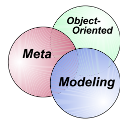


Fachbereich Informatik
Fachgebiet Metamodellierung
Prof. Dr. Thomas Kühne

TECHNISCHE UNIVERSITÄT DARMSTADT

Diplomarbeit

SPin
—
Ein Werkzeug zur Realisierung von Architektur-Stratifikation

vorgelegt von

Felix Klar

Betreuer: Dipl.-Inform. Martin Girschick

Tag der Ausgabe: 01.11.2004 — Tag der Abgabe: 20.04.2005

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Diplomarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 20.04.2005

Felix Klar

Danksagung

An erster Stelle möchte ich meinen Eltern danken, ohne deren finanzielle Unterstützung mein Studium nicht möglich gewesen wäre. Desweiteren geht mein Dank an Tobias Rötschke, der mir in der Anfangsphase meiner Arbeit sehr dabei geholfen hat mich mit Fujaba vertraut zu machen, an die Fujaba-Entwickler, die immer schnelle und gute Unterstützung bei auftretenden Fragen und Problemen gaben, an meinen Betreuer Martin Girschick und an Herrn Prof. Kühne, die mir oft wertvolle Hinweise geben konnten und immer ein offenes Ohr für mich hatten. Abschließend geht mein besonderer Dank an Elisabeth, die mich während meiner Diplomarbeit ertragen musste und durch die es mir möglich wurde meine Ideen und Gedankengänge zu entwickeln. Ihr ist diese Arbeit gewidmet.

Abstract

Diese Diplomarbeit beschäftigt sich mit Stratifikation – einem Modellierungskonzept, das eine System-Architektur als eine Menge von Abstraktionsebenen auffasst. Jede Ebene stellt das System in einem anderen Abstraktionsniveau dar. Desweiteren ist es möglich zwischen diesen Ebenen zu navigieren. So kann ein stratifiziertes System aus verschiedenen Perspektiven betrachtet werden. Komplexe Sachverhalte sind übersichtlicher darstellbar und Änderungen am System können auf hoher Abstraktionsebene erfolgen. Im Rahmen dieser Arbeit wird das Konzept der Stratifikation konkretisiert und als Werkzeug implementiert. Es trägt den Namen SPin und realisiert die Navigation zwischen den Ebenen mit Hilfe von sogenannten *Abstraktions-* und *Verfeinerungsregeln*. Diese können von *Regelentwicklern* erzeugt, getestet und danach *Regelanwendern* zur Verfügung gestellt werden. Damit unterstützt SPin Modellierer und Softwareentwickler bei ihrer Arbeit, weil bestimmte Vorgänge mit Hilfe von Regeln automatisiert werden können. Beispielsweise ist es möglich einem Modell typische Entwurfsmuster und Aspekte hinzuzufügen. Die mit SPin anwendbaren Regeln sind zudem parametrisierbar und ermöglichen dadurch verschiedene Realisierungen eines Entwurfsmusters oder Aspektes. Regeln können so implementiert werden, dass sie eine Modelltransformation durchführen. Auf diese Weise sind mit SPin *Abstrakte Syntaxgraphen*, also beispielsweise UML-Modelle, stratifizier- und transformierbar.

This diploma thesis investigates stratification – a modeling-concept that understands a systems architecture as a number of abstraction levels that contain equivalent information. Navigation between those levels is possible and so a stratified system can be viewed from different point of views. Complex facts may be visualized in a compressed form and system changes can be performed on high abstraction levels. In this work stratification is concretized and implemented as a tool named SPin. It realizes navigation between those levels by so called abstraction and refinement rules. They can be created and tested by rule developers and given to rule users. Therefore SPin supports modelers and software developers, because with those rules typical processes such as design patterns and aspects may be automated. Rules that can be applied with SPin are parameterizable and therefore allow different variations of patterns and aspects. Rules can be implemented to perform a model transformation. This way *abstract syntax graphs* such as UML-models are able to be stratified and transformed by SPin.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Ziel der Arbeit . . . 2

1.2 Struktur der Arbeit . . . 3

1.3 Allgemeine Hinweise . . . 3

2 Stratifikation **4**

2.1 Einsatzmöglichkeiten von Stratifikation . . . 5

2.2 Realisation . . . 7

3 Technische Grundlagen **8**

3.1 Begriffsklärung . . . 8

3.1.1 Was ist „meta"? . . . 8

3.1.2 MDA, MOF und XMI . . . 9

3.2 Eclipse . . . 10

3.3 Auswahl der Grundlage für SPin . . . 11

3.3.1 ArgoUML . . . 12

3.3.2 EclipseUML . . . 12

3.3.3 Fujaba . . . 13

3.3.4 Together . . . 13

4 Fujaba **14**

4.1 Allgemeines . . . 14

4.2 Modellierungs Framework . . . 14

4.3 Story Driven Modeling (SDM) . . . 16

4.4 Modelltransformation mit SDM 18

5 SPin 20

5.1 Anforderungen 20

5.2 Architektur 21

5.2.1 Annotation 21

5.2.2 Transformationsregeln 25

5.2.3 Transformation 32

5.2.4 Hilfskonstrukte 34

5.3 Userinterface 40

5.4 Implementierungsdetails 44

5.4.1 Fujaba Quellen 44

5.4.2 Überblick über Klassen von SPin 44

5.4.3 Debuggen von Transformationsregeln 46

6 Ergebnisse 47

6.1 Stratifizierung 47

6.2 Automatisierung 47

6.3 Hot-Plug-in Module 48

6.4 Evolution 49

6.5 Graphische Templates 49

6.6 Erstellte Transformationsregeln 53

7 Fallbeispiel 57

8 Ausblick 65

Literaturverzeichnis 69

Index 71

A Ausführliche Diagramme 71

B Transformationsregeln 73

Abbildungsverzeichnis

2.1 Stratifikation: Schema 5

3.1 UML: Modellierungs Framework 10

4.1 Fujaba: *Abstract Syntax Graph* (ASG) 15
4.2 Fujaba: UML Metamodell für Klassendiagramme 16
4.3 Fujaba: UML Metamodell für Storydiagramme 17
4.4 Fujaba: Storydiagramm-Beispiel 18
4.5 Modelltransformation: Ablauf 19

5.1 SPin: Teil des Annotierungs-Metamodells 22
5.2 SPin: komplettes Annotierungs-Metamodell 23
5.3 SPin: Annotierung einer *interaction refinement* 25
5.4 SPin: Regeln und deren Verwaltung 27
5.5 SPin: Grundstruktur einer *TransformationRule* 27
5.6 SPin: Informationsstruktur einer *TransformationRule* 28
5.7 SPin: Grundstruktur einer *AbstractionRule* 29
5.8 SPin: Grundstruktur einer *RefinementRule* 30
5.9 SPin: Transformator 32
5.10 SPin: Fabriken 34
5.11 SPin: Synchronisations-Modul 37
5.12 SPin: Synchronisations-Modul – Anwendung 38
5.13 SPin: Methoden-Änderungsmechanismus 38
5.14 SPin: Hilfskonstrukt für Allgemeines 39

5.15 SPin: Mechanismus zur Kommunikation mit Benutzer 40
5.16 SPin: Toolbar und Kontextmenü für UML Klassendiagramme . . . 41
5.17 SPin: Dialog zum Erstellen einer Regel 41
5.18 SPin: Annotierungs-Editor 42
5.19 SPin: Einstellungen 44
5.20 SPin: Beziehungen zwischen SPin und Fujaba 45

6.1 Template-Beispiel: Framework-Implementierung 51
6.2 Template-Beispiel: Annotierung 52
6.3 Template-Beispiel: Template-Parametrisierung 52
6.4 Template-Beispiel: transformierte Annotierung 52
6.5 Template-Beispiel: generierte Implementierung 53

7.1 Fallbeispiel – Simulator: abstrakteste Ebene 57
7.2 Fallbeispiel – Simulator: System nach Verfeinern von „JavaApplication“ 60
7.3 Fallbeispiel – Simulator: System nach Verfeinern von „Observer“ . 61
7.4 Fallbeispiel – Simulator: System nach Verfeinern von „Visitor“ . . . 62
7.5 Fallbeispiel – Simulator: System nach Verfeinern von „Singleton“ . 62
7.6 Fallbeispiel – Simulator: System nach Verfeinern von „Logged“ . . 63
7.7 Fallbeispiel – Simulator: konkreteste Ebene 64

A.1 SPin: Fabriken (volles Detail) 72

B.1 Regel (RR): DeleteMethods 73
B.2 Regel (RR): RepairUMLBaseTypes 74
B.3 Regel (RR): CleanTypeList 75
B.4 Regel (AR): DetectAttributes 76
B.5 Regel (RR): DetectAttributes 76
B.6 Regel (RR): MethodModification 77
B.7 Regel (RR): MethodNameModification 78
B.8 Regel (RR): Singleton 79
B.9 Regel (RR): Visitor 80

B.10 Regel (RR): Observer 81
B.11 Regel (RR): Logged 82
B.12 Regel (RR): JavaApplication 83

Tabellenverzeichnis

5.1 SPin: Zuordnung von Präfix zu Transformationsregel 26
5.2 SPin: Zuständigkeit von Transformatoren für Regeln 33

7.1 Fallbeispiel: Links an Annotationen 60
7.2 Fallbeispiel: zu implementierende Methoden 63

Kapitel 1

Einleitung

„keep it simple — make it abstract!“

– FK

Der Bereich der Softwareentwicklung hat sich im Laufe der letzten 50 Jahre stark verändert. War es zu Beginn noch nötig einen Computer in Maschinensprache zu programmieren, wurde es mit der Einführung höherer Programmiersprachen wie FORTRAN möglich, Systeme in Sprachen zu entwickeln, die dem menschlichen Denken näher liegen. Mitte der 90er Jahre wurde der nächste Schritt gemacht und für Analyse- und Designmethoden eine Notation entwickelt, aus der später die *Unified Modeling Language* (*UML*) entstand. Bei der UML[1] handelt es sich um die Vereinheitlichung der graphischen Darstellung und Semantik von Modellierungselementen. Damit wurde es möglich Sichten auf Softwaresysteme graphisch standardisiert zu beschreiben. Die graphische Modellierung macht es einem Entwickler einfacher ein fremdes System zu verstehen und eigene Systeme zu entwickeln. So sind Zusammenhänge in einem graphischen Modell besser zu erkennen als in geschriebenem Text. Mit Hilfe von sogenannten *CASE-Tools* (*Computer Aided Software Engineering*) kann man ein System graphisch modellieren. CASE-Tools sind Programme, die Computer-unterstützte Softwareentwicklung auf der Basis einer Modellierungssprache ermöglichen. Eine Reihe dieser Programme bieten neben der Möglichkeit des Graphischen Modellierens auch das Generieren von Code aus dem Modell. Das modellgetriebene Entwickeln ermöglicht es unter anderem, die Entwicklungszeit für Systeme zu reduzieren und das Erzeugen von bestimmten Konstrukten zu automatisieren. Der Programmierer wird zum Modellierer und kann das System auf einer höheren Abstraktionsebene beschreiben und aus einer anderen Perspektive betrachten.

[1] http://www.uml.org/

Die Entwicklung eines Systems mit einem CASE-Tool hilft zwar auf der einen Seite das System zu entwicklen, auf der anderen Seite stellt sich jedoch das Problem, dass aufgrund der hohen Komplexität vieler Systeme die graphische Visualisierung des Systems sehr schnell unübersichtlich wird. So kann dieses nicht mehr vernünftig dargestellt werden.

Es gibt verschiedene Möglichkeiten Übersichtlichkeit in Diagrammen zu gewährleisten:

- Elemente kollabieren, also bestimmte Informationen verbergen
- Elemente farblich markieren
- Einsatz von Texturen und unterschiedlichen Schriftstilen
- Einarbeitung von Designkonzepten wie Kontrast, Ausrichtung, Wiederholung und Nachbarschaft
- Aufteilen von zu komplexen Diagrammen in mehrere einfachere Diagramme

Strukturelle Informationen wie Assoziationen und Vererbung bleiben bei diesen Ansätzen jedoch erhalten und beeinträchtigen weiterhin die Übersichtlichkeit. Beispiele für die Umsetzung der oben genannten Möglichkeiten findet man z.B. in [9].

