Modularisierung der Reactions-Sprache

Handbuch / Praktikumsbericht
Praktikum Software Quality Engineering mit Eclipse WS 17/18
Lukas Hennig, 1699498

Dieses Handbuch dient den Reviewern als Vorbereitung auf das Code-Review. Es umfasst eine kurze Übersicht über die Reactions-Sprache und relevante Aspekte der bisherigen Umsetzung, die Vorstellung des bearbeiteten Problems, die Spezifikation in Form von Anforderungen und Entwurfsentscheidungen, sowie eine Beschreibung der Umsetzung und getätigten Änderungen an der Code-Struktur.

Es sind auch Kapitel und Details enthalten, die für das Code-Review weniger relevant sein dürften. Als Reviewer sollte man sich auf folgende Inhalte beschränken können:

- Als Einleitung: Kapitel 1, sowie der erste und letzte Absatz von Kapitel 3.1.
- Für eine Übersicht über das zu implementierende Verhalten: Grob Kapitel 3.2 und 3.3, sowie grob die Grammatik-Ergänzungen aus Kapitel 3.4.
- Für eine Beschreibung der Umsetzung: Kapitel 3.5
- Github Links und Hinweise zur Installation: Kapitel 4

Inhaltsübersicht

1. Kurze Beschreibung der Reactions-Sprache und der bisherigen Umsetzung

Die Reactions-Sprache ist Teil des Vitruvius Frameworks (kurz: Vitruv). Vitruv ermöglicht die automatisierte, änderungsgetriebene Konsistenzerhaltung verschiedener Modelle.
Die Reactions-Sprache ist eine Xtext-basierte domänenspezifische Sprache (DSL) mit der sich sogenannte "Reaktionen" (englisch: "reactions") definieren lassen. Reaktionen reagieren auf Änderungen in einem Quellmodell und führen dann entsprechende Änderungen an einem Zielmodell aus.

Zusätzlich lassen sich sogenannte "Routinen" (englisch: "routines") definieren. Diese umfassen ebenfalls Anweisungen zum Modifizieren des Zielmodells und lassen sich sowohl von Reaktionen als auch von anderen Routinen aufrufen und ausführen. Sie eignen sich daher zum Beispiel zum Auslagern von gemeinsamen Anweisungen, die von mehreren Reaktionen verwendet werden.

Die Reactions-Sprache verwendet Dateien mit der ".reactions"-Endung. Eine Reactions-Datei kann ein oder mehrere sogenannter "Segmente" (englisch: "reactions segments") beinhalten, von denen jedes wiederum die jeweils darin definierten Reaktionen und Routinen umfasst. Jedes Segment spezifiziert außerdem das Paar von Quell- und Ziel-Metamodell, auf denen die enthaltenen

Reaktionen operieren.

Jedes Segment, jede Reaktion und jede Routine hat eine Namen, der innerhalb des Segments eindeutig ist, und daher zum Referenzieren genutzt werden kann. Im Moment können Reaktionen und Routinen nur Routinen innerhalb des gleichen Segments aufrufen.

Die Reaktionen und Routinen eines Eclipse-Projekts werden nach Java-Code kompiliert. Die durch die Codegenerierung entstehende Package-Struktur ist wie folgt:

- Package: mir.reactions.reactions<MetamodelPairName>.<SegmentName>
  - Für jede Reaktion: <ReactionName>Reaction.java
  - <MetamodelPairName>Executor.java
- Package: mir.reactions
  - <MetamodelPairName>ChangePropagationSpecification.java
- Package: mir.routines<SegmentName>
  - Für jede Routine: <RoutineName>Routine.java
  - RoutinesFacade.java

Für jede Reaktion und jede Routine wird eine entsprechende Java Klasse generiert, die die Ausführung der Reaktion bzw. Routine implementiert.

Außerdem wird für jedes Segment eine "Executor"-Klasse generiert, die Modelländerungen von Vitruv an die Reaktionen des Segments weiterleitet.

Und für jweils alle Segmente, die auf dem gleichen Metamodell-Paar operieren, wird eine einzelne "ChangePropagationSpecification"-Klasse generiert. Eine Vitruv-Anwendung übergibt eine Instanz dieser Klasse an Vitruv und Vitruv informiert diese Instanz dann über erfasste Modelländerungen, die dieses Metamodell-Paar betreffen. Die ChangePropagationSpecification-Klasse leitet diese Modelländerungen wiederum an die entsprechenden Executors, und somit an alle Reaktionen weiter, die auf diesem Metamodell-Paar operieren.

Des weiteren wird für jedes Segment eine "RoutinesFacade"-Klasse generiert. Diese beinhaltet Methoden mit denen sich die Routinen des Segments aufrufen lassen. Über den "extension"-Mechanismus von Xtext werden die Methoden dieser Routinenfacade innerhalb der Code-Blöcke innerhalb der Reaktionen und Routinen verfügbar gemacht.

