Fujaba/SPin

Zwei Transformationsbeispiele

Andreas Roth [1264421]
Technische Universität Darmstadt
roth@rbg.informatik.tu-darmstadt.de

Abstract

Dieses Paper behandelt die Anwendung eines Modellierungsansatzes, der Architektur-Stratifikation heißt und in dem Tool SPin umgesetzt wird. SPin ist ein Plug-in für das CASE-Tool Fujaba, das es ermöglicht Fujaba-Modelle mit Annotationen zu versehen und dann mit Hilfe von selbst erstellten Transformationsregeln zu verfeinern.

Im Rahmen des Seminars „Themen der Modellierung“ im Wintersemester 2006 soll dieser Ansatz anhand von zwei vorgegebenen Transformationsbeispielen erprobt und anschließend diskutiert werden. Das Ziel ist es dabei, diese zwei Transformationsregeln so allgemein wie möglich zu definieren, sodass sie nach dem Exportieren in eine Regeldatenbank auch auf andere, ähnliche Ausgangssituationen, angewendet werden können.

1. Einleitung

Das Problem an gewöhnlichen Modellierungsansätzen, die nur eine bestimmte Sicht (z.B. Architektursicht) auf ein System erlauben, ist, dass sie große und komplexe Software Systeme nicht ausreichend erfassen.

Wenn der Abstraktionsgrad eines Modellierungskonzeptes sehr hoch ist, erhält man zwar eine gute Übersicht über die gesamte Architektur des Systems, allerdings bleiben dann wichtige Details des Systems verborgen. Wählt man hingegen einen geringen Abstraktionsgrad, versteht man zwar wichtige Details des Systems, allerdings verliert man dann den Blick für das Gesamtsystem; man sieht den Wald vor lauter Bäumen nicht.
Der Ansatz der diesen Konflikt beheben soll, heißt Architektur-Stratifikation.

Im Folgenden wird dieser Ansatz kurz erläutert (Kapitel 2). Danach wird das CASE-Tool Fujaba (Kapitel 3) vorgestellt. Im Kapitel 4 geht es anschließend um „story driven modeling“ (SDM) - eine Technik, die in Fujaba eingesetzt wird um Methoden zu beschreiben. Darauf aufbauend gehe ich auf das Plug-in SPin (Kapitel 5) ein und erläutere welche Möglichkeiten es gibt Modelle zu annotieren und „refinement rules“ zu definieren.

Nachdem nun das Hintergrundwissen für die Vorgehensweisen in Fujaba und SPin vermittelt wurde, wird anhand von einem einfachen Transformationsbeispiel (Kapitel 6) schrittweise näher auf die Funktionsweise von Fujaba und SPin eingegangen. Anschließend wird noch ein etwas komplexeres Transformationsbeispiel (Kapitel 7) erläutert.

In Kapitel 8 gehe ich dann auf die Dokumentation von und meine Probleme mit Fujaba und SPin ein und beschreibe, wie ich sie lösen konnte. Letztendlich wird in Kapitel 9 der Ansatz der Architektur-Stratifikation zusammengefasst und die Umsetzung in SPin bewertet.

2. Architektur-Stratifikation

Der Ansatz der *Architektur-Stratifikation* versucht den in der Einleitung beschriebenen Konflikt zwischen verschiedenen Modellierungsebenen zu lösen. Es geht bei Architektur-Stratifikation nicht darum ein vorgegebenes Modell in Code zu transformieren, sondern vielmehr darum mehrere Modell-zu-Modell Transformationen durchzuführen die das Modell schrittweise verfeinern [3].

Dieses Modellierungskonzept fasst eine Systemarchitektur als eine Menge von Abstraktionsebenen auf. Dabei hat jede Ebene einen anderen Abstraktionsgrad, wodurch eine hierarchische Ordnung entsteht. Jede dieser Abstraktionsebenen (genannt Stratum) gibt im Vergleich zum darüber liegenden Stratum weitere Details zur Sicht frei, bis das unterste Stratum - welches ein UML-Klassendiagramm mit implementierten Methoden oder direkter Java Code sein könnte - das Gesamtsystem am konkretesten beschreibt. Wichtig ist, dass jedes Stratum immer das Gesamtsystem beschreibt - in dem des Stratums zugrunde liegenden Abstraktionsniveau.

Zwischen den verschiedenen Strata kann man die Ansicht beliebig hin und her wechseln und somit das System aus unterschiedlichen Abstraktionsebenen betrachten. In jedem Stratum ist es möglich Änderungen vorzunehmen, die sich sowohl auf darunter liegende, als auch auf darüber liegende Strata ausbreiten.

3. Fujaba

Fujaba wird von der Fujaba Development Group der Universität Paderborn[1] in Zusammenarbeit mit anderen Universitäten (darunter Kassel, Bayreuth, Darmstadt) entwickelt. Der Name Fujaba ist ein Akronym für **F**rom **U**ml to **J**ava **A**nd **B**ack **A**gain.

Das open-source CASE-Tool Fujaba ist eine Plug-in basierte, Modell-Transformations-Plattform, die durch das Konzept von Story Driven Modeling (siehe Kapitel 4) und Graphen Transformationen, UML-Modelle transformieren kann. Fujaba nutzt UML-Klassendiagramme zur Modellierung der Objektstrukturen eines Programms. Zur operationalen Spezifikation des Verhaltens einzelner Methoden bietet Fujaba einen eigenen Diagrammtyp an, der aus einer Mischung aus grafischen und textuellen Elementen besteht. Die Ausführung dieser Diagramme basiert auf der Theorie von Graphersetzungssystemen. Diese Diagramme beschreiben eine zu suchende Situation in den Laufzeitdaten des Programms und

[1] University of Paderborn, Germany, Fujaba Tool Suite. http://www.fujaba.de

wie die gefundene Struktur transformiert werden soll. Im Idealfall kann der Entwickler anschließend sein vollständiges Programm als kompilierfähigen Java-Code generieren [2].
Fujaba bietet folgende Funktionalitäten:

- Durch die Kombination aus UML-Klassendiagrammen und Storydiagrammen (siehe Kapitel 4) soll eine einfach zu benutzende UML- und Java-Entwicklungsplattform zur Verfügung gestellt werden
- Außerdem kann mit dem Fujaba Code-Generator Java-Code des gesamten modellierten Systems erzeugt werden, wodurch ein ausführbarer Prototyp entsteht
- weitere Code-Generatoren für andere Programmiersprachen können als Plug-ins hinzugefügt werden
- Der umgekehrte Weg von Java-Code zu einer UML-Repräsentation ist (durch ein entsprechendes Plug-in) auch möglich
- Fujaba verfügt über einen Plug-in Mechanismus und eine einfache Plug-in Verwaltung, wodurch neue Features eingefügt und somit die Möglichkeiten von Fujaba durch Plug-ins erweitert werden können

4. Story Driven Modeling

Man kann in Fujaba neben der Definition der Struktur des Systems mittels Klassendiagrammen auch Implementierungen der Methoden einer Klasse erstellen. Diese Implementierung wird in Fujaba mittels der sog. „Storydiagramme“ erstellt. Dazu wird jeder Methode ein *Storydiagramm* [1] zugeordnet, aus dem ein Codegenerator die gewünschte Implementierung erzeugen kann.

Die Notation von Storydiagrammen setzt sich aus grafischen und textuellen Teilen zusammen, dabei ist die Struktur des Diagramms eine leicht veränderte UML-Aktivitätsdiagramm Struktur.
Man unterscheidet zwischen dem Matching- und dem Transformationsteil: Der Matching-Teil sucht nach dem Vorkommen von bestimmten Mustern innerhalb des Programms und im Transformationsteil wird dann definiert wie dieses Muster im Programm verändert werden soll. Ich möchte hier nicht weiter auf den graphentheoretischen Hintergrund von Matching und Transformation eingehen, den man in [4] nachlesen kann.

Im Wesentlichen besteht der Aufbau eines Storydiagramms aus einem Startpunkt (*UMLStartActivity*), und einem oder mehreren Endpunkten (*UMLStopActivity*) [5] mit optionalen Aktivitäten dazwischen. Die Aktivitäten werden mit Kanten (*UMLTransitions*) verbunden, was die Reihenfolge der Abarbeitung festlegt. Aktivitäten können sowohl textuell aufgebaut sein, also Codefragmente enthalten, als auch komplett grafisch sein. Die grafischen Aktivitäten bilden den Kern des *Story Driven Modeling*, sie werden *Story-Pattern* genannt.

Im Story-Pattern können Informationen eines UML-Klassendiagramms - das in einem Fujaba-Projekt existiert - extrahiert werden. Die Notation ähnelt der von stark vereinfachten *Kollaborationsdiagrammen* (in UML2 als Kommunikationsdiagramme bezeichnet). Diese Notation beschreibt nur die Interaktion zwischen den Objekten, ohne die eigentlichen UML-Kollaborationen zu verwenden.

5. SPin

SPin ist ein Plug-in für Fujaba, das den Ansatz von Architektur-Stratifikation innerhalb von Fujaba unterstützt. SPin steht für **S**tratification **P**lug-**in** und wurde 2005 im Rahmen der Diplomarbeit [5] von Felix Klar an der Technischen Universität Darmstadt entwickelt.

SPin stellt ein spezielles Metamodell bereit, das dazu benutzt wird, Fujaba Diagramme mit Annotationen zu versehen. Das *SPin-Metamodell* erweitert dazu das *Fujaba-Metamodell*, sodass man Annotationen in Diagrammen hinzufügen kann und von dort Beziehungen zu anderen Objekten, mittels Kanten, herstellen kann.
Darüber hinaus benötigt man das *UML-Metamodell*. Dieses hilft einem beim Erzeugen von Assoziationen und Attributen, weil es ”weiß”, welche Attribute existieren und welche Assoziationen erzeugt werden können. Damit kann man auf die verschiedenen Diagrammelemente von Fujaba (wie Attribute, Methoden, Klassen, etc.) zugreifen.
SPin bietet in der Version 1.4 folgende Funktionalität:

- Annotieren von verschiedenen Modellelementen in Fujaba. Die Annotationen stellen meist Design Patterns [2] (wie Singleton, Observer oder Visitor) oder wichtige System Aspekte dar.
- Transformationsregeln können definiert und dann in eine Regeldatenbank exportiert werden. Für jede Regel wird eine Java Klasse erzeugt.
- Exportierte Regeln können wiederverwendet werden, indem beim Erstellen einer Annotation, eine bereits vorhandene Annotation in der Datenbank ausgewählt werden kann
- SPin kann aus vorhandenen Java Klassen automatisch die zugehörigen UML Klassen erstellen. Das erlaubt eine grafische Repräsentation von Java Klassen, die als Bytecode vorliegen.

Es gibt in SPin zwei Arten von Transformationen [6]: „refinement rules“, die zu einer konkreteren Repräsentation verfeinern und „abstraction rules“, die umgekehrt zu einer abstrakteren Repräsentation führen. Im Folgenden werden Annotationen und refinement-rules in SPin einmal näher betrachtet.

5.1 Annotationen

In SPin gibt es die Möglichkeit verschiedene Fujaba Diagrammtypen mit Annotationen zu versehen. Diese Annotation kann auf unterschiedliche Elemente des Diagramms verweisen. In der Transformationsregel können unter anderem alle verwiesenen Elemente (wie z.B. *UMLAttr*, *UMLMethod*, *UMLClass*) gematcht und zum Transformieren benutzt werden. Es ist jedoch auch möglich nicht verwiesene Elemente zu matchen. Da es allerdings Sinn macht, auf die für den Aspekt, den man mit der Annotation modelliert, wichtigen Elemente zu verweisen (z.B. wenn das Modell mehrere Klassen hat und die Transformation nur eine davon betrifft), haben Annotationen in SPin die Aufgabe Transformationen zu steuern.

Die Notation der Annotationen in SPin ist ähnlich zu UML-Kollaborationen. Sie beschreibt welche Rolle von den einzelnen referenzierten Objekten im Diagramm eingenommen wird. Dadurch ist es gut möglich, den Diagrammen eine gewisse Semantik - im Sinne von Design Patterns bzw. kleineren Systemaspekten - zu geben. Jede Annotation wird dann später mit einer Transformationsregel verbunden. Wenn diese vollständig definiert ist, kann die Transformation durchgeführt werden, wenn man sie auf dem entsprechenden Annotationsobjekt aufruft:
Da es in Fujaba4 immer nur ein aktives Projekt gibt, das offen ist, wird das aktuelle Projekt zunächst dupliziert und dann die Transformation durchgeführt. Im Vergleich zu anderen Ansätzen arbeitet die Transformation in SPin daher „inplace“, d.h. das Modell wird im wahrsten Sinne des Wortes transformiert (durch Modifikation des bestehenden Modells) und nicht komplett neu aufgebaut.