Ein anderer Ansatz besteht darin das Modell zu abstrahieren, also weniger Details darzustellen. Heutzutage muss ein Modellierer dies selbst bewerkstelligen, indem er das Modell von Hand abstrahiert und die gewünschte Sicht manuell erstellt. Ändert er das Modell in der abstrahierten Sicht, wird die konkretere Sicht nicht aktualisiert und umgekehrt. Man hat also immer nur Momentaufnahmen, die nicht synchron sind. Zudem sind die jeweiligen Sichten in Bezug auf die enthaltenen Informationen nicht vollständig. Nur in ihrer Gesamtheit sind alle Informationen vorhanden.

Eine Lösung für dieses Problem bietet Architekturstratifikation (im Folgendenen nur noch Stratifikation genannt), die in [3] von Atkinson und Kühne vorgestellt wurde. Stratifikation ermöglicht es, eine Architektur als geordnete Hierarchie von Architektur-Ebenen unterschiedlichen Abstraktionsgrades zu beschreiben. Die oberste Ebene enthält die abstrakteste Sicht auf die Architektur und somit deren einfachste Darstellung, die unterste Ebene enthält die ausführlichste. Zwischen den Ebenen kann gewechselt werden, so dass es möglich wird die Architektur aus unterschiedlichen Perspektiven zu betrachten und zu bearbeiten.

1.1 Ziel der Arbeit

Ziel dieser Arbeit ist es ein Werkzeug zu entwickeln, das Stratifizierung ermöglicht. Zum Einsatz kommen soll es zunächst in Klassendiagrammen der UML, weil sie

ein weit verbreiteter Standard im Bereich der Software-Modellierung ist. Zudem sind Klassendiagramme die Kerndiagramme der UML. Sie legen die Struktur der Elemente eines objektorientierten Systems fest [8]. Wie sich in Kapitel 5 zeigen wird, ist das Werkzeug jedoch nicht auf UML Klassendiagramme beschränkt.

Das im Rahmen dieser Arbeit entstandene Werkzeug ist als Plugin für das CASE-Tool *Fujaba* [20] realisiert und trägt den Namen *SPin*[2] [14]. Das Werkzeug soll graphische Modellierung vereinfachen und Softwareentwicklern einen besseren Überblick über ein komplexes System ermöglichen. Dies wird durch Betrachten und Modifizieren des Modells auf verschiedenen Abstraktionsebenen erreicht. Desweiteren wird es durch SPin möglich, einem System bestimmte Aspekte hinzuzufügen oder diese wieder zu entfernen.

1.2 Struktur der Arbeit

Kapitel 2 gibt eine Einführung in die Stratifikation und zeigt einige Besonderheiten auf, die bei Stratifikation zu beachten sind. Technische Grundlagen, die zum Verständnis dieser Arbeit beitragen und einige CASE-Tools, die als Basis für das zu entwickelnde Werkzeug in Frage kommen, werden in Kapitel 3 vorgestellt. In Kapitel 4 wird das ausgewählte CASE-Tool ausführlicher betrachtet. Kapitel 5 stellt die Architektur von SPin vor. In Kapitel 6 werden die Ergebnisse dieser Arbeit präsentiert und in Kapitel 7 wird die Anwendung von SPin für Stratifikation anhand eines Beispiels erläutert. Abschließend gibt Kapitel 8 einen Ausblick über Erweiterungsmöglichkeiten von SPin.

1.3 Allgemeine Hinweise

Diese Arbeit setzt Kenntnisse über das Lesen von UML Diagrammen voraus. Dem geneigten Leser sei als Einführung in die UML [8] empfohlen. Eine Übersicht über Diagramme und Elemente der UML ist unter [16] zu finden.

Es ist zu beachten, dass mit Ausdrücken wie *Programmierer*, *Entwickler* immer auch die weibliche Form gemeint ist, jedoch aus Gründen der Lesbarkeit nicht verwendet wird. Dies ist keinesfalls diskriminierend gemeint.

[2] SPin ist ein Akronym für '**S**tratification **P**lug**in**'

Kapitel 2

Stratifikation

Stratifikation ist ein Modellierungskonzept, das ein zu entwickelndes System als eine Hierarchie von Architektur-Ebenen auffasst. Diese weisen untereinander eine partielle Ordnung auf [5]. Eine einzelne Ebene wird als *Stratum* bezeichnet (n.; -a). Die oberste Ebene enthält die abstrakteste Sicht auf das System und ähnelt dem Ergebnis einer Anforderungsanalyse. Das unterste Stratum beschreibt das System im vollen Detail. Das jeweils nächste Stratum wird durch Verfeinerung (*engl.: refinement*) von Komponenten und deren Interaktion erzeugt. Von einer konkreteren Ebene kann durch Abstraktion zur nächst höheren gewechselt werden. Damit ist eine Navigation zwischen Strata möglich. Auf allen Ebenen ist das beschriebene System semantisch äquivalent, nur die Fülle der Semantiken von Interaktionen und Datenrepräsentation sind verschieden [2]. Jede dieser Ebenen ist in einem bestimmten Kontext sinnvoll. Will man die Gesamtstruktur des Systems verstehen, schaut man sich die höchste Ebene an. Die Sicht auf eine der unteren Ebenen ist hilfreich, wenn man die Implementierung des Systems ändern oder sich mit Details auseinandersetzen möchte. Das System kann also aus verschiedenen Perspektiven betrachtet werden.

Ein Beispiel für Stratifikation eines Systemausschnitts ist in Abbildung 2.1 dargestellt. Die obere Ebene enthält Komponenten, die mittels einer Annotation markiert sind. Diese gibt an, dass sie das Visitor-Pattern [11] realisieren sollen. Anhand der Annotation kann die nächste Ebene generiert werden, die nun eine Implementierung des Visitor-Pattern enthält. Es sei darauf hingewiesen, dass Strata nicht den Ebenen im UML Modellierungsframework entsprechen (siehe Abbildung 3.1). Eine Navigation zwischen den Architektur-Ebenen verlässt nicht die Modell-Ebene.

Übergänge zwischen Strata werden durch Relationen beschrieben und mit *Transformation* bezeichnet. Mit Hilfe von Relationen können Übergänge automatisiert werden. Jene sind ungerichtet und somit unabhängig von der Richtung, in der das System entwickelt werden soll. Ein *bottom-up*- und ein *top-down*-Ansatz sind

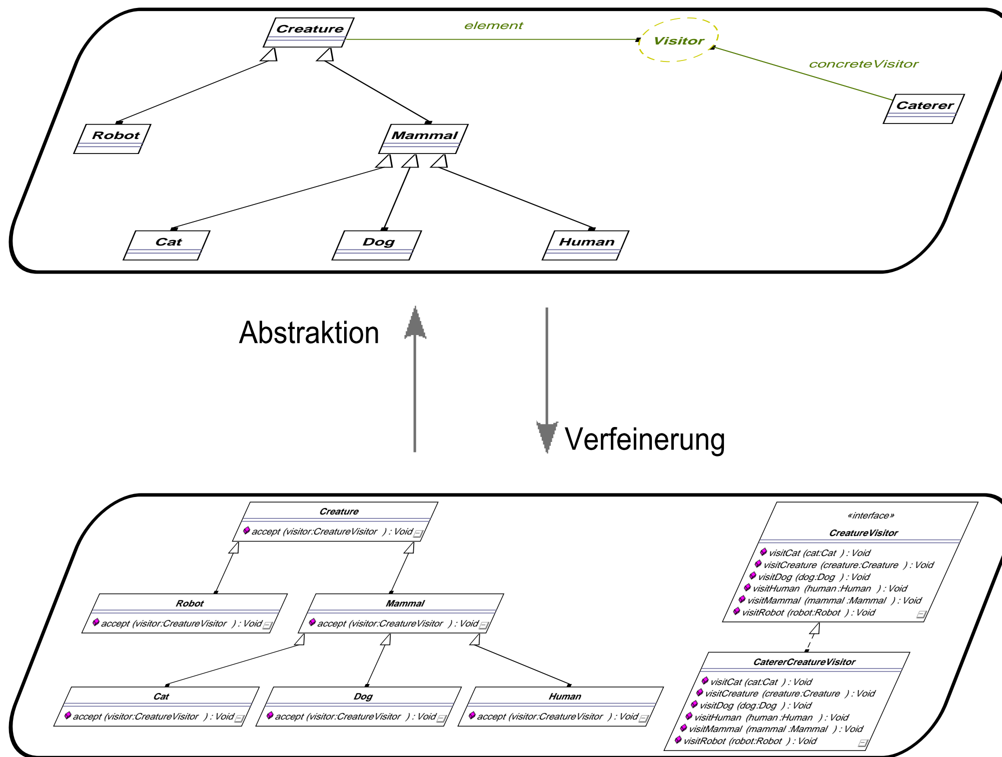


Abbildung 2.1: *Beispiel für Stratifikation am Beispiel des Visitor-Patterns*

deshalb möglich. Wird von einer Ebene abstrahiert, werden bestimmte Details irrelevant und können in Form von Annotierungen abstrakter dargestellt werden. Details werden dabei nicht gelöscht, sondern anders visualisiert. Sie können im Verfeinerungsprozess rekonstruiert werden.

2.1 Einsatzmöglichkeiten von Stratifikation

Softwareentwicklung ist im Allgemeinen ein iterativer Prozess und das Modell veraltet, sobald am Code Änderungen vorgenommen werden. Stratifikation bietet hierfür eine Lösung, da sowohl Modell als auch Code in den Entwicklungsprozess einbezogen sind und Änderungen des einen Auswirkungen auf das andere haben.

Stratifikation ist dafür ausgelegt Verfeinerung von Interaktionen (*engl.: interaction refinement*) zu unterstützen [3]. Einer Interaktion wird ein bestimmter Sachverhalt zugewiesen, beispielsweise „sichere Datenübertragung“. Dieser kann mit Hilfe einer Relation transformiert werden. Das so entstehende System enthält neue oder er-

weiterte Komponenten, die den Sachverhalt realisieren. Bestehende Komponenten müssen evtl. angepasst werden.

Bisherige Ansätze zum *aspektorientierten Programmieren* (*AOP*) [13] sind auf der Implementierungsebene definiert und basieren deshalb auf einem niedrigen Abstraktionslevel. Will man mehrere Aspekte mit Code „verweben", kommt es zu Problemen, da keine Ebenen mit hohem Abstraktionsgrad zur Verfügung stehen, auf denen Aspekte verankert werden könnten. Deshalb kann es an den Stellen zu Konflikten zwischen Aspekten kommen, an denen sich mehrere Aspekte überlagern. Je nach Überlagerungsstelle muss ein jeweils anderer Aspekt dominierend behandelt werden. So ist keine allgemeingültige Strategie zur Verwebung konkurrierender Aspekte formulierbar (siehe [4] und [15]). Stratifikation kann als Ergänzung zum AOP gesehen werden. Mit ihrer Hilfe kann man einem System auf hoher Abstraktionsebene Aspekte hinzufügen. Zusätzlich wird – durch die Abstraktionsebenen-Sichtweise – zwischen Aspekten eine natürliche Ordnung erzeugt. Diese liegt darin begründet, dass bestimmte Aspekte von anderen abhängig sind[1] und erst auf niedrigerer Abstraktionsebene eingeführt werden. Der Kontext für eine Aspektanwendung liegt immer auf der nächst höheren Abstraktionsstufe vor. So werden abhängige Aspekte voneinander getrennt und deren hierarchischer Natur Rechnung getragen. Um Aspekte einzuführen werden Annotationen verwendet.