Immer wenn eine Reaktion oder eine Routine eine andere Routine im gleichen Segment aufrufen will, ruft sie also zunächst die entsprechende Methode der Routinenfacade auf, und diese erstellt dann eine Instanz der entsprechenden Routinen-Klasse und bringt diese zur Ausführung.

2. Involvierte Projekte

Sämtliche hier aufgelisteten Projekte sind Teil des Vitruv Frameworks und lassen sich in dessen Repository finden.

- **tools.vitruv.dsls.reactions:**
  - Beinhaltet die Grammatik der Reactions-Sprache und den Codegenerator.
- **tools.vitruv.dsls.mirbase:**
  - Beinhaltet die Grammatik der Sprache "MirBase", auf die die Reactions-Sprache aufbaut.
- **tools.vitruv.dsls.reactions.ui:**
  - Beinhaltet UI und Editor spezifische Ergänzungen, die für das Schreiben von Reactions-Dateien im Eclipse-Editor nützlich sind.
- **tools.vitruv.dsls.reactions.tests:**

    - Beinhaltet Testfälle für die Reactions-Sprache.
- **tools.vitruv.extensions.dslruntime.reactions:**
    - Beinhaltet Schnittstellen und Basisklassen für die generierten Reaktionen, Routinen, Routinenfacaden und Executors, und ist für die Abbildung von Reaktionen zu entsprechenden ChangePropagationSpecifications zuständig, die von Vitruv verwendet werden.

Zusätzlich gibt es eine Reihe von Projekten innerhalb des Vitruv Frameworks, die Reactions-Dateien enthalten:

- tools.vitruv.demo.applications.familiespersons.families2persons
- tools.vitruv.demo.applications.familiespersons.persons2amilies
- tools.vitruv.dsls.mappings.tests.addressesXrecipients

Der Code für die Reaktionen dieser Projekte wird neu generiert werden müssen, wenn die vorgeschlagenen Änderungen für die Modularisierung in das Vitruv-Projekt übernommen werden.

3. Modularisierung

3.1. Anwendungsfall

Im Moment kann man keine Reaktionen oder Routinen aus anderen Reactions-Dateien (oder generell aus anderen Segmenten, selbst innerhalb der gleichen Datei) referenzieren. Daher können gemeinsame Konsistenzregeln aktuell nicht zwischen Projekten wiederverwendet werden, ohne die entsprechenden Reaktionen und Routinen in die einzelnen Projekte zu kopieren.

Die folgenden Projekte wurden auf Gemeinsamkeiten untersucht und auf ihnen sollen die Modularisierunganpassungen exemplarisch angewandt werden. Die Projekte finden sich im "Vitruv-Applications-ComponentBasedSystems"-Repository:

- tools.vitruv.applications.pcmjava.pojotransformations.pcm2java
- tools.vitruv.applications.pcmjava.ejbtransformations.pcm2java
- tools.vitruv.applications.pcmjava.depinjecttransformations.pcm2java

Jedes dieser Projekte enthält eine Reactions-Datei, die sich ihren Inhalt größtenteils mit denen der anderen Projekte teilt. Nur wenige Reaktionen und Routinen wurden in den einzelnen Projekten leicht angepasst, entfernt, oder neu hinzugefügt.

Die Idee ist es, die gemeinsamen Reaktionen und Routinen dieser Projekte in eine separate Reactions-Datei in einem separaten Projekt (tools.vitruv.applications.pcmjava.common) zu schieben und die jeweiligen Projekte dann mittels dem ergänzten Modularisierungsmechanismus auf diese gemeinsame Reactions-Datei referenzieren zu lassen.

3.2. Anforderungen

Aus der Untersuchung auf Gemeinsamkeiten zwischen den oben aufgelisteten Projekten ergaben sich die folgenden Anforderungen an den zu entwerfenden Modularisierungsmechanismus:

1. **Gemeinsame Routinen und Reaktionen auslagern:** Aufgrund der vielen gemeinsamen Reaktionen macht es Sinn, nicht nur gemeinsame Routinen auszulagern, sondern auch

gemeinsame Reaktionen.

2. **Überschreibbare Routinen:** Es kommt mehrmals vor, dass gemeinsame Reaktionen und Routinen eine andere Routine aufrufen, die im jeweiligen Projekt jedoch modifiziert wurde. Die modifizierte Routine kann nicht ohne weiteres in das gemeinsame Projekt übernommen werden. Damit können jedoch auch die gemeinsamen Reaktionen und Routinen, die diese modifizierte Routine referenzieren, nicht ausgelagert werden.