Zur Zeit ist es in SPin leider nicht möglich dieses Vorgehen re-

[2] Wikipedia. http://de.wikipedia.org/wiki/Fujaba

kursiv durchzuführen, d.h. die Transformation muss auf jedem Annotationsobjekt ausgeführt werden.

Wenn man also an die Idee von Architektur-Stratifikation denkt, dann definiert man mit Annotationen, welche Elemente an der Verfeinerung eines Stratums - in einem Transformationsschritt - beteiligt sind und wie sie parametrisiert werden.

5.2 Verfeinerungsregeln

Wenn man in SPin eine „refinement rule“ erstellt, wird der Ordner ‚Activity Diagrams‘ mit einer neuen Klasse für diese Regel - die mehrere Methoden enthält - in Fujaba angelegt. SPin erstellt für jede Methode ein Gerüst, welches durch den Benutzer erweitert werden kann. In der Methode ‚apply‘ - auf die in den beiden Transformationsbeispielen noch näher eingegangen wird - wird die eigentliche Transformation durchgeführt. Man kann diese Regel (wie in Kapitel 4 zu SDM beschrieben) in zwei Teile aufteilen:
Der Matching Teil der Regel sucht in dem annotierten Modell die Elemente aus, die transformiert werden sollen. Der Transformationsteil führt die eigentliche Transformation - auf den durch das Matching ausgewählten Elementen - durch, indem er angibt, was mit den gematchten Elementen geschehen soll.
Diese Transformationsregeln können vollständig vom Benutzer verändert und neugestaltet werden, SPin stellt lediglich das Basisgerüst einer Regel bereit. Nachdem die Regel vollständig definiert wurde, kann diese mit Hilfe von SPin in die Regeldatenbank exportiert und dann benutzt werden.

6. Getter- / Setter-Methoden für public Attribute

Da nun die grundlegenden Techniken für SPin und Fujaba erläutert wurden, kommt jetzt ein einfaches Transformationsbeispiel. Anhand dieses, werde ich auf die einzelnen Schritte eingehen, die notwendig sind um ein abstraktes Modell (in Form eines UML-Klassendiagramms) in eine konkretere Repräsentation zu transformieren.

In diesem Beispiel soll eine Regel zur Verfeinerung einer Klasse mit mehreren `public` Attributen geschrieben werden. Diese Verfeinerungsregel muss so definiert werden, dass es nach der Verfeinerung des Modells für jedes public Attribut eine getter- / setter-Methode gibt, die die entsprechende Funktionalität implementiert.

6.1 Klassendiagramm in Fujaba

Zunächst einmal wird ein Klassendiagramm in Fujaba erstellt, welches annotiert werden soll. Dazu wählt man im Menü *Diagrams -> New Class Diagram* aus und erstellt mit *Class Diagram -> Create / Edit Class ...* eine neue Klasse, die man ‚MyClient‘ nennt. Dieser Klasse fügt man zunächst nur ein public Attribut `name` (mit Rechtsklick auf die Klasse, *Create / Edit Attributes ...*) hinzu, um später überprüfen zu können, ob das Modell tatsächlich korrekt verfeinert wird.

6.2 Annotation des Klassendiagramms

Nachdem das Klassendiagramm erstellt wurde, kann man es mit den SPin Annotationen annotieren. Zunächst einmal muss allerdings sichergestellt werden, dass in dem Projekt das SPin-Metamodell vorhanden ist, denn sonst können die Transformationsregeln nicht angewendet werden. Um sicherzugehen, dass das Projekt tatsächlich das SPin-Metamodell beinhaltet, fügt man es jetzt hinzu (Abbildung 1). Jetzt sieht man, dass auf der linken Seite von Fujaba mehrere Klassendiagramme erstellt wurden, die die Klassen des SPin-Metamodells beinhalten. Nun kann man mit der Annotation des Diagramms anfangen. Dazu öffnet man den SPin annotation-editor (mit Rechtsklick auf freie Fläche, *SPin -> show annotation-editor...*). Hier erstellt man dann auf der linken Seite eine neue Annotation und setzt den *annotation name* auf ‚Client‘. In der Auswahlbox wählt man jetzt als *annotation target* den Namen der Klasse aus, die man annotieren möchte - also ‚MyClient‘. Weiter muss hier nichts eingestellt werden, also kann man mit *OK* bestätigen. (Abbildung 2)