Wie bereits in Abbildung 2.1 gezeigt wurde, ist es möglich (Entwurfs-)Muster anhand von Stratifikation auf hoher Abstraktionsebene einzuführen. In Architekturen, die nur einen Abstraktionslevel aufweisen, ist es ohne weitere Dokumentation schwer zu ersehen, dass ein Muster angewendet wurde und in welchen Komponenten das Muster realisiert ist. Stratifikation bietet die Möglichkeit, Muster auf einer Abstraktionsebene einzuführen, die ihrem „natürlichen" Abstraktionsgrad entspricht [4]. Wie im Beispiel des Visitor-Patterns (Abbildung 2.1) zu sehen ist, kann so auf hoher Abstraktionsebene beschrieben werden, dass ein Lieferant Speisen an Kreaturen ausliefern soll.

Stratifikation kann dazu verwendet werden komponentenbasierte Frameworks flexibler zu machen [3]. Sie ermöglicht es, ein Framework in einem objektorientierten Ansatz zu erweitern und zu parametrisieren. Traditionelle Framework-Ansätze erlauben das Anpassen und Austauschen von Datentypen und Umdefinieren von Verhalten. Eine Stelle, an der Variationen durchgeführt werden können, wird als *hot spot* bezeichnet. Stratifikation verteilt diese in den Strata, die ihren natürlichen Abstraktionsebenen entsprechen. In einem Framework sind Interaktionen normalerweise vordefiniert und fix. In einem stratifizierten Framework ist es dagegen möglich, Interaktionen anzupassen oder zu ändern, indem auf hoher Abstraktionsebene die Komponenten geändert werden, welche in die Interaktion einbezogen sind [4].

[1] „Verschlüsselung" und „Authorisation" sind beispielsweise abhängig von „Sicherheit", da die beiden erstgenannten Aspekte Verfeinerungen von „Sicherheit" sind

2.2 Realisation

Stratifikation ist mit jeder Beschreibungssprache realisierbar. Es kann sowohl auf einer graphischen Modellierungssprache wie UML als auch auf einer textuell basierten Programmiersprache aufgebaut werden. Eine Modellierungssprache ist für die Zwecke der Stratifikation jedoch geeigneter, denn durch sie kann auf einer abstrakteren Ebene gearbeitet werden.

Verwendet man eine Modellierungssprache, spricht man anstelle von einem zu entwickelnden System auch von einem Modell, welches das System beschreibt oder kurz von Modell.

Kapitel 3

Technische Grundlagen

In diesem Kapitel werden zunächst technische Grundlagen geklärt. Daran anknüpfend wird kurz die Entwicklungsumgebung Eclipse vorgestellt und abschließend der Auswahlprozess des Programmes, auf dem SPin aufsetzen soll, erläutert.

3.1 Begriffsklärung

Für das weitere Verständnis ist es unerläßlich sich einige Begriffe und Standards des Bereichs der Software-Modellierung vor Augen zu führen. Begriffe wie „meta“, MDA, MOF und XMI sollen deswegen nachfolgend kurz erläutert werden.

3.1.1 Was ist „meta“?

Unter „*meta*“ ist zunächst einmal „etwas höher stehendes“, also etwas, das auf einer höheren Ebene anzusiedeln ist und das im Allgemeinen einen höheren Abstraktionsgrad aufweist, zu verstehen. Wenn von „*Metamodell*“ gesprochen wird bedeutet dies, dass für ein Modell ein höheres Modell (ein Metamodell) existiert, welches das Modell beschreibt. Ein zu einem Metamodell zugehöriges Modell wird mit „*Instanz*“ bezeichnet. Dies sei an einem Beispiel verdeutlicht:

> *„Ein UML Klassendiagramm enthält Klassen, die jeweils einen eindeutigen Namen tragen und eine Anzahl von Attributen und Methoden, die jeweils einer Klasse zugeordnet werden. Klassen stehen über Assoziationen in Beziehung zu anderen Klassen.“*
>
> Diese formale Beschreibung eines UML Klassendiagramms ist ein Metamodell, das die generelle Struktur von UML Klassendiagrammen festlegt. Eine Instanz dieses Metamodells ist ein Diagramm, das als

Elemente eine Klasse Namens *Test* und eine Methode 'getHelloWorld(): String' enthält, die der Klasse *Test* zugewiesen ist.

Je weiter man von etwas abstrahiert, umso mehr bewegt man sich auf höhere Metaebenen zu. So kann man von einem Metamodell noch weiter abstrahieren und erhält ein „*Meta-Metamodell*“, mit dem ein Modell gemeint ist, das Modelle von Modellen beschreibt. Die formale Beschreibung eines Meta-Metamodells für UML Klassendiagramme könnte wie folgt lauten: *„Instanzen dieses Meta-Metamodells sind Diagramme, die Elemente enthalten können.“* bzw. *„Diagramme enthalten kein oder mehrere Elemente.“*
Mit diesem Meta-Metamodell hat man auf einer hohen Abstraktionsebene die Gemeinsamkeiten von Diagrammen beschrieben.

3.1.2 MDA, MOF und XMI

Hinter diesen drei Abkürzungen verbergen sich Standards, die von der *OMG*[1] (Object Management Group) entwickelt werden, um Softwareentwicklungsprozesse zu beschleunigen, zu vereinfachen und deren Qualität zu steigern. Der MDA-Ansatz basiert unter anderem auf den Standards MOF, XMI und UML. Für ausführlichere Informationen sei auf die jeweiligen Spezifikationen der Standards verwiesen, sowie auf [17].

MDA[2] (Model Driven Architecture) ist ein noch junger Standard der OMG. Er zielt darauf ab Software-Systeme allein anhand von Modellen zu erstellen, die sowohl graphisch als auch textuell sein können. Dabei muss die Bedeutung von MDA-Modellen formal eindeutig sein. MDA-Modelle werden in der Regel in UML definiert, können jedoch auch in klassischen Programmiersprachen modelliert werden. Ziel der MDA ist die bessere Handhabbarkeit von Komplexität durch Abstraktion und Generierung von Code aus formal eindeutigen Modellen. So soll Programmierung auf einer abstrakteren Ebene, nämlich durch Bearbeiten von Modellen, möglich werden.

An dieser Stelle wird deutlich, dass der Ansatz der Stratifikation als Teil der MDA gesehen werden kann und die Kombination von CASE-Tool, Stratifikation, formalen Modellen und Codegenerator eine Realisierung von MDA ist.

MOF[3] (Meta Object Facility) ist ein Meta-Metamodell, das im *UML Modellierungs Framework* (siehe Abbildung 3.1) auf Ebene M_3 angesiedelt ist. In diesem Framework ist jede Ebene M_i eine Instanz von M_{i+1}, $i \in \{0, 1, 2\}$. Mit MOF ist es

[1] http://www.omg.org/
[2] http://www.omg.org/mda/
[3] http://www.omg.org/mof/

möglich modellierte Daten und Modelle über Modellierungssprachgrenzen hinweg auszutauschen.

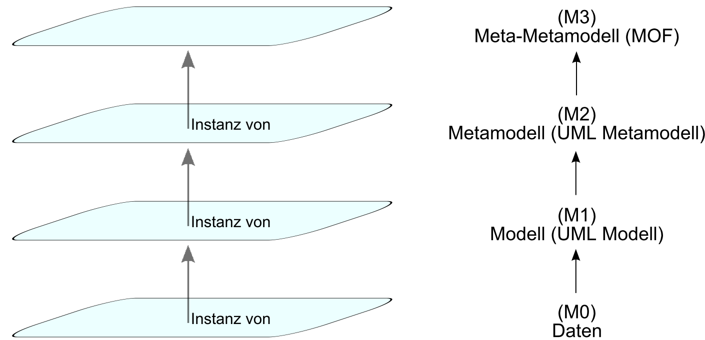


Abbildung 3.1: *Das UML Modellierungs Framework*

XMI[4] (XML Metadata Interchange) ist ein Format, das verwendet wird um MOF-Modelle auf XML (eXtensible Markup Language)[5]abzubilden. XMI definiert, wie XML-Tags benutzt werden um MOF-konforme Modelle im XML-Format zu speichern. MOF-basierte Metamodelle (M_2-Ebene) werden dazu in XML DTDs (XML Document Type Definitions) abgebildet und die korrespondierenden Modelle (M_1-Ebene) werden in XML-Dokumente übersetzt, die die Grammatik der DTD befolgen.

3.2 Eclipse

Eclipse[6] ist eine relativ populäre freie Entwicklungsumgebung, die nicht auf bestimmte Programmiersprachen beschränkt ist. Sie wurde von IBM im November 2001 als Open-Source-Projekt veröffentlicht. Eclipse stellt einen Plug-in Mechanismus bereit, an den sich Plug-ins für jede Art von zu entwickelnder Sprache anhängen können. Sowohl textuelle als auch graphische Elemente können mit Eclipse bearbeitet werden. Man benötigt dazu lediglich ein Plug-in, das die benötigte Funktionalität bereitstellt. So bietet beispielsweise das in Eclipse standardmäßig enthaltene Plugin *JDT* (*Java Development Tools*) gute Unterstützung bei der Entwicklung in Java. SPin, das in Java implementiert ist, wird deswegen mit Hilfe von Eclipse entwickelt.

[4] http://www.omg.org/technology/xml/
[5] http://www.w3.org/XML/
[6] http://www.eclipse.org

Ein CASE-Tool, das sich als Plug-in in Eclipse integriert, wäre optimal, da in diesem Fall die Vorteile von graphischer und textueller Entwicklung in einer Entwicklungsumgebung vereint wären und man Funktionalität von anderen Plug-ins mitbenutzen könnte. Tatsächlich existieren für Eclipse bereits Plug-ins für graphische Modellierung und einige CASE-Tool Entwickler bieten neben der „Standalone“ Variante eine Version für Eclipse an. Die Infrastruktur, die für die Realisierung eines CASE-Tools in Eclipse benötigt wird, stellt Eclipse bereits in Form von Plug-ins zur Verfügung. Diese Plug-ins sind namentlich EMF und UML2, deren Beziehung zu den oben vorgestellten Standards erläutert werden soll.

EMF (Eclipse Modelling Framework) [1] ist ein Modellierungs Framework für Eclipse, welches auf dem Meta-Metamodell *Ecore* arbeitet. EMF wird von IBM entwickelt und wurde Ende 2002 zum Open Source Projekt für Eclipse erklärt. Bezogen auf MOF ist EMF eine Implementierung eines Teils von MOF. Dieser Teil ist Ecore und korrespondiert mit einer Teilmenge der neuesten MOF-Spezifikation (MOF 2.0), die *EMOF* (Essential MOF) genannt wird. EMF und Ecore sind also auf der M_3-Ebene anzusiedeln. Mit EMF kann man die Struktur von Modellen programmatisch abfragen und ändern, außerdem kann EMF aus Modellen (Java-) Code generieren und Modelle im XMI-Format speichern. Eine gute Einführung in EMF gibt [7].

UML2[7] ist ein Eclipse Projekt, das eine EMF-basierte Implementierung des UML 2.0 Standards der OMG für Eclipse darstellt. UML2 ist für die Datenhaltung und Verwaltung verantwortlich und stellt eine API zur Verfügung, die von CASE-Tool-Entwicklern genutzt werden kann, um UML Modelle zu visualisieren.

3.3 Auswahl der Grundlage für SPin

Die Fragen, die es nun zu stellen gilt, sind: „Wie kann man Stratifikation realisieren?“ und „Auf welcher Plattform baut man auf?“.

Zur Durchführung von Stratifikation benötigt man ein graphisches Modell, das bearbeitet und transformiert werden kann. Es bietet sich an auf einem CASE-Tool aufzubauen. Selbst ein CASE-Tool zu entwickeln, das Stratifikation ermöglicht, wird ausgeschlossen, da die Entwicklung eines CASE-Tools, das heutigen Ansprüchen genügt, den Rahmen dieser Arbeit sprengen würde. Viele CASE-Tools besitzen zudem einen Mechanismus, mit dessen Hilfe man dieses erweitern kann – einen sogenannten *Plug-in*-Mechanismus. Die Kriterien, die ein CASE-Tool erfüllen muss, damit es für Stratifikation nutzbar ist, werden zunächst festgelegt.

