Änderungsgetriebene Konsistenz zwischen Java-Code und Software-Architekturmodellen

Thomas Czogalik

Betreuer: Heiko Klare, Max Kramer

1 Einleitung

Das Praktikum `Software Quality Engineering mit Eclipse` fand im Sommersemester 2017 statt. Mein Thema war `Änderungsgetriebene Konsistenz zwischen Java-Code und Software-Architekturmodellen` und wurde von Heiko Klare und Max Kramer vom Lehrstuhl Software Design and Quality (SDQ) betreut.

Bei der Entwicklung eines Software-Systems können verschiedene Modelle, auf unterschiedlichen Abstraktionsebenen zum Einsatz kommen, um dieses zu beschreiben. Die dargestellten Informationen können voneinander abhängig und redundant sein. Entwickler müssen bei Änderungen an einem Modell evtl. auch Änderungen an anderen Modelle vornehmen, damit diese konsistent bleiben. Automatisierte Verfahren sind für diese Konsistenzwiederherstellung nötig, da ein Entwickler kaum in der Lage ist den Überblick über alle Abhängigkeiten zu behalten. Einen solchen Ansatz verfolgt Vitruvius [KrBL13]. Durch einen änderungsgetriebenen Ansatz und verschiedener Transformationssprachen, können verschiedene Modelle automatisch konsistent gehalten werden. Eine solche Transformationssprache ist die Reactions-Sprache [Kram17]. Die Reactions-Sprache ist eine Domänenspezifischesprache (DSL) um Konsistenzbeziehungen zwischen Metamodellen zu beschreiben. Sie besteht aus Reactions und Routinen. Reactions definieren wann Konsistenzerhaltungslogik ausgeführt werden soll. Routinen definieren dabei wie die Konsistenz (wieder) hergestellt wird. Dabei können Routinen von Reactions oder anderen Routinen aufgerufen werden und können wiederverwendet werden. Abbildung 1 zeigt ein Beispiel für eine solche Reaction. Dabei wird in der Reaction zunächst definiert, wann diese ausgeführt werden soll. Die Reaction wird ausgeführt, wenn ein Java-Interface in ein Java-Paket eingefügt wird. Wenn dies der Fall ist, soll eine Routine mit dem Namen `createPCMInterface` aufgerufen werden. Diese Routine erstellt ein PCM-Interface und initialisiert es mit dem gleichen Namen wie das Java-Interface. Im Anschluss wird noch eine Korrespondenz zwischen beiden Elementen erstellt, damit man zwischen ihnen navigieren kann. Dies ist unter anderem nötig, wenn zum Beispiel Änderungen an einem der beiden Elemente durchgeführt werden.

Im Folgenden wird ein Überblick über das Praktikum gegeben. Dabei wird in Abschnitt 2 beschrieben was meine Aufgaben waren und das Ergebnis präsentiert. In Abschnitt 3 werden Probleme mit dem Ergebnis beschrieben. Abschnitt 4 bewertet die Reactions-Sprache und das Praktikum. Schließlich wird in

```
reaction InterfaceCreated {
        after element java::Interface inserted in java::Package
        call createPCMInterface(newValue)
}
```

```
routine createPCMInterface(java::Interface jInterface) {
        action {

                val pcmIface = create pcm::Interface and initialize {
                        pcmIface.entityName = jInterface.name
                }

                add correspondence between pcmIface and jInterface

        }
}
```

Abbildung 1. Beispiel für eine Interface-Reaction in der Reactions-Sprache

Abschnitt 5 beschrieben, wie es mit der Implementierung nach dem Praktikum weitergehen und wie die Reactions-Sprache erweitert werden kann.

2 Tätikgeit

Im Folgenden Abschnitt werden die Aufgaben und das Ergebnis des Praktikums beschrieben.