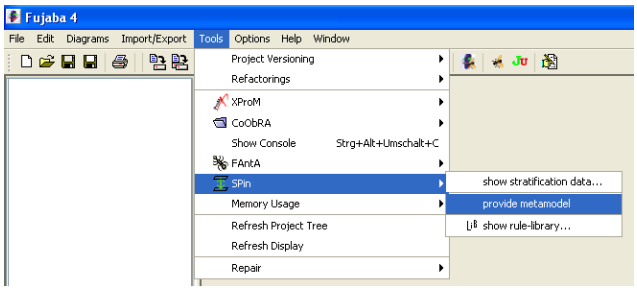

Abbildung 1. SPin - Hinzufügen des Metamodells

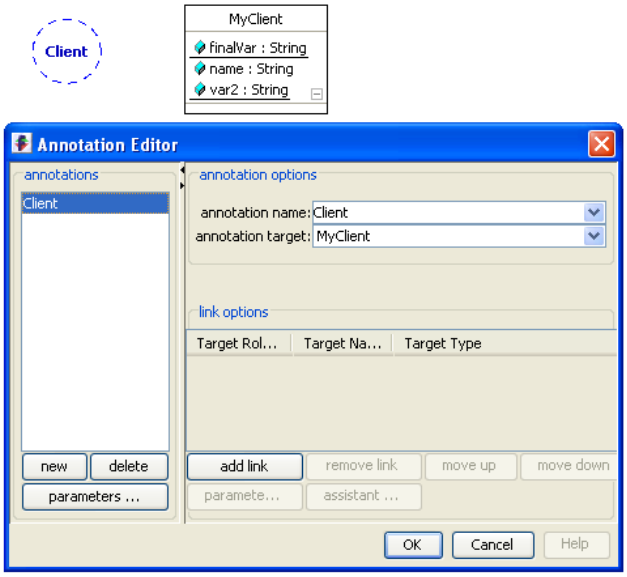

Abbildung 2. SPin - Annotation Editor

6.3 Definieren der Verfeinerungsregel

Nachdem das Modell nun vollständig annotiert ist, kann man eine Regel definieren, die festlegt, wie das Modell transformiert werden soll. Dazu erstellt man der Übersicht halber ein neues Klassendiagramm, das man ‚Rules‘ nennt. Hier klickt man dann mit der rechten Maustaste auf eine freie Fläche und wählt *SPin -> create rule...* aus. Im Dialog ‚Create TransformationRule‘ wählt man nun *RefinementRule* und benennt diese wie die Annotation, damit die Verbindung zwischen beiden klappt - also ‚Client‘. Nach dem Bestätigen mit *OK* wird ein Klassendiagramm mit zwei Klassen erstellt (Abbildung 3). Um diese Klassen braucht man sich nicht weiter zu kümmern, wichtig ist nur, dass in der Klasse ‚RRClient‘ die Methode ‚apply‘ vorhanden ist. Außerdem wird auf der linken Seite von Fujaba ein Ordner ‚Activity diagrams‘ angelegt, in dem es eine Klasse ‚RRClient‘ gibt, die mehrere Methoden (unter anderem die ‚apply‘ Methode) enthält. Hier interessiert aber zunächst nur die ‚apply‘ Methode, für die bereits eine UML-Aktivitätsdiagramm

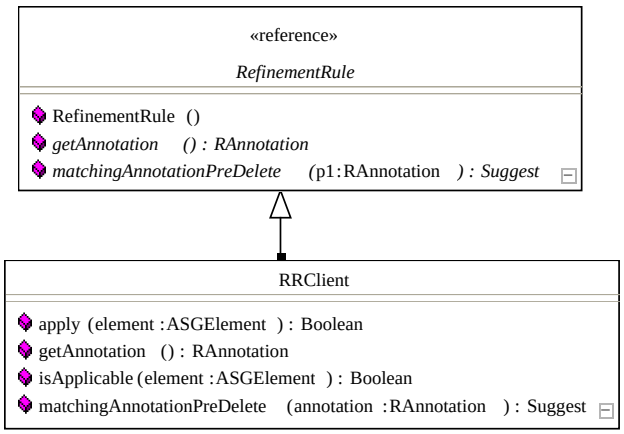

Abbildung 3. SPin - Refinement Rule

Struktur existiert. Dieses Diagramm ist das in Kapitel 4 beschriebene *Storydiagramm*. Abbildung 4 zeigt die automatisch generierte Struktur, die wir nun erweitern werden.

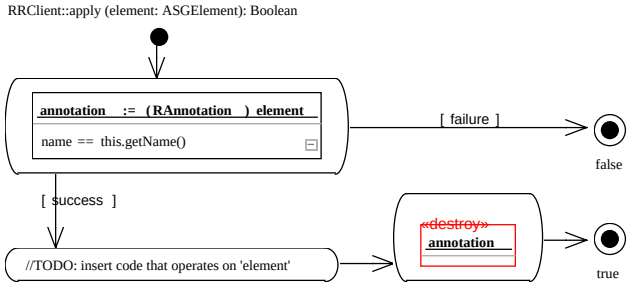

Abbildung 4. SPin - Apply Methode

6.4 Die `apply` Methode

Die apply Methode ist die Regel, wie das Modell transformiert werden soll. Hier kann man auf verschiedene Elemente zugreifen, die mit der Annotation verbunden sind, wie die Methoden und Attribute der Zielklasse. In Abbildung 5 sieht man in der ersten Aktivität die Notation von Story-Pattern.
Das ist der „Matching“-Teil der Regel. Diese Notation scheint auf den ersten Blick nicht sehr intuitiv zu sein. Man würde statt der Verwendung der Notation über Objektdiagramme im Metamodell lieber eine konkrete Syntax verwenden. Allerdings modelliert man hier genau die Kante zwischen zwei UML Objekten (der Annotation und der Klasse), die nicht durch die konkrete Syntax - wie das annotierte Klassendiagramm - erfasst werden kann (mehr dazu Ende 9. Kapitel).

In diesem Matching-Teil wird die richtige Klasse, auf die die Annotation verweist, gefunden. Diese Klasse enthält dann die public Attribute, die durch getter-/setter-Methoden gelesen bzw. gesetzt werden sollen. Falls die Klasse nicht gematcht werden kann, wird die Regel abgebrochen (*[failure]*). Ansonsten kommt man über die Kante *[success]* zur nächsten Aktivität. Diese Aktivität besteht - im Gegensatz zur Diagrammstruktur der vorhergehenden - nur aus textuellen Elementen (also Codefragmenten). Man holt sich nun eine SPin Helper Klasse (*UMLFactory*), die dazu benutzt werden kann Methoden, Klassen, etc. zu erzeugen.