[7] http://www.eclipse.org/uml2

Zum einen muss eine programmatische Schnittstelle zu den Elementen eines Diagramms bestehen, so dass eine Modelltransformation automatisiert durchgeführt werden kann. Desweiteren ist es wünschenswert, die Möglichkeit zu haben ein eigenes Metamodell für Diagramme zu definieren und in das CASE-Tool einzugliedern, um neue graphische Elemente einführen zu können. Das Tool muss weiterhin in der Lage sein aus einem Modell eine Implementierung für eine Programmiersprache zu generieren (*Forward Engineering*). Idealerweise sollte es möglich sein, Programmcode wieder zurück in ein Modell zu wandeln (*Reverse Engineering*). Letzteres ist für die Zwecke der Stratifizierung jedoch nicht notwendig. Weitere Kriterien sind die Austauschbarkeit der Modelle mit anderen Modellierumgebungen und die Programmiersprache, in der man ein Plug-in für das CASE-Tool schreiben kann. Diese soll Java sein. Java ist frei verfügbar, plattformunabhängig und es existieren eine Reihe guter Entwicklungsumgebungen. Java bietet weiterhin den Vorteil der Reflektion, d.h. es ist möglich Informationen über Klassen während der Laufzeit abzufragen, was von großem Nutzen ist, wie sich in Kapitel 5 zeigen wird.

Einen guten Überblick über CASE-Tools[8] geben [12] und [16]. Die CASE-Tools, die in die nähere Wahl kommen sind: ArgoUML, EclipseUML, Fujaba und Together die im Folgenden kurz betrachtet werden.

3.3.1 ArgoUML

ArgoUML[9] ist ein freies UML Modellierungswerkzeug, das von der University of California entwickelt wurde und als Open Source Projekt vertrieben wird, an dem jeder mitentwicklen kann. ArgoUML setzt Ideen der Kognitiven Psychologie um, die die Modellierung teilweise vereinfachen und dem Modellierer während der Arbeit kontextbezogene Hilfestellung bieten. Es basiert auf offenen Standards wie XMI, SVG und PGML. Für den Plug-in-Entwickler bietet ArgoUML eine gute Dokumentation und Anleitung.

3.3.2 EclipseUML

EclipseUML[10] von der Firma Omondo ist ein Werkzeug zum Modellieren von UML Diagrammen. Es ist als Plug-in für Eclipse realisiert und nach eigenen Angaben das führende UML Plug-in für Eclipse. Es ist für akademische Zwecke und nichtkommerzielle Einrichtungen kostenfrei verfügbar. Die Modellierung mit EclipseUML funktioniert recht gut, jedoch war es zum Zeitpunkt der Erstellung dieser Arbeit nicht möglich, programmatisch auf die Funktionen von EclipseUML zuzugreifen. Die angekündigte OpenAPI war auch nach mehrfacher Rücksprache nicht

[8] es existieren bereits über 100 solcher Progamme, die teils kommerziell, privat oder universitär entwickelt werden

[9] http://argouml.tigris.org/

[10] http://www.eclipseuml.com/

zugänglich.

3.3.3 Fujaba

Fujaba [20] ist ein freies CASE-Tool, das an der Universität Paderborn entwickelt wird und an dessen Entwicklung sich zehn weitere Universitäten beteiligen (unter anderem Aachen, Antwerpen, Darmstadt, Kassel, Leiden und Siegen). Es ist ebenso wie ArgoUML ein Open Source Projekt und bietet einen guten Plug-in-Mechanismus, über den man uneingeschränkten Zugriff auf Diagramme und deren Elemente erlangt. Neue Diagramme und Elemente können hinzugefügt und über einen Adaptionsmechanismus mit fremden Diagrammelementen verknüpft werden. Einen großen Vorteil bietet Fujaba bei der Implementierung von Methoden einer UML Klasse. Mit einem Fujaba-eigenen Diagrammtyp kann eine UML Methode durch Kombination von graphischen und textuellen Elementen implementiert werden. Ein Codegenerator erzeugt aus diesen Diagrammen und UML Klassendiagrammen eine lauffähige Implementierung des modellierten Systems in einer bestimmten Programmiersprache, wobei Fujaba um Codegeneratoren für verschiedene Programmiersprachen erweiterbar ist. Standardmäßig wird Java-Code erzeugt. Modelle werden in einem propriäteren Format gespeichert, man hat jedoch durch ein Plug-in die Möglichkeit Modelle im XMI-Format zu exportieren. Fujaba stellt desweiteren eine Technik Namens *Story Driven Modeling* zur Verfügung, auf die in 4.3 genauer eingegangen wird.

3.3.4 Together

Borland bietet mit *Together*[11] ein Produkt, dass durch seinen Funktionsumfang überzeugen kann. Together bietet Plug-in-Entwicklern mit seiner OpenAPI [6] eine Programmierschnittstelle, über die man Zugriff auf Diagramme und deren Elemente erlangen kann. Die OpenAPI ist sehr gut dokumentiert und neue Diagrammelemente und Diagramme können dem System hinzugefügt werden. Der einzige Nachteil an Together ist die Tatsache, dass es ein kommerzielles Produkt ist und somit nicht jedermann zur freien Verfügung steht. Für akademische Zwecke kann eine vergünstigte Lizenz beantragt werden.

Die Entscheidung, auf welchem CASE-Tool SPin aufsetzen soll, fiel auf Fujaba. Es ist frei verfügbar und seine Quellen liegen offen. Durch die integrierte Technik des Story Driven Modelling ist es zudem einfach Mustererkennungsprozesse zu implementieren, die die umzusetzende Modelltransformation sehr erleichtern.

[11] http://www.borland.com/together/

Kapitel 4

Fujaba

In diesem Kapitel wird das CASE-Tool Fujaba näher betrachtet. Es wird zunächst ein allgemeiner Überblick über Fujaba gegeben, anschließend wird auf Strukturen eingegangen, auf denen SPin aufbaut und auf Besonderheiten von Fujaba, die von SPin zur Realisierung der Stratifikation genutzt werden.

4.1 Allgemeines

Die Entwicklung von Fujaba wurde 1997 von der Software Engineering Gruppe der Universität Paderborn gestartet. Im Jahr 2002 wurde Fujaba einem Redesign unterzogen und von einem monolithischen System zu einer Plug-in Architektur umgebaut. Das Hauptziel von Fujaba ist es, eine einfach zu erweiternde UML und Java-Entwicklungsplattform zur Verfügung zu stellen. Fujaba unterstützt das Generieren von Sourcecode aus UML Modellen. Dabei ist die Sprache des generierten Sourcecodes standardmäßig Java, aber nicht auf diese beschränkt, da Fujaba weitere Sourcecode-Generatoren hinzugefügt werden können. Die umgekehrte Richtung, also das Erzeugen von UML Modellen aus Sourceode, ist (durch ein entsprechendes Plug-in) ebenfalls möglich. Das wohl interessanteste Feature von Fujaba ist seine Fähigkeit, Modelle mit einer Mischung aus graphischen und textuellen Elementen so zu programmieren, dass der Codegenerator daraus einen lauffähigen Prototypen generieren kann, der kompilier- und ausführbar ist. Diese Fähigkeit von Fujaba wird in 4.3 näher betrachtet.

4.2 Modellierungs Framework

Das Modellierungs Framework von Fujaba verwendet auf oberster Ebene die Implementierung eines *Abstrakten Syntax Graphen* (*ASG*). Abbildung 4.1 stellt ein stark vereinfachtes Modell dieses ASG dar. Die drei wesentlichen Elemente des

ASG sind *ASGElement*, *ASGElementRef* und *ASGDiagram*. Diese drei Elemente beschreiben, wie ein ASG aufgebaut ist:

- ein Diagramm kann kein oder mehrere Elemente enthalten
- ein Diagramm ist dabei selbst ein Element
- ein Element kann in einem oder mehreren Diagrammen enthalten sein
- ein Element kann von anderen Elementen referenziert werden

Ein ASG besteht somit aus einem Element, das weitere Elemente enthalten kann, sofern es selbst ein Diagramm ist.

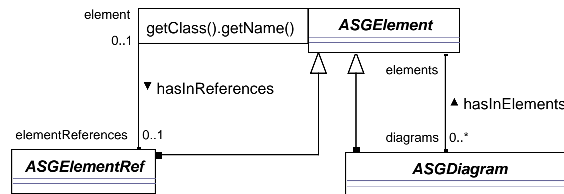


Abbildung 4.1: *Das Fujaba ASG Metamodell: der* Abstract Syntax Graph *(ASG)*

Der Referenzierungs-Mechanismus erlaubt, dass zwei voneinander unabhängige Metamodelle zusammenarbeiten können. Entwickelt ein Plug-in Entwickler A ein eigenes Metamodell M_A, dessen Elemente ein Metamodell M_B eines anderen Entwicklers B referenzieren sollen, Entwickler B aber M_A nicht kennt, so schreibt Entwickler A eine Adapter-Klasse (eine Klasse, die von *ASGElementRef* erbt) für jedes Element $e_A \in M_A$, welches ein Element $e_B \in M_B$ referenzieren soll und kann so eine Referenz von e_A auf e_B realisieren. Zu beachten ist, dass es immer nur einen Adapter für eine Instanz von e_B geben darf. Ein Adapter muss deswegen so implementiert werden, dass er eine Menge von Instanzen der Elemente aus A verwalten kann.

Mit der Adaptertechnologie ist es einem Element möglich, alle Elemente über Zustandsänderungen zu informieren, von denen es referenziert wird. Die referenzierenden Elemente können dann entsprechend auf diese Änderungen reagieren. Als Beispiel für den Referenzierungs-Mechanismus sei auf den Adaptionsmechanismus des Annotierungs-Metamodells von SPin verwiesen, der in Abbildung 5.2 auf Seite 23 dargestellt ist.

Fujaba implementiert einen Teil des UML Metamodells der OMG. Dieser Teil realisiert Klassen-, Objekt-, Paket-, Usecase- und (modifizierte) Aktivitätsdiagramme. Der Teil, der Klassendiagramme realisiert, ist in Abbildung 4.2 vereinfacht dargestellt.

Abbildung 4.2: *vereinfachter, nicht vollständiger Teil des Fujaba UML Metamodells für Klassendiagramme*

Man sieht deutlich die Anbindung des UML Metamodells für Klassendiagramme an das ASG Metamodell. Die Anbindung ist über Vererbung realisiert, deswegen erbt die Basisklasse für UML Elemente (*UMLIncrement*) von *ASGElement* und *UMLClassDiagramm* von *ASGDiagram*. Alle Elemente eines UML Klassendiagramms erben von *UMLDiagramItem*. Zugriff auf sie erhält man über das Diagramm. Attribute und Methoden bilden jedoch eine Ausnahme. Sie sind Teil einer Klasse und tauchen deswegen nicht direkt im Diagramm als Elemente auf. Man greift auf sie über die Klasse zu.

4.3 Story Driven Modeling (SDM)

Wie bereits mehrfach angesprochen, kann mit Fujaba aus einem UML Modell eine komplette, lauffähige Implementierung generiert werden. Möglich wird dies, indem neben dem Festlegen der Struktur eines Modells (Klassendiagramme) auch die Erstellung der Implementierung von Methoden einer Klasse möglich ist.

Die Implementierung einer Methode wird mit leicht modifizierten Aktivitätsdiagrammen realisiert, sogenannten *Storydiagramm*en [10]. Dazu wird jeder Methode ein Storydiagramm zugewiesen, aus dem ein Codegenerator die gewünschte Implementierung erzeugt. Abbildung 4.3 zeigt den Teil des Metamodells von Fujaba,

der Storydiagramme definiert.