   Um nun dennoch diese gemeinsamen Reaktionen und Routinen auslagern zu können, bietet es sich an, eine Art "Overriding"-Mechanismus umzusetzen: Damit könnte eine leere, oder Standard-Version der modifizierten Routine in das gemeinsame Projekt aufgenommen werden, die dann dort von den anderen gemeinsamen Reaktionen und Routinen referenziert werden kann. Und die jeweiligen Projekte müssten dann in der Lage sein, das Verhalten dieser Routine entsprechend zu überschreiben.

3. **Routinen mittels einfachen Namen aufrufen:** Auf Wunsch hin sollen referenzierte, externe Routinen auf die gleiche Weise benutzbar sein, wie wenn diese lokal im gleichen Segment definiert worden wären. Das bedeutet konkret: Sofern keine Namenskonflikte auftreten, soll es keine Notwendigkeit geben, externe Routinen beim Aufruf mittels irgendeiner Form von qualifizierten Namen zu referenzieren.

   Unter Anbetracht des obigen Anwendungsfalls, sowie ähnlicher Anwendungsfälle in anderen Vitruv-Anwendungen, werden im Moment keine großen Hierarchien beim direkten und transitiven Referenzieren von anderen Segmenten erwartet. Dementsprechend können Namenskonflikte in den meisten erwarteten Fällen durch einfaches Umbenennen der betroffenen Elemente aufgelöst werden. Routinen standardmäßig mittels einfachen Namen zu referenzieren erscheint also angemessen zu sein.

3.3. Entwurfsentscheidungen

Ich habe mich dazu entschlossen, die Modularisierung mittels eines Import-Mechanismus umzusetzen: Man wird innerhalb eines Segments andere Segmente über deren Namen "importieren" können.

Dieses Kapitel ergänzt die obigen Anforderungen um zusätzliche Entwurfsentscheidungen, die bezüglich der Funktionsweise dieses Import-Mechanismus getroffen wurden.

1. **Importieren von Segmenten:** Innerhalb eines Segments können andere Segmente über deren Namen "importiert" werden. Sobald ein Segment importiert wurde, können dessen Routinen aufgerufen werden und dessen Reaktionen ergänzen die eigenen Reaktionen.
2. **Überschreibbare Reaktionen:** Es bietet sich an, einen Overriding-Mechanismus auch für Reaktionen umzusetzen, mittels dem sich das Verhalten von importierten Reaktionen umdefinieren lässt. Reaktionen müssen dann nicht zwangsweise immer ihr Verhalten in eine überschreibbare Routine auslagern, um "importier-freundlich" zu sein.
3. **Importieren von nur Routinen:** Anstatt immer sowohl Reaktionen als auch Routinen eines Segments zu importieren, wird es auch möglich sein, nur die Routinen zu importieren.
4. **Routinen mit qualifizierten Namen:** Für den Fall, dass Namenskonflikte mit oder zwischen importierten Routinen auftreten, die nicht durch Umbenennungen gelöst werden können, wird es die Möglichkeit geben, importierte Routinen auch mittels qualifizierten Namen aufzurufen.

   Der qualifizierte Name setzt sich aus dem Namen des importierten Segments (genauer: dem

Importpfad zu dem importierten Segment; dazu später mehr) und dem Namen der Routine zusammen.

Das kann besonders dann nützlich sein, wenn Namenskonflikte zwischen importierten Segmenten auftreten und man keine Kontrolle über diese hat und daher nicht einfach Routinen umbenennen kann.

Auf Wunsch hin wird die Verwendung von Routinen über qualifizierte Namen standardmäßig nicht verfügbar sein. Stattdessen wird es ein optionales Flag bei der Deklaration des Imports geben, mit dem man zwischen der Verwendung von einfachen Namen und qualifizierten Namen umschalten kann.

Bezüglich dem Umgang mit transitiven Importen wurden die folgenden Entscheidungen getroffen:

5. **Keine zyklischen Imports:** Ein Segment kann sich weder direkt noch transitiv selber importieren.
6. **Kein mehrmaliges Importieren von Reaktionen:** Es kann innerhalb der gesamten Importhierarchie (= Baum aller Importe, einschließlich transitiver Importe) nur einen einzigen Import für die Reaktionen eines Segments geben.

   Diese Einschränkung garantiert, dass es innerhalb der Importhierarchie jeweils genau einen Pfad vom Wurzel-Segment zum Segment einer importierten Reaktion gibt. Und damit liegen auch alle Überschreibungen, die diese Reaktion betreffen (Überschreibungen der Reaktion selber, und Überschreibungen von Routinen, die von dieser Reaktion verwendet werden), auf diesem einen Pfad. (Damit wird das "Diamond-Inheritance-Problem" für importierte Reaktionen vermieden.)
7. **Bezüglich Namenskonflikten zwischen importierten Reaktionen:** Da die Namen von Reaktionen innerhalb eines Segments bereits eindeutig sind, und Reaktionen aktuell nur bei der Deklaration von Überschreibungen referenziert werden und dort immer zusammen mit dem Namen ihres Segments auftreten (siehe Kapitel "3.4. Änderungen an der Reactions-Sprache"), gibt es aktuell keinen Grund dafür, dass die Namen von Reaktionen auch über Importe hinweg eindeutig sein müssen.