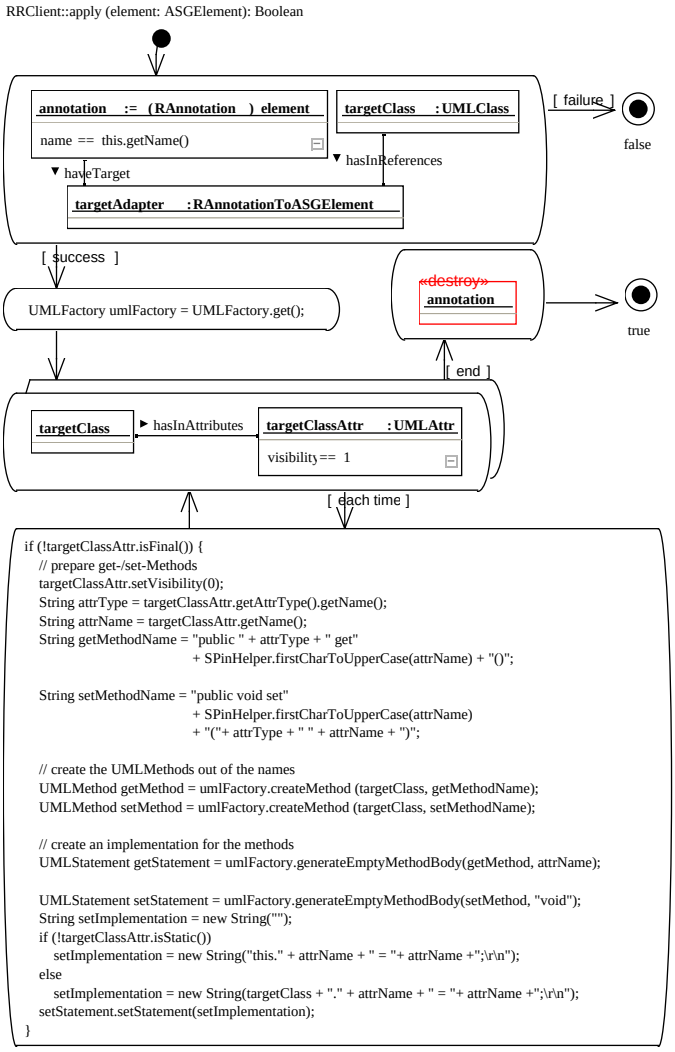

Abbildung 5. SPin - Fertige Apply Methode

Die Aktivität mit der ausgehenden *[each time]*-Kante ist wieder eine Story-Pattern Aktivität mit einer besonderen Funktion: Es wird hier eine Schleife modelliert, was die doppelte Umrandung der Aktivität anzeigt. Diese Schleife läuft solange, bis alle Attribute der Klasse gematcht wurden. Dabei wird in jedem Schleifendurchlauf überprüft, ob das Attribut public und nicht final ist und dann der neuen Klasse mit Hilfe der UMLFactory Klasse als privates Attribut hinzugefügt.
Desweiteren werden die Getter-/Setter-Methoden in der Zielklasse für die Attribute erstellt und mit der entsprechenden Implemetierung versehen (in der Aktivität mit der eingehenden *[each time]*-Kante). Das ist der Transformationsteil der Regel, der hier rein textuell aufgebaut ist. Nachdem alle Attribute gematcht wurden wird die Schleife über die Kante *[end]* verlassen. Die Aktivität mit der *<<destroy>>* Beschriftung sorgt letztendlich dafür, dass das Annotationsobjekt aus dem UML Diagramm entfernt wird.

Es sei angemerkt, dass man den Transformationsteil der Regel, der hier nur mit Hilfe von Java-Code definiert wurde, auch nur mit Diagrammelementen (des Story-Pattern) beschreiben kann. Darauf gehe ich im nächsten Transformationsbeispiel noch näher ein.

7. Umsetzung des Server-Proxy Pattern

In dem zweiten Transformationsbeispiel geht es darum die `public`-Methoden einer Klasse mit Hilfe der Klassen *java.rmi.Remote* und *java.rmi.server.UnicastRemoteObject* „remote-fähig“ zu machen. Die Abbildung 6 stellt dar, wie die Transformation einer Klasse mit public-Methoden aussehen soll.

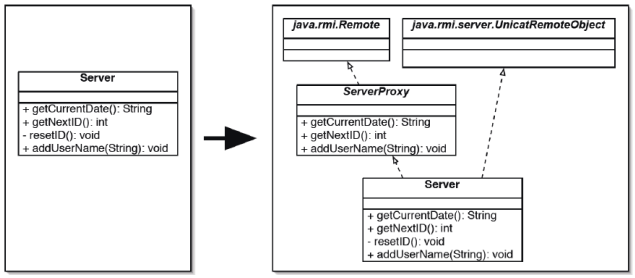

Abbildung 6. Aufgabenbeschreibung

Man erstellt, wie im ersten Beispiel beschrieben, wieder ein Klassendiagramm mit der Klasse und den Methoden, die man remote-fähig machen möchte und annotiert diese mit einer SPin Annotation (Abbildung 7). Mit dieser verweist man dann auf die Klasse.

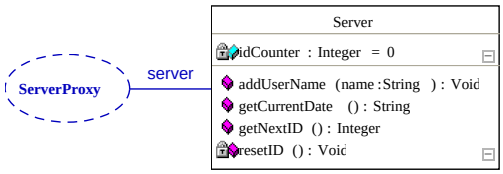

Abbildung 7. SPin - Annotation des Klassendiagramms

Nachdem man wieder eine „refinement-rule“ erstellt hat, kann man die *apply*-Methode definieren.
Dazu sind folgende Schritte umzusetzen:

1. Schnittstelle „<ServerProxy>Interface“ hinzufügen
2. Referenz-Schnittstellen aus Java-API hinzufügen
   - java.rmi.Remote
   - java.rmi.server.UnicastRemoteObject
3. in ServerProxy: `public`-Methoden der Server-Klasse ergänzen
4. Exceptions hinzufügen
5. Konstruktor für die Server-Klasse erstellen
6. Vererbungsbeziehungen herstellen

Die Abbildung im Anhang zeigt die Umsetzung dieser Schritte in der vollständigen apply-Methode der Verfeinerungsregel.
Da die Abbildung zu groß ist, um sie auf Anhieb zu verstehen, hier die einzelnen Schritte:
In Abbildung 8 geschieht das matchen, der durch die Annotation verwiesenen Elemente. Im linken Pfad sieht man, wie die *Server*-Klasse, die durch einen Link mit der Annotation verbunden ist, gematcht wird. Der rechte Pfad matcht das Klassendiagramm, in dem sich die *Server*-Klasse befindet. Man braucht es später, um darin die benötigten Klassen erstellen zu können. Falls die Elemente korrekt gematcht werden konnten, gelangt man über die *[success]* Kante zur Abbildung 9.

Im Unterschied zu der Regel aus dem ersten Beispiel, sieht man hier Objekte und Kanten, die mit *<<create>>* beschriftet sind.