Die Hauptaufgabe des Praktikums war es die Konsistenzerhaltung zwischen Java-Code und PCM-Elementen mithilfe der Reactions-Sprache umzusetzen und damit die bestehende Java Implementierung zu ersetzen. Gleichzeitig sollte evaluiert werden wie gut das mit der Reactions-Sprache möglich ist und wo es noch Probleme oder Verbersserungsmöglichkeiten gibt. Die Java Implementierung war, zu großen Teilen, durch JUnit-Plugin-Tests getestet worden. Diese Tests sollten sicherstellen, dass das Verhalten gleich bleibt, wenn die Implementierung ausgetauscht wird. Ein JUnit-Plugin-Test bietet die Möglichkeit ein Plugin in einer eigenen virtuellen Maschine zu starten. In dieser Instanz wird der Test gestartet. Damit verschiedene Modelle konsistent gehalten werden können, benötigt es zunächst Regeln, die die Elemente des einen Modells auf Elemente des anderen Modells abbilden. Die Abbildungsregeln für Java-Code und PCM-Elemente sind in Abbildung 2 abgebildet. Diese stammen aus der Dissertation von Michael Langhammer [Lang17]. Dort kann auch eine genauere Beschreibung der Zuordnung gefunden werden. Außerdem wurden die beiden Arbeiten von Max Kramer [Kram17] und Heiko Klare [Klar16] benutzt um ein Verständnis für

PCM metamodel element	**Source code language element**
Repository	Three *packages*: main, contracts, data types
BasicComponent	Package within the main *package* and a public component realisation *class* within the package
OperationInterface	*Interface* in the contracts package
Signature&Parameters	*Methods&parameters*
CompositeDatatype	*Class* with getter and setter for inner types
CollectionDatatypes	*Class* that inherits from a Java collection type (e.g. `ArrayList`)
RequiredRole	*Field* typed with required interface in the component-class and *constructor parameter* for the field in the component-class
ProvidedRole	Main class of providing component *implements* the provided interface
SEFF	*Method* in the component realisation *class* that overrides the corresponding interface method
System	*package* and public class within the package
CompositeComponent & Subsystem	*Package* within the main Repository package and public class within the package
AssemblyContext	*field* in the class and *instantiation* of the mapping class
RequiredRole	*Member* typed with required interface and *constructor parameter* for member
ProvidedRole	class of the *System implements* the provided interface
ProvidedDelegationConnector	delegation call to the corresponding field within the overwritten method of the provided interface
RequiredDelegationConnector	*constructor parameter*, typed with the interface of the required delegation connector that is given to the constructor of the requiring components-realisation class
AssemblyConnector	assignment of the constructor parameter from

Abbildung 2. Übersicht der Zuordnung von PCM-Elementen und Java-Code

die Reactions-Sprache zu erhalten. Im Anschluss wurde die Java Implementierung analysiert und nach und nach durch Reactions und Routinen ausgetauscht. Mithilfe der bestehenden Tests wurde die Reactions-Implementierung fortlaufend validiert. Jede Testklasse validierte dabei eine Abbildungsklasse. D.h. die Testklasse `ClassMappingTest` validiert alle Abbildungsregeln die mit Java-Klassen zu tun haben. Die Tests sind so aufgebaut, das jede mögliche Zuordnung aus Abbildung 2 einmal getestet wird. Außerdem gibt es Tests für einige Editierfunktionen, wie das Umbenennen von Elementen.

Im Ergebnis wurden die Abbildungsregeln für Repositories, Komponenten, Systeme, Datatypes, Rollen und den SEFF (Service Effect Specification) umgesetzt. Zudem wurde gegen Ende des Praktikums bemerkt, dass nicht alle Abbildungsregeln der Java Implementierung getestet wurden. Dazu gehören der SEFF Inhalt, AssemblyContext und AssemblyConnector. Außerdem wurden einige Editierfunktionen, wie das Löschen oder Ändern von Elementen nicht vollständig getestet. Da das Erstellen von Tests nicht Teil des Praktikums war, wurde entschieden sich zunächst auf die vorhandenen Tests und die damit einhergehenden Reactions zu konzentrieren.