Bezüglich transitiv importierten Routinen wurden die folgenden Entscheidungen getroffen:

8. **Bezüglich Namenskonflikten aufgrund von transitiv importierten Routinen:** Jegliche Namenskonflikte aufgrund von Routinen, die transitiv importiert wurden, wirken sich auf das importierte Segment als Ganzes aus: Es wird keine Möglichkeit geben, transitive Importe auszuschließen. Das betrifft gleichermaßen qualifizierte Routinen-Namen, die durch Importe entstehen, die qualifizierte Namen verwenden, als auch einfache Routinen-Namen durch Importe ohne qualifizierten Namen.
9. **Bezüglich überschriebenen, transitiven Routinen:** Alle Aufrufe von transitiv importierten Routinen werden, unabhängig von dem zum Aufruf verwendeten Namen, die Überschreibungen von Routinen aus den in der Importhierarchie dazwischen liegenden Segmenten berücksichtigen.
10. **Bezüglich Importieren eines Segments ohne qualifizierten Namen:** Die importierten Routinen werden wie folgt aufrufbar sein:
    1. Die Routinen des importierten Segments, sowie alle Routinen, die **transitiv ohne qualifizierten Namen** importiert wurden, werden über ihre einfachen Namen verwendbar sein. (siehe 3.2.3.)
    2. Alle Routinen, die **transitiv mit qualifizierten Namen** importiert wurden, werden über die gleichen qualifizierten Namen auch innerhalb des importierenden Segments verwendbar sein.

11. **Bezüglich Importieren eines Segments mit qualifizierten Namen:** Die importierten Routinen werden wie folgt aufrufbar sein:
    1. Die Routinen des importierten Segments, sowie alle Routinen, die **transitiv ohne qualifizierten Namen** importiert wurden, werden über den qualifizierten Namen verwendebar sein, der sich aus dem Namen des importierten Segments und dem Namen der Routine zusammensetzt: <ImportedSegmentName>.<RoutineName>
    2. Alle Routinen die **transitiv mit qualifizierten Namen** importiert wurden, werden über den qualifizierten Namen verwendebar sein, der sich aus dem Namen des importierten Segments und dem qualifizierten Namen innerhalb des importierten Segments zusammensetzt:

       <ImportedSegmentName>(.<TransitiveImportedSegmentName>)+.<RoutineName>

       Insgesamt beschreibt der qualifizierte Namen damit den Pfad innerhalb der Importhierarchie bis zum ersten Segment, dass entweder die Routine ohne qualifizierten Namen importiert, oder die ursprüngliche Deklaration der Routine enthält.
12. **Mehrmaliges, unabhängiges Importieren von Routinen innerhalb der Importhierarchie:** Unter Berücksichtigung des Szenarios, dass bestimmte, häufig autretende Routinen in ein gemeinsames Segment ausgelagert wurden und nun von einer Reihe verschiedener Segmente verwendet werden, wird es möglich sein, dass innerhalb der Importhierarchie mehrere Segmente die Routinen des gleichen Segments importieren, überschreiben und verwenden können.

    Um unerwartete Verhaltensänderungen in den Reaktionen und Routinen anderer Segmente zu vermeiden, werden sich die Überschreibungen von Routinen nur auf Routinenaufrufe auswirken, die entweder direkt vom überschreibenden Segment ausgehen, oder von Segmenten, die das überschreibende Segment importieren und darüber die überschriebenen Routinen aufrufen.
13. **Überschreiben von transitiv importierten Routinen:** Unter Berücksichtigung des Szenarios, dass man hingegen sehr wohl das Verhalten von Reaktionen und Routinen anderer Segmente verändern will, die in der Importhierarchie tiefer als eine Ebene liegen, wird es möglich sein, auch transitiv importierte Routinen zu überschreiben.

    Dazu wird man bei der Deklaration der Überschreibung der Routine den Importpfad angeben müssen, der vom aktuellen Segment innerhalb der Importhierarchie zu dem Segment führt, aus dem man die Routine überschreiben will.

    Die Überschreibung wirkt sich dann entsprechend nur auf Routinenaufrufe entlang dieses Importpfades aus.

    Beispiel: Importhierarchie A->B, B->C, A->C, A->D, D->C, "->" repräsentiert hier einen Routinen-Import mit qualifizierten Namen: Sowohl A, B und D importieren die Routinen von C. Und A importiert außerdem B und D.
    A überschreibt nun eine Routine unter Angabe des Importpfades "B.C". Die Überschreibung wirkt sich dann nur auf Routinenaufrufe von A nach B.C, von B nach C und von A nach B aus (falls daraus Aufrufe von B nach C resultieren).
    Direkte Aufrufe von A nach C, oder Aufrufe von A nach D und von D nach C werden hingegen nicht beeinflusst.
14. **Überschreiben von transitiv importierten Reaktionen:** Transitiv importierte Reaktionen werden genau wie direkt importierte Reaktionen überschrieben: Durch die Angabe des Segments in dem sich die zu überschreibende Reaktion befindet.