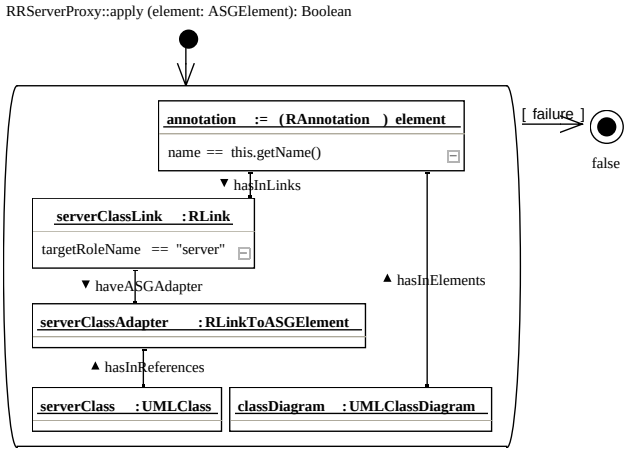

Abbildung 8. SPin - Apply Regel des 2. Beispiels

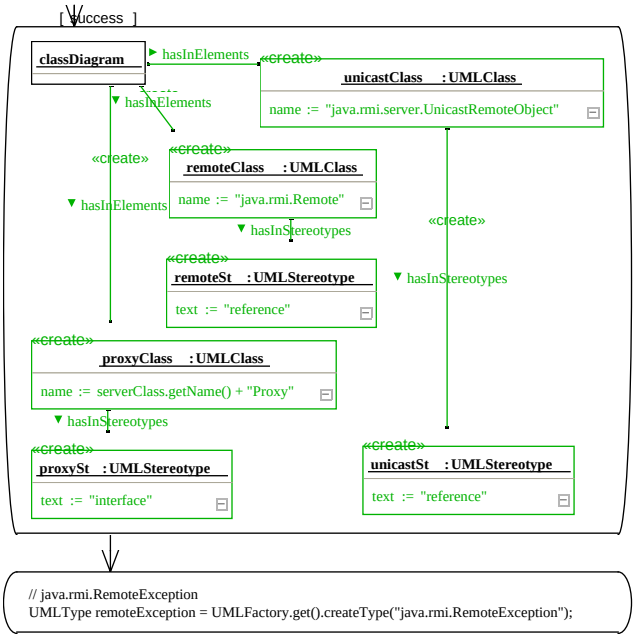

Abbildung 9. SPin - Apply Regel des 2. Beispiels

Diese Objekte werden direkt erzeugt und müssen nicht mit Hilfe von Java-Code generiert werden. Genauso, wie in den Aktivitäten aus Beispiel 1, die Code enthalten, ist es hier auch möglich Klassen, Attribute, etc. zu erzeugen, allerdings ohne die SPin Helper Klasse *UMLFactory* zu benutzen. Es ist also in SPin sowohl möglich eine Regel mit Benutzung von Java-Code, als auch komplett mittels Diagrammelementen umzusetzen und zwischen diesen beiden Möglichkeiten nach Bedarf, in einer Regel frei zu mischen. Dieser Teil der Regel setzt die Schritte 1. und 2. um, erzeugt also die beiden *RMI*-Klassen und die *ServerProxy*-Klasse. Um diese Klassen im richtigen Klassendiagramm zu erzeugen, benutzen wir das vorhin gematchte Klassendiagramm. Dazu erstellt man ein neues Objekt, dessen Name, dem des bereits gematchten Klassendiagramms gleicht (*classDiagramm*) und wählt die Eigenschaft „Bound“. Dadurch behält das Objekt den gebundenen Wert bei und man kann auf diesen zugreifen. Ist die Erzeugung geschehen, gelangt man

über eine Kante zur Aktivität aus Abbildung 10.

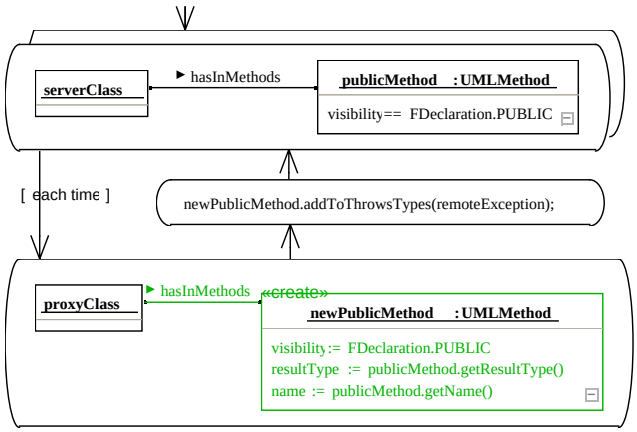

Abbildung 10. SPin - Apply Regel des 2. Beispiels

In dieser Abbildung erfolgt der 3. und 4. Schritt, es werden also die *public*-Methodensignaturen aus der *Server*-Klasse in die *ServerProxy*-Klasse übernommen. In einer Schleife wird überprüft, ob die *Server*-Klasse weitere - noch nicht gematchte - `public`-Methoden hat. Diese werden anschließend in der *ServerProxy*-Klasse erzeugt. Außerdem wird jeder Methode die *RemoteException* hinzugefügt, um konform zu den *java.rmi*-Klassen zu bleiben.

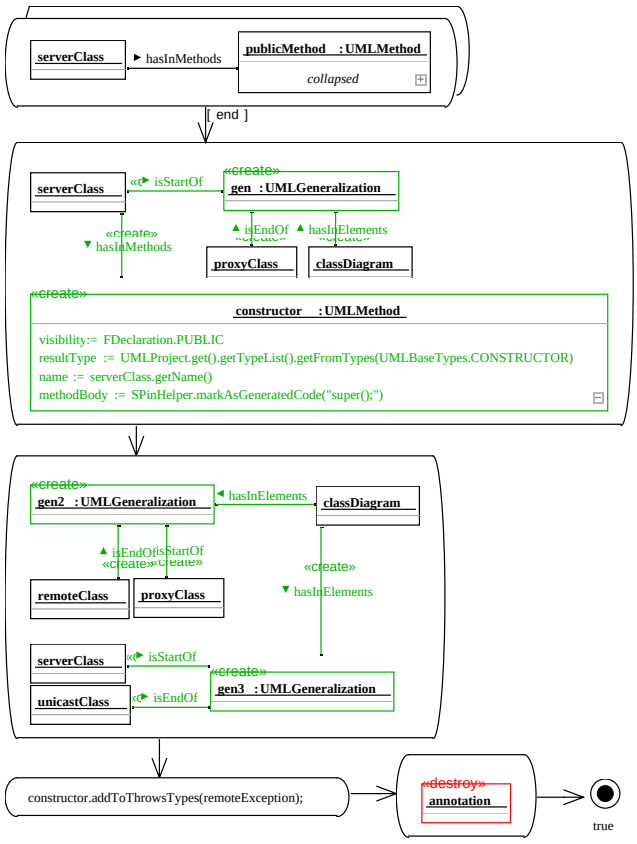

Abbildung 11. SPin - Apply Regel des 2. Beispiels

Nachdem in der Schleife keine noch nicht gematchten Methoden mehr gefunden werden, gelangt man mit der *[end]*-Kante (Abbildung 11) zur nächsten Aktivität. In dieser wird der 5. Schritt ausgeführt, also der Konstruktor für die *Server*-Klasse erstellt. Außerdem werden hier und in der Folge-Aktivität die Vererbungsbeziehungen zwischen den erstellten Klassen hergestellt (6. Schritt).