Die Implementierung besteht dabei aus zwei Reactions-Dateien, die für unterschiedliche Arten von Reaktionen verantwortlich sind. In `Java2PcmClassifier` befinden sich alle Reactions, die ausgelöst werden, wenn Pakete, Klassen, Schnittstellen oder Typreferenzen erstellt werden. In `Java2PcmMethods` befinden sich alle Reactions, die ausgelöst werden, wenn Parameter, Felder, Methoden erstellt oder Elemente umbenannt werden. Eine Hilfsklasse bietet zudem Helfermethoden an, die benötigt werden um zum Beispiel Referenzen aufzulösen oder bestimmte Elemente zu filtern. Diese Methoden sind nötig, da ihre Funktionalität nicht mit der Reactions-Sprache umsetzbar ist. In Unterabschnitt 4.1 wird näher auf die Hilfsklasse eingegangen. Außerdem befinden sich in `Java2PcmUserSelection` alle möglichen Auswahlmöglichkeiten die der Benutzer im Lauf der Konsistenzerhaltung haben kann, um zwischen bestimmten Reaktionen auswählen zu können. Auf Github [1] kann die Implementierung eingesehen werden. Außerdem mussten die Tests teilweise angepasst werden. Dies war nötig da sie zum einen sehr an die Java Implementierung gekoppelt waren und zum Anderen bereits mit der Java Implementierung nicht funktioniert haben. Außerdem ermöglichten die Tests bestimmte Benutzerinteraktion nicht und mussten deshalb erweitert werden. Im Ergebniss laufen fast alle Tests erfolgreich durch. Auf die fehlerhaften Tests wird in Abschnitt 3 eingegangen.

Um das Ergebnis, neben den Tests, weiter zu validieren und um weiteres Feedback zu erhalten wurde gegen Ende des Praktikums ein Code-Review vorbereitet und durchgeführt. Im Anschluss wurden die Ergebnisse des Reviews in die Implementierung eingebaut.

[1] https://github.com/vitruv-tools/Vitruv-Applications-ComponentBasedSystems

3 Probleme

In diesem Abschnitt soll auf noch bestehende Probleme mit dem Ergebnis eingegangen werden.

Eines der Ziele des Praktikums war es, dass neben der Umsetzung der Reaktionen mit der Reactions-Sprache auch alle vorhandenen Tests weiterhin erfolgreich durchlaufen und sich das Verhalten des Systems nicht ändert. Dieses Ziel ist schwierig zu validieren, da einige Tests aus nicht bekannten Gründen manchmal fehlschlagen und beim zweiten Ausführen erfolgreich durchlaufen. Mögliche Ursachen für das Scheitern sind zum Beispiel zu lange Wartezeiten auf asynchrone Tasks. Außerdem kann die TUID nicht immer aufgelöst oder das korrespondierende Element nicht gefunden werden. Eine TUID (temporary unique identifier) ist ein eindeutiger Identifizierer für Modellelemente über mehrere verschiedene Modelle hinweg. Führt man diese fehlgeschlagenen Tests nochmal oder einzeln aus, taucht der Fehler nicht mehr auf. Dieses Verhalten war bereits vor dem Praktikum und in anderen Bereichen des Projekts zu beobachten.

Ein weiteres Problem, im Ergebnis, ist das nicht immer die Reaction ausgelöst wird, die erwartet wird. Dies ist vor allem ein Problem beim Ändern von Parametern. Wird ein Parameter in einer Signatur verändert, wird zunächst ein neuer Parameter angelegt und im Anschluss der Alte gelöscht. Somit wird nicht ein Änderungs-Event sondern ein Hinzufüge- und Entfern-Event ausgelöst. Noch dazu wird beim Entfernen ein fehlerhafter Parameter vom Monitor geliefert. Dies konnte bis zum Abschluss des Praktikums nicht behoben werden.

4 Auswertung

Im Folgenden wird zunächst die Reactions-Sprache und im Anschluss das Praktikum ausgewertet.

4.1 Reactions-Sprache

Ich finde, dass die Reactions-Sprache durch ihren Aufbau und die verschiedenen Konstrukte einfach zu verstehen ist, da sie sich fast wie natürliche Sprache lesen lässt. Dadurch können nach ein paar Beispielen bereits die ersten Reactions geschrieben werden. Die verschiedenen Konstrukte führen dazu, dass der Reaction-Code kompakter wird und sich der Benutzer der Sprache um viele Dinge nicht kümmern muss. Ein Beispiel ist hier das Konstrukt `retrieve correspondence` um das korrespondierende Element zu erhalten. Während dies in der Reactions-Sprache innerhalb einer Zeile geht, müssen in Java mehrere Hilfsmethoden aus unterschiedlichen Bibliotheken und Null-Checks aufgerufen werden. Solche Konstrukte führen dazu, dass der Reactions-Code aus ca. 960 Zeilen, während der ursprüngliche Java-Code aus ca. 1420 Zeilen besteht. Trotzdem weisen die Reactions die gleiche Funktionalität auf wie die ursprüngliche Java-Implementierung, da alle Tests erfolgreich sind, unter Beachtung von Abschnitt 3. Somit kann die Reactions-Implementierung die Java-Implementierung ersetzen.