3.4. Änderungen an der Reactions-Sprache

Grammatik / Syntax

Innerhalb von "ReactionsSegment" kann man nun, nach der Definition des Quell- und Ziel-Metamodells und bevor man Reaktionen oder Routinen deklariert, beliebig viele "ReactionsImports" definieren. Die Grammatik für einen ReactionsImport sieht wie folgt aus:

'import' (routinesOnly?='routines')? importedReactionsSegment=[ReactionsSegment] (useQualifiedNames?='using qualified names')?;

- Der Import wird mit dem Schlüsselwort "import" eingeleitet.
- Mit dem optionalen Schlüsselwort "routines" kann man auswählen, ob man sowohl Reaktionen als auch Routinen, oder nur Routinen importieren möchte. (siehe 3.3.3.)
- "importedReactionsSegment" ist eine Cross-Referenz auf ein anderes ReactionsSegment. Das Linking erfolgt über den Namen des Segments. Mittels Scoping werden hierfür die Segmente bestimmt, die vom aktuellen Projekt aus sichtbar sind. Für Java-Projekte benutzt dieser Mechanismus zum Beispiel den Classpath.
- Mit der optionalen Schlüsselphrase "using qualified names" wird festgelegt, ob importierte Routinen anstatt mit einfachen Namen, mit qualifizierten Namen importiert werden. (siehe 3.3.4., sowie 3.3.10. und 3.3.11. bezüglich der Bedeutung für transitive Imports)

Reaktionen enthalten nun zusätzlich eine optionale Cross-Referenz, mit der das Segment angegeben werden kann, dessen Reaktion mit dem angegebenen Namen überschrieben werden soll. Die Syntax dafür sieht wie folgt aus:

reaction <OverriddenSegment>::<OverriddenReactionName> {..}

Routinen enthalten nun ebenfalls ein zusätzliches optionales Element, allerdings zum Angeben des Importpfades, das vom aktuellen Segment aus über die Importhierarchie zu dem Segment führt, dessen Routine mit dem angegebenen Namen man überschreiben möchte. Die Syntax dafür sieht wie folgt aus:

routine (<ImportedSegment>.)*<OverriddenSegment>::<OverriddenRoutineName> {..}

In der Grammatik ist der Importpfad als Kette von Importpfaden umgesetzt, in der jedes Element eine Cross-Referenz auf sein eigenes Segment enthält, sowie eine Referenz zum vorherigen Importpfad-Element in der Kette. Diese Kettenstruktur ist beim Scoping der einzelnen Segmente im Importpfad nützlich: Mit der Referenz auf das vorherige Segment im Importpfad kann man relativ einfach für jede Stelle im Importpfad die Segmente bestimmen, die an dieser Stelle möglich sind.

Validierungsregeln

Im Zuge der Modularisierungsanpassungen wurden eine Reihe von Validierungsregeln hinzugefügt, die entweder in einem Fehler oder einer Warnung resultieren:

- Fehler: Gleiche Segment-Namen innerhalb einer Reactions-Datei.
- Fehler: Segment-Namen können nicht länger mit "_" beginnen. ("_" wird in der Routinenfacade für interne Methoden verwendet und würde daher kollidieren)
- Warnung: Segment-Namen sollten mit Kleinbuchstaben beginnen.
- Warnung: Gleiche Segment-Namen innerhalb aller sichtbaren, externen Segmente.
- Fehler: Importierte Segmente müssen das gleiche Metamodell-Paar benutzen.
- Fehler: Mehrmaliges Importieren des gleichen Segments innerhalb eines Segments.
- Fehler: Keine zyklische Importe: Ein Segment kann sich nicht direkt oder transitiv selber

importieren.
- Fehler: Mehrmals importierte Reaktionen eines Segments innerhalb der Importhierarchie.
- Fehler: Namenskonflikte mit und zwischen importierten Routinen, sowohl für einfache Namen, als auch qualifizierte Namen. Schließt die Namen von transitiv importierte Routinen mit ein.
- Fehler: Routinen-Namen können nicht länger mit "_" beginnen. ("_" wird in der Routinenfacade für interne Methoden verwendet und würde daher kollidieren)
- Fehler: Importpfad beim Überschreiben von Routinen ist unvollständig.
- Fehler: Das Segment für den angegebenen Importpfad beim Überschreiben von Routinen kann nicht gefunden werden.
- Fehler: Es kann keine überschriebene Routine mit dem angegebenen Namen im angegebenen Segment gefunden werden.
- Fehler: Die Parameter der überschreibenden Routine stimmen nicht mit den Parametern der überschriebenen Routine überein.
- Fehler: Es kann keine überschriebene Reaktion mit dem angegebenen Namen im angegebenen Segment gefunden werden.