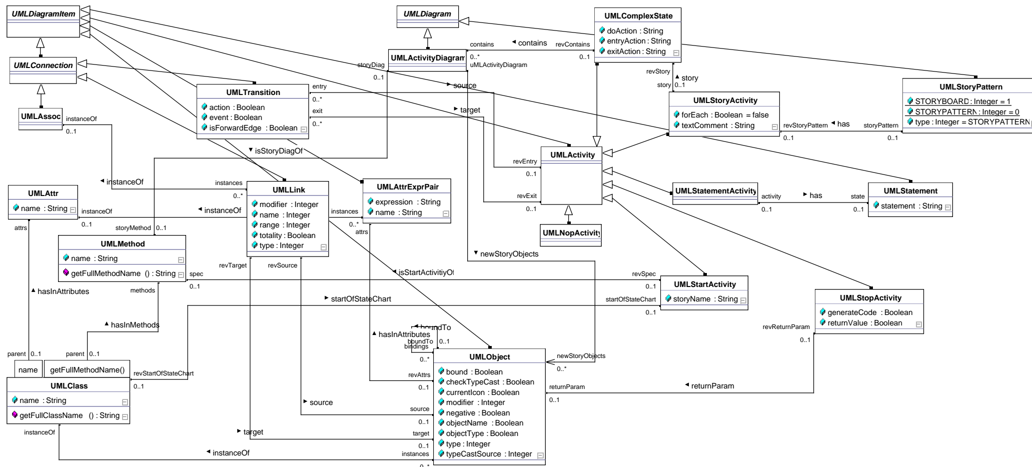


Abbildung 4.3: *vereinfachter, nicht vollständiger Teil des Fujaba UML Metamodells für Storydiagramme*

Der Aufbau eines Storydiagramms (*UMLActivityDiagramm*) besteht aus einem Startpunkt (*UMLStartActivity*), optional weiteren Aktivitäten (Klassen die von *UMLActivity* erben) und einem oder mehreren Endpunkten (*UMLStopActivity*). Aktivitäten werden mit *UMLTransition*s verbunden, die die Ausführungsreihenfolge der Aktivitäten festlegen. Die erwähnten optionalen Aktivitäten können sowohl Codefragmente (*UMLStatementActivity*) als auch Elemente enthalten, die zur graphischen Programmierung verwendet werden können (*UMLStoryActivity*). Eine *UMLStoryActivity* enthält ein *UMLStoryPattern*, das den Kern des *Story Driven Modeling* (*SDM*) bildet.

Um ein Story-Pattern erstellen zu können, benötigt Fujaba ein UML Klassendiagramm, aus dem strukturelle Informationen extrahiert werden. Dies muss im aktuellen Projekt vorhanden sein. Anhand der Informationen kann ein Story-Pattern spezifiziert werden. Die Notation eines Story-Pattern ähnelt der eines UML Kollaborationsdiagramms (entspricht dem Kommunikationsdiagramm in der UML2.0-Spezifikation) und enthält folgende Elemente: Objekte (*UMLObject*), Links zwischen den Objekten (*UMLLink*) und Attribut-Wert-Paare (*UMLAttrExprPair*), wobei:

- · Objekte Instanzen von Klassen,
- · Links Instanzen von Assoziationen und
- · Attribut-Wert-Paare Instanzen von Attributen sind.

Durch diese Kombination von Aktivitäts- und Kollaborationsdiagramm wird es mit einem Storydiagramm möglich, Schleifen und *Wenn-Dann*-Konstrukte graphisch zu beschreiben und Mustererkennungsprozesse und Verwaltungsaufgaben auf Instanzen von Klassen durchzuführen. Die Verwaltungsaufgaben bestehen im Erzeugen und Löschen von Objekten, Wertänderung von Attributen und Setzen oder Aufheben von Beziehungen zwischen Objekten. Bezogen auf das ASG Metamodell und die Elemente *ASGElement* und *ASGDiagram* (siehe Abbildung 4.1) ist darunter zu verstehen:

- Elemente in ein Diagramm einfügen,
- Elemente aus einem Diagramm entfernen,
- Elemente erzeugen und
- Elemente löschen.

Abbildung 4.4 zeigt die mit einem Storydiagramm implementierte Methode 'fahre():Boolean'. Diese Methode gehört zur Klasse *Fahrzeug*, welche die Attribute *raeder:Integer* und *faehrt:Boolean* enthält. Im Storydiagramm wird geprüft, ob das Fahrzeug mindestens ein Rad enthält; wenn ja wird es in einen fahrenden Zustand versetzt, wenn nicht wird abgebrochen.

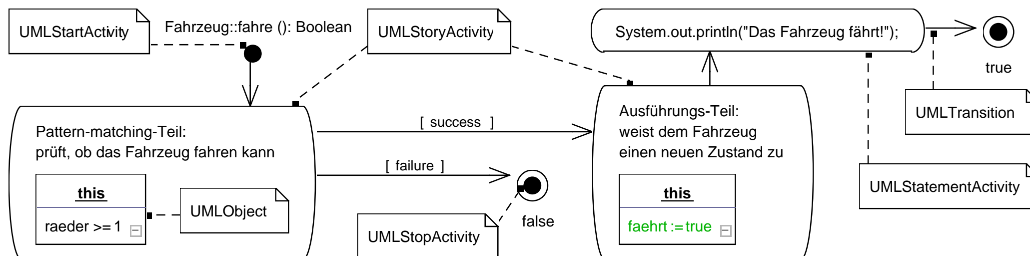


Abbildung 4.4: *Implementierung der Methode 'fahre():Boolean' mit einem Storydiagramm*

Weitere Beispiele für Storydiagramme können [10] und dem Anhang entnommen werden.

4.4 Modelltransformation mit SDM

Um zu verstehen, wie die beschriebenen Story-Patterns funktionieren und wie sie mit der Modelltransformation zusammenhängen, wird zunächst darauf eingegangen, auf welcher Theorie Story-Patterns basieren. Anschließend wird der Ablauf einer Modelltransformation erklärt (siehe Abbildung 4.5).

Modelltransformationssprachen besitzen als Grundlage eine *Graph Grammatik*. Story-Patterns basieren auf der *TGG* (Triple Graph Grammar), die erstmals 1994 von Schürr in [18] vorgestellt wurde. Eine Transformation wird anhand von Transformationsregeln durchgeführt, die Instanzen der Grammatik sind. Die TGG ist also ein Metamodell für Transformationsregeln.

Eine Regel besteht aus einer *Left-Hand-Side* (*LHS*), die das Muster festlegt, nach dem im Modell gesucht werden soll, einer *Right-Hand-Side* (*RHS*), die angibt was aus dem Muster werden soll und einem *correspondence graph*, der die Beziehung der Elemente aus der LHS und der RHS angibt. Er sagt aus, wie die Elemente der LHS geändert werden müssen, damit die RHS entsteht.

Allgemein gesprochen wird bei der Modelltransformation ein Modell in ein anderes Modell transformiert. Das Metamodell des Ausgangsmodells muss dabei nicht zwingend mit dem des Zielmodells übereinstimmen. Im Ausgangsmodell wird nach dem Vorkommen eines Musters gesucht. Wird dieses Muster gefunden, wird es anhand einer festgelegten Regel transformiert. Das so entstehende Modell ist das Zielmodell.

Story-Patterns sind eine Umsetzung von Transformationsregeln, die auf der TGG basieren. LHS und RHS werden in einem Bild dargestellt nämlich in einem Story-Pattern bzw. einem Storydiagramm. Elemente des Bildes sind die UML Elemente, die in 4.3 angesprochen wurden.

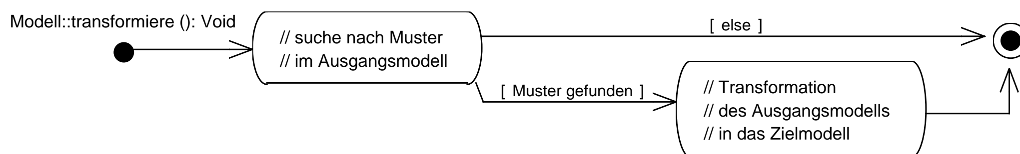


Abbildung 4.5: *schematischer Ablauf einer Modelltransformation*

Kapitel 5

SPin

Dieses Kapitel beschreibt das im Rahmen dieser Arbeit zur Realisierung von Stratifikation entwickelte Werkzeug SPin. Zu Anfang wird auf die Anforderungen eingegangen, die an SPin gestellt werden, nachfolgend wird in 5.2 die Architektur von SPin vorgestellt und in 5.3 die Benutzungsschnittstelle erklärt. Abschließend wird in 5.4 auf einige Details der Implementierung von SPin eingegangen.

5.1 Anforderungen

Die Hauptanforderungen an ein Werkzeug, das Stratifikation realisieren soll, wurden in [3] postuliert. SPin muss demnach

- eine Annotationssprache und
- einen Satz von Transformationen

zur Verfügung stellen. Die Annotationssprache soll Markierungen bereitstellen, die in ein Modell eingefügt werden können und mit Transformationen verknüpfbar sind. Diese Markierungen werden durch die in Abschnitt 5.2.1 eingeführten *RefinementAnnotation*s realisiert. Das Transformationssystem muss in der Lage sein, Regeln auf einem Modell anzuwenden, die dieses in einen anderen Zustand transformieren. Genauer gesagt soll ein festgelegtes Muster im Modell erkannt und in ein anderes Muster transformiert werden. Zwischen den Transformationen sollen Relationen definierbar sein.

Ausdruckstarke Modelle und Muster, die nützliche und wiederverwendbare Entwurfsabstraktionen beschreiben, sollen mit Hilfe von SPin in kurzer Zeit erstellt werden können. Zudem soll es einfach sein, Modelle zu annotieren und Regeln zu erstellen. Die Verfeinerung von Interaktionen (interaction refinement) soll möglich sein. Die Navigation zwischen den Ebenen soll das Modell transformieren.

Im Laufe der Transformation wird kein neues Modell erzeugt, sondern das alte modifiziert.

SPin unterscheidet zwei Benutzergruppen, die unterschiedliche Rollen inne haben. Die erste Gruppe besteht aus *Regelentwicklern*, die zweite aus *Regelanwendern*, wobei in der Praxis Regelentwickler und Regelanwender durchaus eine Person sein können.[1] Die Aufgabe eines Regelentwicklers besteht darin Regeln zur Verfügung zu stellen, die von Regelanwendern genutzt werden können. Jeder, der die Programmiersprache Java beherrscht und sich mit dem Modell und den Komponenten vertraut macht, die er mit Hilfe einer Regel transformieren bzw. modifizieren möchte, kann die Rolle des Regelentwicklers übernehmen. Regelanwender und Regelentwickler sollen so weit wie möglich von SPin unterstützt werden.

5.2 Architektur

SPin ist in vier Teile gegliedert. Jeder erfüllt eine bestimmte Aufgabe und ist für eine Benutzergruppe vorgesehen. Zudem stellt jeder Teil eine API bereit, die von Entwicklern genutzt werden kann, um auf die Funktionalität von SPin programmatisch zuzugreifen. Es folgt eine Kurzbeschreibung der einzelnen Teile. Eine ausführlichere Beschreibung kann den darauf folgenden Abschnitten entnommen werden.

Annotation *(Regelanwender)* Einem Diagramm können Annotierungen hinzugefügt werden, die einen abstrakten Sachverhalt beschreiben.

Transformationsregeln *(Regelentwickler)* Ermöglichen es abstrakte Syntaxgraphen zu transformieren. So können z.B. UML Diagramme mit Hilfe einer Regel transformiert und Annotierungen konkretisiert werden.

Transformation *(Regelanwender)* Das Transformationssystem wendet Regeln an.

Hilfskonstrukte *(Regelentwickler)* Fabriken, Textmodifizierer und ein Synchronisations-Modul helfen bei der Automatisierung von Vorgängen, wie z.B. der Erstellung und Implementierung von Transformationsregeln.

