen zuverlässig dauerhaft vorliegen, und nicht durch Verwendung der echten Objekte, die sich gegebenenfalls ändern, auch ändern.
- tools.vitruv.extensions.changevisualization.handler

    - hier liegt alles Eclipse Plugin spezifische. Integration in Preferences… Aktuell nur das was beim Aufruf des Menüeintrags passieren soll
- tools.vitruv.extensions.changevisualization.table
    - Alles zum Visualisieren als Tabelle. Wie gesagt eventuell nicht eingecheckt oder wenn doch, dann nicht review-relevant.
- tools.vitruv.extensions.changevisualization.tree
    - Alles zum Visualisieren als Baum
- tools.vitruv.extensions.changevisualization.tree.decoder
    - Hier sind die Klassen die die spezifische Visualisierung einzelner Changes oder Objekte im Baum und der Detailansicht umsetzen
- tools.vitruv.extensions.changevisualization.ui
    - Alle Klassen die zur generischen Visualisierung gehören, also nicht Baum oder Tabellenspezifisch sind
- tools.vitruv.extensions.changevisualization.utils
    - Hilfsklassen die an vielen Stellen nützlich sind