Trotz der genannten Vorteile musste neben den Reactions eine Hilfsklasse genutzt werden. Von den 960 Zeilen ist die Hilfsklasse für 180 Zeilen verantwortlich. Die Hilfsklasse ist nötig, da bestimmte Funktionalitäten von der Reactions-Sprache nicht unterstützt werden. Eine solche Funktionalität ist, dass Routinen keine Rückgabewerte haben. In meiner Implementierung musste ich deshalb eine Hilfsmethode für das Extrahieren eines Paketnamens und für das vergleichen von Signaturen schreiben. Hier muss jedoch diskutiert werden, ob dies überhaupt gewollt ist, dass Routinen Rückgabewerte haben sollen. Neben diesen Methoden ist ein Großteil der Hilfsklasse dafür da um Typreferenzen aufzulösen. Vor allem hier könnte der Code deutlich reduziert werden, weil das Auflösen von Referenzen in allen Modellen auftaucht. Ein Sprachkonstrukt oder sogar eine automatische Auflösung im Hintergrund wäre hier hilfreich, da sonst der Benutzer sich um jede Auflösung selber kümmern muss. Neben der Hilfsklasse verursacht die Benutzerinteraktion viel ähnlichen Code. Eine Benutzerinteraktion besteht aus drei Teilen. Zuerst wird die Nachricht und die Möglichkeiten erstellt. Dann wird die Benutzerinteraktion initiiert. Schließlich werden je nach Eingabe unterschiedliche Aktionen ausgeführt. Diese Schritte werden momentan in Xtend-Code geschrieben. Ein Konstrukt, dass dem Nutzer der Sprache hilft diese drei Schritten zu definieren, wäre hier hilfreich und weniger fehleranfällig, bei der Definition, als Xtend-Code. Außerdem ist es nicht möglich Routinen zu überladen. Dies führt dazu das anstatt zum Beispiel einer Routine `createElement` mit Parameter `pcm::Repository` oder `pcm::BasicComponent` jedes Mal ein neuer Name gewählt werden muss. Dies führt nicht unbedingt zur Lesbarkeit bei, da man direkt in der Signatur erkennen kann, was der Zweck dieser Routine ist.

Neben den Sprachfunktionalitäten muss der Editor für die Reactions-Sprache angepasst und erweitert werden. Dieser unterstützt nämlich wenig Funktionen aus dem Eclipse-Kontext. Zu den fehlenden Funktionen zählen ein Formatierer, eine Autovervollständigung und die Refaktorierung von Name, aber auch zum Beispiel das extrahieren von Routinen. Diese Funktionalitäten sind mir besonders aufgefallen, da ich diese, bei der Nutzung von Eclipse, häufig verwende. Weiter Nutzerstudien oder Praktika könnten ähnliche oder andere Funktionen offenbaren, die Benutzer der Reactions-Sprache benötigen. Durch das Bereitstellen dieser Editor Funktionen könnte die Benutzung der Reactions-Sprache noch einfacher werden, da diese im Eclipse Kontext erwartet werden.

4.2 Praktikum

Meine Erwartungen an das Praktikum waren, dass ich an einem aktuellen Forschungsprojekt mitarbeiten darf und meine Arbeit im Anschluss auch weiterverwendet wird. Ich habe mich für das Thema entschieden, weil ich mehr über das Vitruvius Projekt und die Reactions-Sprache erfahren wollte.