3.5. Implementierung und Code-Struktur

Importierte Routinen

Bisher gab es nur die Möglichkeit Routinen im selben Segment aufzurufen. In Kapitel 1 wurde dafür bereits grob die Funktionsweise der generierten RoutinesFacade-Klasse vorgestellt.

Beim Importieren von Routinen aus anderen Segmenten hat man nun die Möglichkeit, die Routinen entweder mit, oder ohne qualifizierten Namen zu importieren. Unter der Berücksichtigung von transitiv importierten Segmenten ergeben sich die folgenden drei Gruppen:

- **Gruppe der ohne qualifizierten Namen importierten Segmente:** Segmente, deren Routinen ohne qualifizierten Namen importiert wurden, und Segmente, die transitiv ohne qualifizierten Namen von anderen Segmenten aus dieser Gruppe importiert wurden.
- **Gruppe der mit qualifizierten Namen importierten Segmente:** Segmente, deren Routinen mit qualifizierten Namen importiert wurden, und Segmente, die transitiv mit qualifizierten Namen von Segmenten aus der Gruppe der ohne qualifizierten Namen importierten Segmente importiert wurden.
- **Gruppe aller anderen transitiv importierten Segmente:** Deren Routinen lassen sich entweder über den qualifizierten Namen eines Segments aus der Gruppe der mit qualifizierten Namen importierten Segmente aufrufen, oder über einen längeren qualifizierten Namen, der mehr als ein Segment enthält.

Für Routinen aus der Gruppe der ohne qualifizierten Namen importierten Segmente werden in der eigenen Routinenfacade entsprechende Routinen-Methoden generiert. Dies geschieht auf die gleiche Art und Weise, wie es auch für die lokal im gleichen Segment deklarierten Routinen der Fall ist. Damit lassen sich diese importierten Routinen also genau wie lokal deklarierte Routinen über ihren einfachen Namen aufrufen.

Für jedes Segment aus der Gruppe der mit qualifizierten Namen importierten Segmente wird in der eigenen Routinenfacade ein öffentliches Feld generiert, das als Typ die Routinenfacaden-Klasse des importierten Segments verwendet und genau wie das importierte Segment benannt ist. Über dieses

Feld können dann die Routinen aus dem importierten Segment über qualifizierte Namen aufgerufen werden.

Für Segmente aus der Gruppe aller anderen transitiv importierten Segmente muss nichts weiter unternommen werden: Deren Routinen sind entweder Teil der Routinenfacaden der Segmente aus der Gruppe der mit qualifizierten Namen importierten Segmente, oder ihre Routinenfacade lässt sich über die öffentlichen Felder der Routinenfacaden dieser Segmente erreichen.

Überschriebene Routinen

Erweiterte Routinenfacaden

Für jedes Segment, aus dem Routinen überschrieben werden, wird eine erweiterte Routinenfacade im Package "mir.routines..[<ImportPfadSegment>.]*<ÜberschriebenesSegment>" generiert. In diesem Package landen auch die Routinen-Klassen für alle aus diesem Segment überschriebenen Routinen. Für das Generieren der erweiterten Routinenfacade ist die neue Klasse "OverriddenRoutinesFacadeClassGenerator" zuständig.

Die erweiterte Routinenfacade erbt von der erweiterten Routinenfacade des nächsten Segments entlang des angegebenen Importpfades, das ebenfalls Routinen des spezifizierten Segments überschreibt. Falls kein solches Segment existiert, erbt sie von der originalen Routinenfacade.

Durch die Vererbung beinhaltet die erweiterte Routinenfacade bereits alle Methoden für Routinen des überschriebenen Segments, samt der Ersetzungen durch überschriebene Routinen entlang des Importpfades.

In der erweiterten Routinenfacade werden Methoden für die überschriebenen Routinen generiert, die beim Aufruf die überschreibende Routine ausführen. Diese Methoden überschreiben die entsprechenden Routinen-Methoden der geerbten Routinenfacade.

RoutinesFacadesProvider

Bisher haben Reaktionen und Routinen immer für sich selber eine eigene Instanz der Routinenfacade des eigenen Segments erstellt und diese dann für ihre Routinenaufrufe verwendet. Damit die von importierten Reaktionen und Routinen verwendete Routinenfacade mit entsprechenden erweiterten Routinenfacaden ausgetauscht werden kann, erstellen Reaktionen und Routinen ihre Routinenfacaden nicht länger selber, sondern bekommen diese über den Konstruktor vorgegeben.