5.2.1 Annotation

Um Stratifikation zu realisieren, führen wir ein neues Metamodell ein: das *Annotierungs-Metamodell*. Die Elemente des Metamodells sollen in jegliche Art von Diagrammen eingebettet werden und mit Elementen des Diagramms in Beziehung stehen können. Desweiteren sollen die Elemente parametrisierbar sein.

[1]insbesondere sind Regelentwickler immer auch Regelanwender, denn sie müssen ihre entwickelten Regeln testen

Das Annotierungs-Metamodell stellt im wesentlichen drei neue Elemente zur Verfügung. Dies sind die *RefinementAnnotation* (Klasse *RAnnotation*), der *RefinementLink* (Klasse *RLink*) und der *RefinementParameter* (Klasse *RParameter*). Die erstgenannten haben als gemeinsame Oberklasse die Klasse *RElement*, die als Grundfunktionalität Sichtbarkeitszustand und Parametrisierung zur Verfügung stellt. Bild 5.1 zeigt ein UML Klassendiagramm, das die wesentlichen Elemente des Annnotierungs-Metamodells enthält. Bild 5.2 zeigt das vollständige Annotierungs-Metamodell, in dem zusätzlich der Adaptionsmechanismus an das ASG Metamodell berücksichtigt ist.

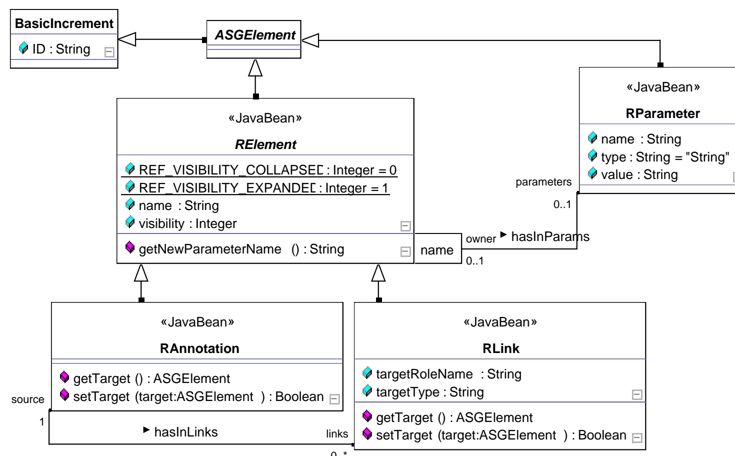


Abbildung 5.1: *Teil des Annotierungs-Metamodells, der die wesentlichen Elemente enthält.*

RefinementAnnotation, RefinementLink und RefinementParameter werden im Folgenden, sofern eindeutig, mit Annotation, Link und Parameter bezeichnet.

Visualisierung der Elemente des Annotierungs-Metamodells

Alle Elemente des Annotierungs-Metamodells sollen in einem Diagramm visualisiert werden können. Damit dies möglich ist, müssen diese von *ASGElement* erben. Wie in 4.2 beschrieben wurde, ist dies eine von Fujaba bereitgestellte Klasse, die die Basis für alle Elemente bildet, die in Diagrammen enthalten sein können. Desweiteren müssen Datenänderungen eines Elements seiner graphischen Repräsentation mitgeteilt werden. Das strukturelle Modell von SPin wurde in Fujaba modelliert und aus diesem Modell eine Basis-Implementierung für Java generiert. Der Fujaba Java-Codegenerator ist in der Lage generierten Klassen einen Mechanismus hinzuzufügen, der interessierten Instanzen Änderungen der Datenstruktur mitteilt. Dazu müssen die entsprechenden UML Klassen mit dem Stereotypen

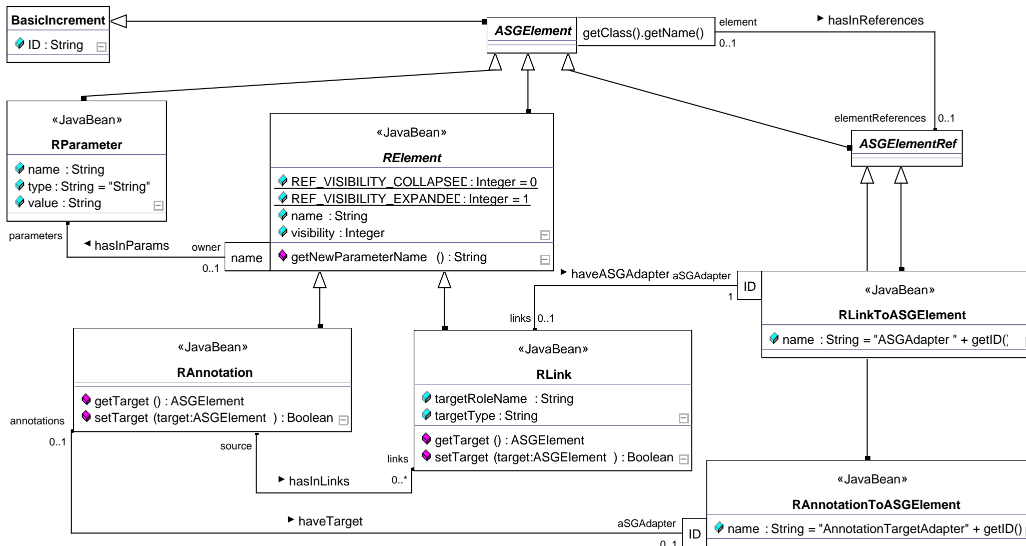


Abbildung 5.2: *Komplettes Annotierungs-Metamodell inklusive Adaptionsmechanismus zum ASG Metamodell.*

<<JavaBean>> annotiert werden. Aus diesem Grund besitzen alle Elemente des Annotierungs-Metamodells, die eine graphische Repräsentation haben sollen, den Stereotypen <<JavaBean>>.

RefinementAnnotation

Jede Annotation trägt einen Namen, der kenntlich macht, welcher Transformationsregel sie zugeordnet ist. Annotationen können einem Diagramm hinzugefügt und optional über einen Adapter (Klasse *RAnnotationToASGElement*) an ein Diagramm-Element gebunden werden[2]. Eine Annotation stellt einen komplexen Sachverhalt abstrahiert dar, indem Details verborgen werden. Wie der Sachverhalt im einzelnen aussieht, ist in der zugehörigen Transformationsregel kodiert. Die Annotation ist also eine Markierung in einem Diagramm, die angibt, dass etwas transformiert werden soll. Für Stratifikation bedeutet dies, dass durch sie ein abstrakter Sachverhalt beschrieben wird, der in einen konkreteren aufgelöst werden soll.

Eine Annotation kann sowohl expandiert als auch kollabiert dargestellt werden. Im expandierten Zustand werden sämtliche Informationen, die die Annotation beeinhaltet visualisiert und seine Links angezeigt. Im kollabierten Zustand werden diese Informationen und die Links nicht angezeigt und die Annotation wird, sofern

[2]z.B. an eine UML Klasse, Methode, Assoziation oder auch an das Diagramm

sie an ein Element gebunden ist, in der Nähe dieses Elements platziert.

RefinementParameter

Parameter sind dafür vorgesehen, Elemente des Annotierungs-Metamodells mit Standarddatentypen wie Zeichenketten, Ganzen Zahlen und Gleitkommazahlen zu parametrisieren.

RefinementLink

Links werden dazu verwendet eine Annotation mit Elementen eines abstrakten Syntaxgraphen (*ASGElement*) zu parametrisieren. Jeder Link ist genau einer Annotation zugeordnet und über einen Adapter (*RLinkToASGElement*) an ein *ASGElement* gekoppelt. Der Link weist dem gekoppelten *ASGElement* einen Rollennamen zu, über den dieses im späteren Mustererkennungsteil einer Transformationsregel (siehe 5.2.2) identifiziert werden kann.

Ein RefinementLink ist prinzipiell vergleichbar mit einem UML Link, der zwei Klassen verbindet. Jedoch ist der RefinementLink dafür ausgelegt, ein Element vom Typ *RAnnotation* mit einem *ASGElement* zu verbinden, d.h. mit einem Diagramm-Element beliebigen Typs.

Beispiel *interaction refinement*

Mit Hilfe von Stratifikation kann die Semantik einer Interaktion (eine Interaktion ist z.B. eine Assoziation zwischen zwei Klassen) beeinflusst werden. Dazu wird dieser eine Annotation zugewiesen, die mit Parametern und Links versehen wird. Links werden eingesetzt um Elemente zu kennzeichnen, die in den Verfeinerungsprozess einbezogen werden sollen (Stichwort *hot spots*). Dies soll an einem einfachen Verschlüsselungsbeispiel verdeutlicht werden. Abbildung 5.3 zeigt, wie man eine Assoziation zwischen zwei Klassen annotieren kann. Der Annotation „Encrypted“ ist als Ziel die Assoziation „talksTo“ zugewiesen. Dem Verfeinerungsprozess für „Encrypted“ können zusätzliche UML Klassen, die eine Verschlüsselungs-Strategie implementieren, übergeben werden. Diese können mit Hilfe von Links, die den Rollennamen „encryptionStrategy“ tragen, kenntlich gemacht werden.

Warum ein eigenes Metamodell?

Es soll kurz erläutert werden, warum ein neues Metamodell eingeführt und nicht z.B. das UML Metamodell verwendet wird, um Annotierungen in ein Diagramm einzufügen. Anstelle einer Annotation hätte man Klassen und Stereotypen, anstelle der Links Assoziationen und anstelle der eigenen Parameter UML Parameter

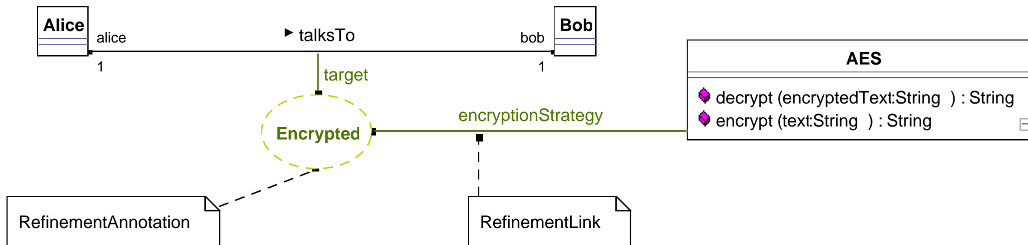


Abbildung 5.3: *Beispiel für die Annotierung einer* interaction refinement

verwenden können. Gründe, warum dies nicht gemacht wurde, sind:

- Annotierungen des gleichen Namens sollen in einem Diagramm mehrfach vorhanden sein können → eine UML Klasse muss jedoch einen eindeutigen Namen tragen
- Annotierungen sollen in jeder Art von Diagramm eingesetzt werden können, nicht nur in UML Klassendiagrammen
- Annotierungen sollen von anderen Elementen visuell unterscheidbar sein und deswegen unter anderem eine andere Form als eine Klasse haben
- für einfache Annotierungszwecke ist das UML Metamodell zu komplex
- Assozationen sind nicht mächtig genug, da sie nur Klassen verbinden können und nicht beliebige Elemente eines Diagramms

Desweiteren besteht durch das Einführen eines neuen Metamodells die Möglichkeit, dieses an die Zwecke der Transformation anzupassen. Es kann außerdem für die Mustererkennung optimiert werden. So wird es leichter, den Mustererkennungsteil einer Transformationsregel zu implementieren, in dem eine Annotation mit bestimmter Struktur erkannt werden soll, da die Strukturen in wenigen Objekten gebündelt sind. Eine Regel, die in ihrem Mustererkennungsteil das Vorhandensein dieser Strukturen überprüft, ist die *RefinementRule*, die im nächsten Abschnitt vorgestellt wird.

5.2.2 Transformationsregeln

Transformationsregeln sind als Java-Klassen realisiert. Sie haben festgelegte Einsprungsfunktionen, die von einem Transformator (siehe 5.2.3) aufgerufen werden. Regeln können mit einem Texteditor, einer Java-Entwicklungsumgebung oder einem CASE-Tool erstellt und bearbeitet werden. In letzterem Fall wird eine Regel als UML Klasse modelliert.