Diese Erwartungen konnten jeweils erfüllt werden. Zum einen wurde meine Arbeit bereits in den Master-Branch integriert und zum anderen war das Thema interessant, da ich Einblicke in Vitruvius und in die Reactions-Sprache bekommen habe. Außerdem wurden viele Techniken, aus den Vorlesungen des

Lehrstuhls, praktisch angewandt. Techniken waren unter anderem Modellgetriebene Softwareentwicklung, Continuous-Integration und Code-Reviews. Während des Praktikums wurde ich sehr gut von meinen Betreuern betreut. Bei den regelmäßigen Treffen präsentierte ich neue Implementierungen. Bei Problemen oder Fragen konnte mir, während der Treffen, aber auch oft per Mail, immer geholfen werden. Durch das eigene Zeitmanagment, konnte man sich die Arbeit bis zum nächsten Treffen gut einteilen und war deutlich flexibler während des gesamten Semesters. Bereits zu Beginn des Praktikums wurde klargemacht, dass bestimmte Grundlagen wie zum Beispiel Java und Eclipse Voraussetzungen sind. Nach dem Praktikum erkenne ich auch, dass dies sinnvoll ist. Ohne dieses Vorwissen müsste man sich teilweise noch tiefer einarbeiten. Den neben dem Projekt muss man sich, je nach Thema oder Vorwissen, in weitere Techniken (Modellgetriebene Softwareentwicklung (MDSD)) oder Sprachen (Xtend, Reachtion-Sprache) einarbeiten. Da ich bereits einige Vorlesungen am Lehrstuhl (STW2, MDSD, Software Evolution) besucht hatte und außerdem in meiner Tätigkeit als Werkstudent mit MDSD, Xtend, Xtext und Eclipse-Plugin-Entwicklung zu tun hatte, fiel mir die Einarbeitung deutlich leichter. Dadurch konnte ich viel schneller mit der Implementierung beginnen. Außerdem lohnt sich das Praktikum, wenn man die Techniken (z.B. MDSD, Code-Review), Sprachen (z.B. Java, Xtend), etc. aus den Vorlesungen des Lehrstuhls in der Praxis erleben möchte.

5 Ausblick

In diesem Abschnitt soll ein Ausblick über eine mögliche Weiterentwicklung der Reaction-Sprache und Weiterführung der Praktikumsimplementierung gegeben werden.

Obwohl die Reaction-Sprache sehr intuitiv und einfach zu verstehen ist gibt es dennoch einige Verbesserungsmöglichkeiten wie sie in Unterabschnitt 4.1 besprochen wurden. Die Möglichkeit der Routinenüberladung und der Routinenrückgabe sollte nochmal diskutiert werden, da ein solches Verhalten möglicherweise nicht gewünscht ist. Das auflösen von Typreferenzen und ein besserer Umgang mit der Nutzerinteraktion sollten jedoch auf jeden Fall umgesetzt werden. Neben den Sprachfunktionalitäten sollten vor allem die fehlenden Editorfunktionen umgesetzt werden, da diese viel zur Benutzbarkeit der Reaction-Sprache beitragen können. Dabei sind der Formatierer, die Autovervollständigung und Refaktorierungsfunktionen Funktionen die ich selber sehr oft in Eclipse verwende. Mögliche weitere Nutzerstudien könnten noch weitere häufig genutzte Editorfunktionen offenbaren.

Wie bereits in Abschnitt 2 erwähnt muss das Ergebnis um Tests für das Löschen von Elementen und für Felder und Parameter erweitert werden. Außerdem muss die Implementierung um die Löschreaktionen ergänzt werden. Die Feld-Reactions sind implementiert und müssen nur mit weiteren Tests validiert und evtl. angepasst werden. Schließlich muss noch, wie in Abschnitt 3 angesprochen, die Reaction für die Änderung von Parametern richtig Implementiert und getestet werden.

Literatur

Klar16. Heiko Klare. Designing a Change-Driven Language for Model Consistency Repair Routines. Diplomarbeit, Karlsruhe Institute of Technology (KIT), 2016.

Kram17. Max Emanuel Kramer. *Specification Languages for Preserving Consistency between Models of Different Languages*. Dissertation, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, 2017.

KrBL13. Max E. Kramer, Erik Burger und Michael Langhammer. View-centric engineering with synchronized heterogeneous models. In *Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*, VAO '13, New York, NY, USA, 2013. ACM, S. 5:1–5:6.

Lang17. Michael Langhammer. Automated coevolution of source code and software architecture models, [2017].