Für das Erstellen der Routinefacaden ist eine neue Klasse "RoutinesFacadesProvider" zuständig: Für jedes Segment wird eine RoutinesFacadesProvider-Klasse generiert, die für jeden gültigen Importpfad eine Routinenfacade für das entsprechende Segment an dieser Stelle in der Importhierarchie liefern kann. Wenn die Routinen eines importierten Segments von einem anderen Segment entlang des Importpfades überschrieben werden, dann kann hier die entsprechende erweiterte Routinenfacade anstatt der originalen Routinenfacade zurückgegeben werden.

Eine Instanz des eigenen RoutinesFacadesProvider wird vom Executor jedes Segments erstellt. Der Executor benutzt den RoutinesFacadesProvider zum Anfragen der Routinenfacaden für die zu instanziierenden Reaktionen.

Immer wenn eine Routinenfacade vom RoutinesFacadesProvider instanziiert wird, bekommt sie eine Referenz auf den RoutinesFacadesProvider und den eigenen Importpfad übergeben. Die Routinenfacade kann damit dann den RoutinesFacadesProvider nach den Routinenfacaden für die aufgerufenen Routinen anfragen, sowie für die Felder, in denen die Routinenfacaden für die mit qualifizierten Namen importierten Segmente hinterlegt sind (siehe 3.5. "Importierte Routinen"). Der

eigene Importpfad wird benötigt, damit für das gleiche Segment, aber an unterschiedlichen Importpfaden in der Importhierarchie, beim RoutinesFacadesProvider nach der richtigen Routinenfacade angefragt werden kann (siehe 3.3.12.).

Um nicht für jede Reaktion und jeden Routinenaufruf die Routinenfacaden für alle Segmente in der Importhierarchie neu erstellen zu müssen, werden die erstellten Routinenfacaden vom RoutinesFacadesProvider gecacht.

Allerdings hatten Routinenfacaden bisher zwei Zustandsinformationen, die sie von der Reaktion oder der Routine bei ihrer Instanziierung erhalten haben:

- ReactionExecutionState: Wird einmalig vor jeder Ausführung einer Reaktion erzeugt und dann immer weiter an die aufgerufenen Routinen übergeben.
- Aufrufhierarchie des Aufrufers: Die Reaktion oder die Routine, die die Routinenfacade erstellt hat und verwendet, hat für Debugging-Zwecke Informationen über die bisherige Aufrufhierarchie. Diese Information wird an aufgerufene Routinen weitergegeben, die damit dann die Aufrufhierarchie fortsetzen können.

Da sich nun aber alle Reaktionen und Routinen die gleichen Instanzen der Routinenfacaden teilen, muss vor jeder Verwendung die bisherige Aufrufhierarchie gesichert, die eigene Aufrufhierarchie gesetzt, und anschließend die zuvor gesichterte Aurfufhierarchie wiederhergestellt werden (nötig falls der vorherige Aufrufer noch weitere Routinen aufruft). Das wurde in den Klassen "AbstractRepairRoutineRealization" und "AbstractReactionRealization" im DSL-Runtime-Projekt implementiert.

Außerdem teilen sich alle Routinenfacaden nun ihre Zustandsinformationen: Damit sind unnötig redundante Kopien der Zustandsinformationen nicht länger über alle Routinenfacaden in der Importhierarchie verteilt, und die Zustandsinformationen können leichter wieder "aufgeräumt" werden, nachdem die Ausführung einer Reaktion beendet ist.

Importierte Reaktionen:

Die Importhierarchie wird nach importierten Reaktionen durchsucht, und diese werden in der Executor-Klasse zusammen mit den eigenen Reaktionen instanziiert und dann bei Aufrufen von Vitruv, aufgrund von Modelländerungen, auch zusammen mit den eigenen Reaktionen ausgeführt.

Überschriebene Reaktionen:

Die Importhierarchie wird mittels Tiefensuche nach Reaktionen durchsucht und bei jedem Backtracking werden die eigenen Reaktionen des aktuell betrachteten Segments mit in das Ergebnis aufgenommen und überschriebene Reaktionen werden im bisherigen Ergebnis ersetzt.
Dadruch landen beim Executor dann alle importierten Reaktionen und überschriebene Reaktionen wurden in der richtigen Reihenfolge ersetzt.

3.6. Übersicht über alle geänderten und ergänzten Dateien

Hier findet sich eine Auflistung (in keiner speziellen Reihenfolge) aller geänderten und ergänzten Dateien, gruppiert nach den jeweiligen Projekten in denen sie sich befinden. Für manche Dateien finden sich Hinweise darüber, was genau geändert wurde, oder für was zum Beispiel eine neue Klasse verwendet wird.

Reactions-DSL-Runtime-Projekt:

Ergänzt:

- AbstractRoutinesFacadesProvider.xtend
- RoutinesFacadeExecutionState.xtend
- RoutinesFacadeProvider.xtend
- ReactionsImportPath.xtend
  - Utility-Klasse, die Importpfade repräsentiert.