Die letzte Abbildung (Abb. 12) zeigt das Modell, nach dem Ausführen der Transformation auf der Annotation. Man sieht die hergestellten Vererbungsbeziehungen und die übernommenen *public*-Methoden im *ServerProxy*. Man kann außerdem die beiden Java Klassen *Remote* und *UnicastRemoteObject* durch SPin mit der Java VM synchronisieren (dazu wählt man einfach im Kontextmenü „sync with VM“), worauf ich allerdings hier nicht näher eingehen werde.

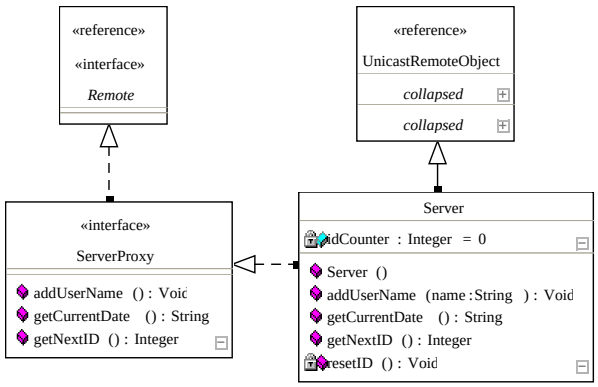

Abbildung 12. SPin - Transformiertes Modell

8. Dokumentation

Als Dokumentation zur Einarbeitung in Fujaba und SPin dienten mir zum Einen das SPin Paper [6] und die Diplomarbeit [5] von Felix Klar, in der das Fujaba Plugin entstanden ist und zum Anderen die Beschreibung eines Beispiels auf der Seite des Fachbereichs Metamodellierung [3]

Im Folgenden werden die Probleme bzw. Fragen, die beim Umsetzen der beiden Transformations-Beispiele wegen fehlender bzw. nicht ausreichender Dokumentation aufgetreten sind und deren Lösung besprochen.

- **Problem 1:** Nachdem das UML Klassendiagramm in Fujaba mit Annotationen annotiert wurde, muss eine Regel erstellt werden, die die Transformation des Models durchführt. Was ist der Unterschied im Kontextmenü *‚create rule...‘* von SPin zwischen *‚RefinementRule‘*, *‚TransformationRule‘* und *‚AbstractionRule‘*?
  Lösung: Verwirrende Bezeichnung: Dialog heißt create Transformation rule, man kann aber auch neben TransformationRule Abstraction- und RefinementRule erstellen. Eine RefinementRule verfeinert das Modell und eine AbstractionRule führt den umgekehrten Weg durch.
- **Problem 2:** Wieso wird bei der Erstellung einer Regel keine ‚apply‘ Methode eingefügt, wie es in den Dokumentationen angegeben ist?
  Lösung: Es muss zuerst das SPin Metamodell dem Projekt hinzugefügt werden. Das erfolgt über das Menü.

[3] http://www.mm.informatik.tu-darmstadt.de/research/arcstra/example/

- **Problem 3:** Warum funktioniert es nicht, wenn man mit Rechtsklick auf eine Annotation klickt und im Kontextmenü ‚open corresponding rule' wählt?
  Lösung: Der Name für die Regel im Dialog ‚create rule' muss gleich dem Namen der Annotation sein, für die diese Regel ausgeführt werden soll, sodass eine Zuordnung geschehen kann.
- **Problem 4:** Das Modell wird beim Aufrufen des Kontextmenüs einer Annotation und auswählen von ‚refine selected annotation(s)' nicht verfeinert.
  Lösung: Nur eine Neuinstallation des Fujaba Systems und erneutes Importieren von SPin konnte dieses Problem lösen.

Die Installation von Fujaba sowie von SPin verliefen ohne Probleme nach Beschreibung auf der Metamodellierung Webseite. Jedoch habe ich es nicht geschafft eine selbst erstellte Transformationsregel zu exportieren. Nach mehreren erfolglosen Lösungsversuchen mußte ich Fujaba und SPin neu installieren (mit den gleichen Versionen!) und konnte dann die Regeln einwandfrei exportieren.

Womit ich auch viele Probleme hatte war die *Undo*-Funktionalität beim Erstellen einer Regel. So kam es des öfteren vor, dass durch Drücken der <STRG><Z> Tastenkombination, die ganze Regel zerstört wurde und ich sie nicht wiederherstellen konnte.

Leider gibt es kein Benutzerhandbuch zu SPin, was die Einarbeitung sehr schwierig macht. Man muss stets in der Diplomarbeit nach entsprechenden Passagen suchen. Das Paper zu SPin [6] ist auch nur eine kleine SPin Einführung, man erhält hierdurch eine gute Einleitung in die Themen Architekturstratifikation und Story Driven Modelling, sowie weiteren Modellierungs-Hintergrund, der in Fujaba und SPin eine Rolle spielt. Das Beispiel auf der Metamodellierungs Webseite ist zu komplex, um eine sinnvolle Einleitung in die Benutzung von SPin mit Fujaba zu ermöglichen. Geeigneter wäre hier vielleicht eine Schritt für Schritt Erklärung, wie man ein einfaches Beispiel (z.B. Singleton Pattern) erstellt, annotiert, Verfeinerungsregeln definiert und letztendlich verfeinert (ohne die schon vorhandene Regel zu benutzen).

Sehr hilfreich bei der Definition der Transformationsregel war die Fujaba API[4], da man die entsprechenden Felder und Methoden, die man matchen möchte, nachschauen kann.