Was eine Regel macht, hängt von ihrer Implementierung ab. Vorgesehen ist, dass diese eine Transformation auf einem Modell durchführt. Dies kann eine *Modell-Modell-*, *Modell-Code-*, *Code-Modell* oder *Code-Code-Transformation* sein. Es sei erwähnt, dass bei falscher Implementierung einer Regel das Modell unbrauchbar werden kann. Außerdem ist es möglich, andere Vorgänge in einer Regel durchzuführen als eine Transformation auf Modellen. Dem Regelentwickler stehen Java- und Fujaba-API zur Verfügung, die er in vollem Umfang einsetzen kann. Deswegen sollte darauf geachtet werden, dass nur Regeln, die aus einer vertrauenswürdigen Quelle stammen und deren Quellcode offen liegt, angewendet werden.

Eine Regel muss von der Basisklasse für alle Transformationsregeln, *TransformationRule*, erben. Weitere Basisklassen für Regeln sind *AbstractionRule* und *RefinementRule*, auf die später eingegangen wird. Jeder von SPin unterstützte Regeltyp hat eine bestimmte Kennung. Diese wird als Präfix dem Klassennamen einer Regel hinzugefügt. Die Zuordnung von Regeltyp zu Präfix kann Tabelle 5.1 entnommen werden. Der Klassenname einer Regel legt ihren Namen fest, das Präfix ist jedoch nicht Teil des Namens. Die Konvention für Klassennamen von Regeln sieht in der Backus-Naur-Form (BNF) wie folgt aus:

<Klassenname einer Regel> ::= <Präfix><Java-Klassenname>
<Präfix> ::= TR, AR, RR

Der Klassenname einer Regel vom Typ *RefinementRule* mit Namen *MeineRegel* würde also *RRMeineRegel* lauten.

Regeltyp	Präfix
TransformationRule	TR
AbstractionRule	AR
RefinementRule	RR

Tabelle 5.1: *Zuordnung von Präfix zu Transformationsregel*

SPin bietet einen automatisierten Export-Mechanismus für Regeln an. Dieser generiert aus dem Modell der Regel eine Java-Datei, die anschließend mit dem Java-Compiler kompiliert wird. Diese beiden Dateien und weitere Metadaten über die Regel werden in einem JAR-Archiv abgelegt, welches im *Bibliotheks-Verzeichnis* gespeichert wird. Nach Abschluß dieses Vorgangs wird die Regel automatisch der *Regel-Bibliothek* (siehe unten)hinzugefügt und steht somit zum Einsatz bereit.

Aufbau von Transformationsregeln

Die Struktur von Transformationsregeln ist in Abbildung 5.4 dargestellt. Man sieht dort die Basis aller Transformationsregeln, *TransformationRule*. Sie besitzt zwei Einsprungsfunktionen, die von einem Transformator aufgerufen werden können.

Dies sind 'apply(ASGElement):Boolean' und 'isApplicable(ASGElement):Boolean'. Desweiteren besitzt *TransformationRule* ein *RuleInfo*-Objekt, welches dafür vorgesehen ist eine Regel detailliert zu beschreiben.

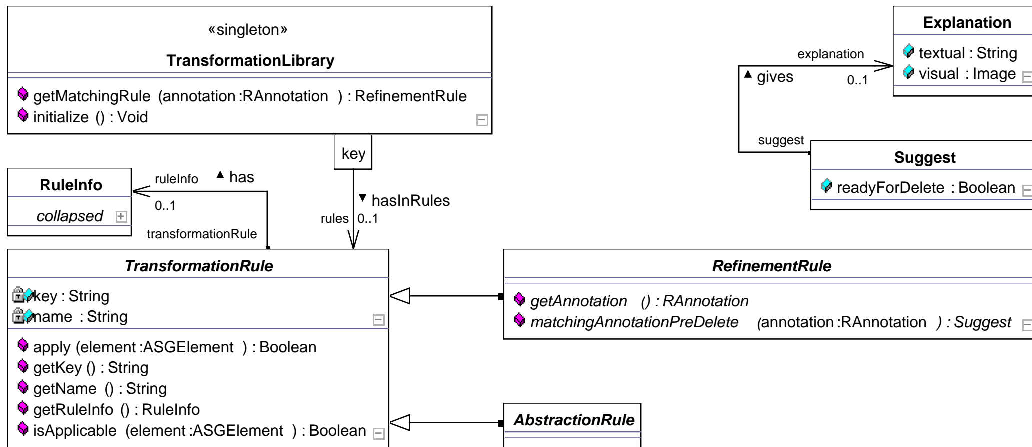


Abbildung 5.4: *Transformationsregeln und deren Verwaltung*

Die Implementierungs-Grundstruktur der Methode 'apply(ASGElement):Boolean' folgt dem Schema, das in Abbildung 5.5 dargestellt ist. Sie hat einen Mustererkennungs- und einen Ausführungsteil. Im Mustererkennungsteil wird geprüft, ob das übergebene Element transformiert werden kann. Sofern dies möglich ist, wird im Transformationsteil das Element, bzw. Muster transformiert, in dem das Element enthalten ist. Der Rückgabewert gibt an, ob das Element bzw. Muster modifiziert wurde und wird im Transformationssystem benötigt (siehe 5.2.3).

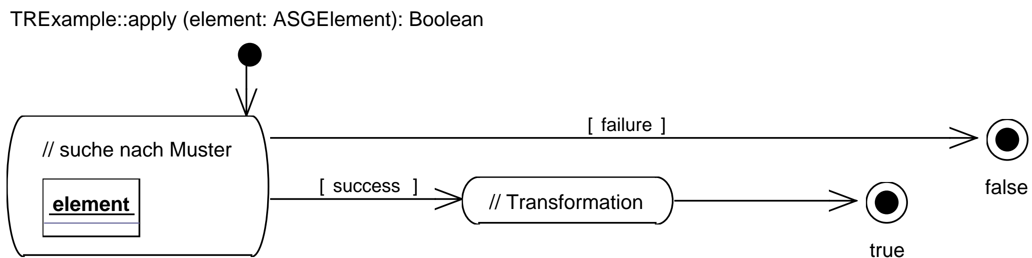


Abbildung 5.5: *Grundstruktur einer Regel vom Typ TransformationRule*

Die Methode 'isApplicable(ASGElement):Boolean' prüft, ob die Regel auf dem angegebenen Element anwendbar ist. Diese Methode ist von SPin generierbar und enthält eine Kopie des Mustererkennungsteils der Methode 'apply(ASGElement):

Boolean'.

Abbildung 5.6 zeigt die Informationsstruktur einer Regel. Ein *RuleInfo*-Objekt gibt die Version der Regel, eine Beschreibung der Regel und Informationen über den Regelentwickler an. Desweiteren gibt das *RuleInfo*-Objekt Auskunft darüber, für welche CASE-Tools die Regel einsetzbar ist[3]. Eine Beschreibung kann textuell oder graphisch sein und soll eine Erklärung dessen enthalten, was die Regel macht und welche Voraussetzungen zu erfüllen sind, damit das Ergebnis der Regel nutzbar wird. Im Falle einer Regel, die Code für die Programmiersprache Java 1.5 erzeugt, würde das Attribut *requires:StringArray* beispielsweise den String *Java 1.5* enthalten. Über das *RuleInfo*-Objekt können ferner die lokale JAR-Datei, die Quelle der JAR-Datei und der Name des Pakets, in dem sich die Regel befindet, erfragt werden. Zusätzlich können Voraussetzungen spezifiziert werden, die erfüllt sein müssen, damit die Regel angewendet werden kann. Nutzt eine Regel beispielsweise ein bestimmtes Fujaba Plug-in oder Klassen einer eigenen Bibliothek, so würde das Attribut *ruleRequires:StringArray* einen Eintrag haben, der den Namen der Plug-in- bzw. Bibliotheks-JAR-Datei enthält.

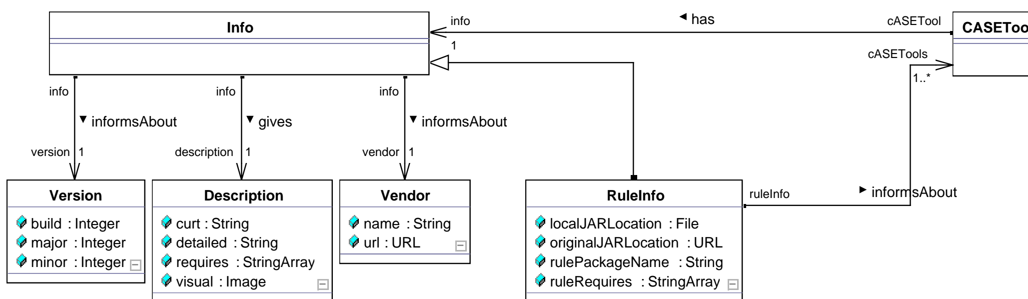


Abbildung 5.6: *Informationsstruktur einer TransformationRule*

Für die Zwecke der Stratifikation werden zwei Regeltypen benötigt, die als Basis-Regeltypen in SPin integriert wurden. Dies sind *Abstraktionsregeln* (*AbstractionRule*) und *Verfeinerungsregeln* (*RefinementRule*), die invers zueinander sind. Beide zusammen bilden die in Kapitel 2 angesprochenen Relationen. Damit Navigation zwischen Strata möglich ist, werden Informationen benötigt, die angeben, wie man wieder zum Original zurückkehren kann. In Fujaba existiert ein *Undo*-Mechanismus mit dem dies möglich ist. Desweiteren kann der aktuelle Stand vor der Anwendung einer Regel in einem anderen Projekt gespeichert werden.

[3]Dieser Mechanismus wurde vorsorglich eingebaut, falls SPin so erweitert wird, dass es in mehreren CASE-Tools einsetzbar ist.

Abstraktionsregeln

Eine Abstraktionsregel besitzt neben der geerbten Struktur aus der Basisklasse keine zusätzlichen Strukturen. Der Unterschied zur Basisklasse besteht darin, dass die Implementierungs-Grundstruktur der Methode 'apply(ASGElement):Boolean' im Mustererkennungsteil zusätzlich prüft, ob das Element in einem Diagramm enthalten ist (siehe Abbildung 5.7). Dies ist ein typisches Muster für den Abstraktionsprozess, da meist überprüft werden muss, ob eine Menge bestimmter Elemente in einem Diagramm enthalten ist. Sofern das Diagramm-Objekt im Mustererkennungsteil nicht benötigt wird, kann es daraus entfernt werden.

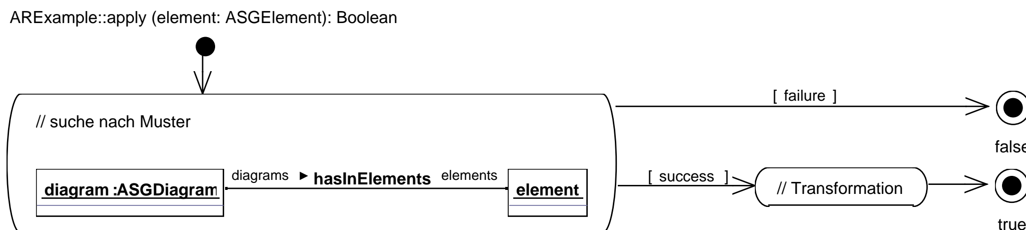


Abbildung 5.7: *Grundstruktur einer Regel vom Typ AbstractionRule*

Verfeinerungsregeln

Verfeinerungsregeln besitzen zwei zusätzliche Methoden. Dies ist zum einen die Methode 'getAnnotation():RAnnotation', welche die benötigte Struktur der Annotation zurückgibt, die im Mustererkennungsteil der Methode 'apply(ASGElement):Boolean' benötigt wird. Jene Methode ist generierbar und kommt im Editor für Annotationen zum Einsatz. Sie ermöglicht es, den Regelanwender bei der Erstellung einer Annotation zu unterstützen, die mit einer Verfeinerungsregel korrespondiert. Die zweite Methode, 'matchingAnnotationPreDelete(RAnnotation): Suggest', ermöglicht es der Verfeinerungsregel auf das Löschen einer Annotation zu reagieren. Sie kann z.B. einen Hinweis geben, wenn es nicht ratsam ist die Annotation zu löschen. Es ist Aufgabe des Regelentwicklers diese Methode zu implementieren. Eine Implementierung dieser Methode könnte mit dem Benutzer in Form eines Dialogs kommunizieren und Operationen ausführen, die dafür sorgen, dass das Löschen der Annotation problemlos möglich ist.