Geändert:

- AbstractReactionRealization.xtend
- AbstractReactionsExecutor.xtend
- AbstractRepairRoutineRealization.xtend
- AbstractRepairRoutinesFacade.xtend

Reactions-Projekt:

Ergänzt:

- OverriddenRoutinesFacadeClassGenerator.xtend
- RoutinesFacadesProviderClassGenerator.xtend
- ReactionsImportHelper.xtend
  - Enthält sämtliche Helper-Funktionen, die sich auf den Import-Mechanismus beziehen.
- ReactionsImportScopeHelper.xtend
- ReactionsLanguageUtil.xtend

Geändert:

- MANIFEST.MF
  - Exportiere util-Package zur Verwendung im UI-Projekt.
- ReactionsLanguage.xtext
  - Ergänzungen an der Grammatik der Reactions-Sprache: ReactionsImports, überschriebene Reaktionen und überschriebene Routinen
- ExecutorClassGenerator.xtend
  - Sichtbarkeit der generierten Executor-Klasse explizit public (war vorher bereits public, und muss es auch sein, da die generierte ChangePropgationSpecification-Klasse den Executor referenziert).
  - Methode zum Erstellen des RoutinesFacadesProvider für das Segment.
  - Beinhaltet nun auch die importierten Reaktionen.
- ReactionClassGenerator.xtend
  - Sichtbarkeit der generierten Reaction-Klasse public, da Reaktionen von anderen Segmenten importiert und dort instanziiert werden können müssen.
  - Anpassungen aufgrund der Änderungen an AbstractReactionRealization.
- RoutineClassGenerator.xtend
  - Anpassungen aufgrund der Änderungen an AbstractRepairRoutineRealization.
- RoutineFacadeClassGenerator.xtend
- ClassNamesGenerators.xtend
- ReactionsLanguageConstants.xtend
- ReactionsLanguageHelper.xtend
- ParameterGenerator.xtend
- ReactionsLanguageFormatter.xtend
- ReactionsEnvironmentGenerator.xtend

- ReactionsLanguageJvmModelInferrer.xtend
- ReactionsLanguageScopeProviderDelegate.xtend
- ReactionsLanguageValidator.xtend
  - Um neue Validierungschecks ergänzt.

UI-Projekt:

- ReactionsLanguageOutlineTreeProvider.xtend
  - Anzeigetexte für neue Sprachelemente im OutlineTree-View hinzugefügt.

4. Github Links und Installation

Vitruv Projekt: https://github.com/vitruv-tools/Vitruv
Modularization Branch: https://github.com/vitruv-tools/Vitruv/tree/modularization
Modularization Branch ohne neu generierten Code (übersichtlicher im Vergleich mit dem master Branch): https://github.com/vitruv-tools/Vitruv/tree/modularization-slim
Vitruv-Applications-ComponentBasedSystems: https://github.com/vitruv-tools/Vitruv-Applications-ComponentBasedSystems

Um die Modularisierungsänderungen zu verwenden, wird Vitruv benötigt. Auf Github findet sich im Wiki eine Installationsanleitung für Vitruv: https://github.com/vitruv-tools/Vitruv/wiki/Getting-Started
Die Abhängigkeiten für Vitruv sollten über die Update-Seite am einfachsten in Eclipse installiert werden können: http://vitruv.tools/updatesite/nightly
Es werden neben den Abhängigkeiten von Vitruv keine zusätzlichen Abhängigkeiten benötigt.

Anschließend kann man das Vitruv Repository klonen und die Vitruv Projekte in Eclipse importieren. Um alle Abhängigkeiten aufzulösen, kann es nötig sein, die Reactions-Sprache durch Ausführen der "GenerateReactionsLanguage.mwe2"-Datei neu zu generieren. Bei Problemen kann es helfen, vorher sicherzustellen, dass der "src-gen" Ordner zumindest leer bereits vorhanden ist.

Sobald Vitruv in Eclipse eingerichtet ist und alle Abhängigkeiten aufgelöst worden sind, kann man den "modularization" Branch mit Git auschecken und dann eine Eclipse-Application-Instanz starten. Darin sollten dann alle Modularisierungsänderungen aktiv sein.

Im Vitruv-Applications-ComponentBasedSystems-Repository finden sich Projekte mit Reactions-Dateien, an denen man sich beispielhaft orientieren kann, falls man ein eigenes Projekt aufsetzen will und darin die Reactions-Sprache samt Modularisierungsänderungen testen will.
Alternativ gibt es noch dieses Repository hier, mit dem ich selber herumexperimentiere und mir anschaue, wie der generierte Code für die geschriebenen Reactions-Dateien samt Imports und Overrides aussieht: https://github.com/lhenni/ReactionsTest