9. Zusammenfassung

Zunächst einmal steht der Einarbeitungsaufwand und der Aufwand den ich investieren musste, um die Transformationsregel für das erste Beispiel zu definieren, in keinem Verhältnis zum Ergebnis. Man stellt sich also die Frage: Welchen Sinn hat SPin und Architektur-Stratifikation?
Die Antwort ist: Es macht in diesem Beispiel insofern Sinn, als dass man sich nicht immer für die konkrete Repräsentation mit den getter-/setter-Methoden interessiert. Das könnte zum Beispiel der Fall sein, wenn man ein größeres Projekt aus einzelnen Klassen zusammenbaut, in denen viele getter-/setter-Methoden benötigt werden, die aber immer die selbe Funktion haben und man deswegen den Blick für wichtige Funktionen verliert. Man kann also ein Annotationsobjekt für diese Klassen erstellen und bei Bedarf der konkreteren Repräsentation (z.B. für Codegenerierung) das Modell verfeinern.

Es sei noch einmal wiederholt: Es geht bei Architektur-Stratifikation nicht darum sein Modell in Code zu transformieren, sondern - wie in der Einleitung beschrieben - zwischen den Abstraktionsgraden der Ansichten wechseln zu können. Dieser Ansatz unterstützt also den Softwareentwicklungsprozess maßgeblich: Nach einer Anforderungsanalyse entsteht das abstrakteste Stratum, das man jetzt nach und nach implementieren würde. Stattdessen versucht man bei der Stratifikation dieses abstrakte Stratum schrittweise zu verfeinern, indem man sich überlegt welche System-Teilaspekte wichtig sind, und verfeinert werden sollten. Dieser Vorgang ist viel intuitiver, denn anstatt sich Gedanken über Details der Implementierung zu machen, sucht man nur nach Mustern bzw. Teilaspekten des Systems (was auch teilweise durch die Anforderungsanalyse schon geschehen ist) und benutzt (im Idealfall schon vorhandene) Transformationsregeln um die gewünschte Transformation durchzuführen. Ein Nachteil dieses Modellierungskonzeptes ist, dass die Transformationsregel für ein bestimmtes Muster so konkret sein muss, dass sie genau auf dieses Muster passt und es in einer Programmiersprache umsetzt, aber auch speziell genug sein muss um Variationen des Musters umsetzen zu können. Im schlechtesten Fall würde man für einen System-Aspekt, den man transformieren möchte, mehrere verschiedene Regeln haben. Dadurch könnte die Anzahl der Transformationsregeln sehr groß werden und man würde den Überblick verlieren. Ein weiterer Nachteil, den Architektur Stratifikation hat, ist das man beim Umsetzen von Regeln eine gewisse Syntax „aufgezwungen" bekommt (Storydiagramme), die dann nicht immer intuitiv ist.
Der große Vorteil den SPin bietet um diesen Ansatz zu unterstützen, ist das Exportieren und Wiederverwenden von Transformationsregeln. Dadurch braucht man nur einmal die „Denkarbeit" zu leisten.

Nach der sehr langen Einarbeitungsphase und erfolgreichen Umsetzung des ersten Beispiels war das Umsetzen des zweiten - etwas komplexeren Beispiels - verhältnismäßig einfach, da der Ablauf zum Aufbau einer Regel immer ähnlich ist. Wenn man einmal das Konzept von Storydiagrammen und Pattern-Matching verstanden hat, lässt es sich in Fujaba/SPin leicht umsetzen, da die entsprechenden Diagrammelemete sehr intuitiv sind.

Wie schon im ersten Transformationsbeispiel kurz angesprochen, wäre es wünschenswert, in Fujaba Storydiagrammen, die Möglichkeit zu haben, zwischen den Ansichten in den Pattern-Matching Aktivitäten zu wechseln: Nach der Modellierung der Kante zwischen UML Objekten könnte man in eine Sicht wechseln, wo man die konkrete Syntax (Abbildung 13), also die Annotation und die referenzierten Klassen sieht. Umgekehrt würde man zum Modellieren in die konkretere Ansicht (Abbildung 14) wechseln, in der

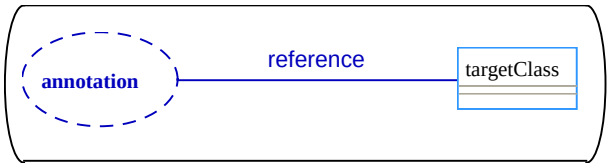

Abbildung 13. SDM Aktivität - Abstrakte Sicht

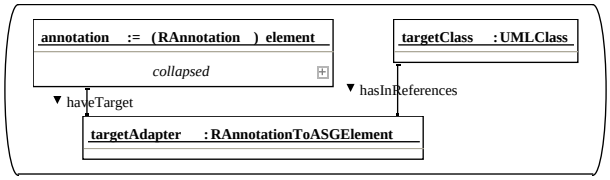

Abbildung 14. SDM Aktivität - Konkrete Sicht

[4] http://wwwcs.uni-paderborn.de/cs/fujaba/documents/developer/Fujaba/helpdoc.html

man direkt auf die Elemente der Kante zugreifen und diese somit modellieren kann.

Literatur

[1] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. *Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language and Java*. tech. report, AG-Softwaretechnik, Fachbereich 17, Universität Paderborn, 1999.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Entwurfsmuster*. Addison-Wesley, 1996.

[3] M. Girschick, T. Kühne, F. Klar. *Generating Systems from Multiple Levels of Abstraction*. International Conference on Trends in Enterprise Application Architecture, 2006.

[4] L. Grunske, L. Geiger, A. Zündorf, N. Van Eetvelde, P. Van Gorp, and D. Varró *Using Graph Transformation for Practical Model Driven Software Engineering*. in Sami Beydeda, Matthias Book, Volker Gruhn (eds.): Model-driven Software Development, pp. 91-118, Springer (2005).

[5] F. Klar. *SPin - Ein Werkzeug zur Realisierung von Architektur-Stratifikation*. Diplomarbeit, Technische Universität Darmstadt, April 2005.

[6] F. Klar., T. Kühne, M. Girschick. *SPin - A Fujaba Plugin for Architecture Stratification*. in: Giese, H. & Zündorf, A. (eds.): Proceedings of the 3rd Int. Fujaba Days 2005: MDD in Practice, pp. 17 - 23, 15.-18. September 2005.

Anhang

A. apply-Regel des zweiten Beispiels