Die Implementierungs-Grundstruktur der Methode 'apply(ASGElement):Boolean' weicht von der einer *TransformationRule* ab. Jene wird über ihren Namen mit einer Annotation assoziiert. Im Mustererkennungsteil muss deswegen geprüft werden, ob das übergebene Element vom Typ *RAnnotation* ist und den gleichen Namen, wie die Verfeinerungsregel trägt (siehe Abbildung 5.8). Weitere Mustererken-

nungsstrukturen, wie das Vorhandensein von Links oder Parametern, können dem Annotations-Objekt über ein Kontextmenü hinzugefügt werden. Nach Abschluss des Transformationsteils wird die Annotation gelöscht. Dies ist jedoch nicht zwingend notwendig und hängt von der jeweiligen Verfeinerungsregel ab. Wenn die Annotation nicht gelöscht werden soll, wird das Objekt *annotation* mit der Markierung <<destroy>> aus dem *Storypattern* entfernt.

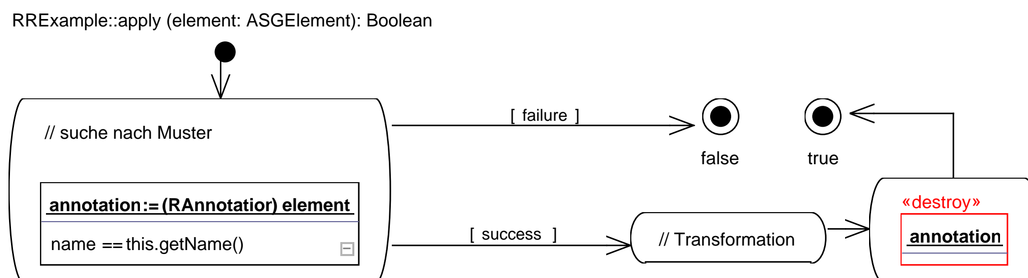


Abbildung 5.8: *Grundstruktur einer Regel vom Typ RefinementRule*

Umgang mit Regeln

Folgende Empfehlungen gelten für Regelentwickler:

- Jede Regel in einem eigenen Fujaba Projekt bearbeiten. Dies verhindert, dass während der Entwicklungsphase der Regel, Modelle aus anderen Diagrammen versehentlich modifiziert und eventuell unbrauchbar werden.
- Regeln mit Hilfe von Fujaba und einer Java-IDE entwickeln. Der Mustererkennungsteil einer Regel lässt sich mit dem in 4.3 vorgestellten Story Driven Modeling sehr einfach implementieren. SPin bietet zudem weitere Unterstützung für das Erstellen von Fragmenten, die im Mustererkennungsteil oft verwendet werden. Für die textuell zu implementierenden Teile des Transformationsprozesses ist es derzeit ratsam eine Java-Entwicklungsumgebung, wie z.B. Eclipse oder Together einzusetzten, da Fujaba in der derzeitigen Version nicht für textuelle Entwicklung optimiert ist. So fehlt beispielsweise eine *Autocomplete*-Funktion und ein Mechanismus, der während des Entwickelns automatisch testet, ob der erzeugte Code kompilierbar ist.

Das Verhalten einer Regel wird von der Implementierung der Methode 'apply(ASGElement):Boolean' bestimmt und somit vom Regelentwickler festgelegt. Er kann die gesamte Funktionalität der Java-API nutzen und die Funktionalität von Fujaba und SPin einsetzen, um Transformation durchzuführen. Dies schließt auch

das Erzeugen neuer Annotationen ein. Im Transformationsteil können neue Annotationen erstellt und einem Diagramm hinzugefügt werden.

Damit wird es möglich Relationen zwischen Transformationsregeln zu definieren. Eine für Annotation A verantwortliche Verfeinerungsregel RRA kann beispielsweise in ihrem Transformationsteil Annotation A löschen und eine neue Annotation B erzeugen, die anschließend von Verfeinerungsregel RRB transformiert werden kann.

Einsatz des SDM zum Erstellen von Transformationsregeln

Wird eine Regel erstellt, können die Vorteile des visuellen und textuellen Programmierens kombiniert werden. Mustererkennung und Wertzuweisung lassen sich meist besser graphisch darstellen, während andere Operationen als Text besser darstellbar sind.

Mit Hilfe des Story Driven Modeling können in einem Modell Muster gefunden werden. Wie in 4.3 bereits erwähnt, benötigt man für das Story Driven Modeling ein UML Klassendiagramm, welches das Metamodell (M_{i+1}) der Objekte (M_i) beschreibt, die im Storypattern bearbeitet werden sollen. Will man Elemente von UML Klassendiagrammen im SDM verwenden,[4] sind die Objekte eines Storypattern Instanzen dieser Elemente. Es wird deswegen das Modell der Fujaba-Implementierung des UML Metamodells für Klassendiagramme benötigt. Ein Teil dieses Modells wurde in 4.2 bereits dargestellt. Ist dieses Klassendiagramm im Projekt vorhanden, wird es möglich, dessen Elemente im SDM zu verwenden. Somit kann man Mustererkennungsprozesse auf UML Klassendiagrammen durchführen. Der Vorteil liegt darin, dass es im Allgemeinen einfacher ist, eine Mustererkennung graphisch zu formulieren, als diese mit selbstgeschriebenem Java-Code zu implementieren. Der vom SDM generierte Code ist fehlerfrei und die graphische Darstellung der Mustererkennung intuitiver als die textuelle.

Doch wie erhält man ein Klassendiagramm, welches das Metamodell der Objekte enthält, die im SDM bearbeitet werden sollen? Mögliche Lösungen werden in 5.2.4 im Abschnitt „Synchronisations-Modul“ vorgestellt.

Regel-Bibliothek

Die Regel-Bibliothek ist die zentrale Verwaltungsinstanz für Transformationsregeln. Sie durchsucht das lokale Bibliotheks-Verzeichnis nach gültigen Transformationsregeln, die in Form von Java-Klassen in JAR-Archiven gespeichert sind. Wird eine gültige Regel gefunden, erzeugt die Bibliothek eine Instanz der Regel-Klasse und fügt diese einer internen Liste hinzu. Auf den Regel-Instanzen kann operiert werden.

[4] wie dies bei der Modelltransformation von UML Klassendiagrammen der Fall ist

Dieser Mechanismus gestattet es, Regeln während der Laufzeit in die Bibliothek zu laden oder aus ihr zu entfernen. Durch das Zusammenspiel dieses Lade-Mechanismus mit dem bereits vorgestellten Export-Mechanismus ist es möglich Regeln zu bearbeiten und ohne Neustart des Programms zu verwenden.

Exkurs: Hardwareentwicklung Der Einsatz einer Bibliothek, in der Elemente gespeichert werden, die als Vorlagen oder Bausteine verwendet werden, ist nicht neu. In der Hardwareentwicklung ist dies bereits seit Jahren gängige Praxis (Stichwort *Computer-Aided Integrated Circuit Design*). Mit einer Hardwarebeschreibungssprache und der entsprechenden Entwicklungsumgebung ist es möglich einen Hardwarebaustein graphisch zu modellieren. Die Entwicklungsumgebung erlaubt es (Standard-)Bausteine aus einer Bibliothek auszuwählen und in einen eigenen Baustein zu integrieren. Aus dem resultierenden Modell, das Verhalten und Struktur des Hardwarebausteins enthält, kann ein Plan generiert werden, der die Geometrie des Bausteins beschreibt. Diesen kann ein Chiphersteller verwenden um den gewünschten Hardwarebaustein zu produzieren.

Einen Überblick über den Ablauf des Hardware-Entwurfs, der dem des Software-Entwurfs ähnlich ist, gibt z.B. [21].

5.2.3 Transformation

Transformationsregeln werden mit Hilfe eines Transformators (*Transformator*, siehe 5.9) ausgeführt. Dazu ruft dieser die in 5.2.2 beschriebenen Einsprungsfunktionen der Klasse *TransformationRule* auf.

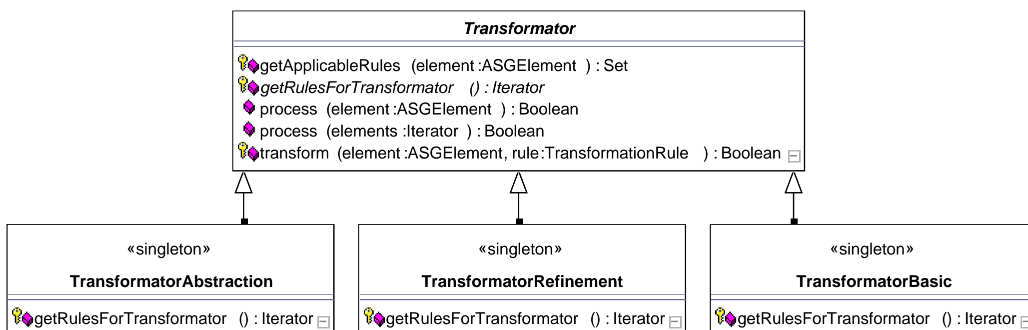


Abbildung 5.9: *Die Transformator-Architektur von SPin*

Transformator ist eine abstrakte Basisklasse, die konkreten Transformatoren die benötigte Funktionalität bereitstellt. Diese müssen die Methode 'getRulesForTransformator():Iterator' implementieren, die eine Menge von Transformationsregeln zurückgibt, welche der Transformator im Transformationsprozess verwenden

soll. Regelentwickler können mit Hilfe dieses Mechanismus eigene Transformatoren implementieren, die nur eigens erzeugte Regeltypen ausführen. Gestartet werden diese durch Aktionen, die über ein Fujaba-Plug-in dem Fujaba-Userinterface hinzugefügt werden können.

In SPin gibt es drei Transformator-Implementierungen: *TransformatorAbstraction*, *TransformatorRefinement* und *TransformatorBasic*. Die beiden ersten ermöglichen die Navigation zwischen verschiedenen Strata und sind für die Zwecke der Stratifikation vorgesehen. Letzterer ist ein Basis-Transformator, der alle Regeln bis auf die für Stratifikation verwendet. Mit ihm können Regeln ausgeführt werden, die von *TransformationRule* erben, ohne dass ein Regelentwickler einen eigenen Transformator implementieren muss. Welche Regeln der jeweilige Transformator im Transformationsprozess verwendet, ist in Tabelle 5.2 dargestellt.

Transformator	zugeordnete Regeln
TransformatorAbstraction	*AbstractionRule*
TransformatorRefinement	*RefinementRule*
TransformatorBasic	!(*AbstractionRule* || *RefinementRule*)

Tabelle 5.2: *Zuständigkeit von Transformatoren für Regeln*

Der Transformationsprozess

Der Transformationsprozess wird durch den Aufruf der Methode 'process' gestartet. Diese kann sowohl ein einzelnes Element als auch eine Menge von Elementen eines ASG entgegennehmen. Sollen mehrere Elemente transformiert werden, geschieht dies der Reihe nach. Der Transformator überprüft zunächst, welche Regeln auf dem zu transformierenden Element anwendbar sind. Dazu holt er sich aus der Bibliothek sämtliche Regeln, für die er zuständig ist und überprüft anschließend in der Methode 'getApplicableRules(ASGElement):Set', welche dieser Regeln das zu transformierende Element transformieren könnten. Um dies herauszufinden, wird die Methode 'isApplicable(ASGElement):Boolean' auf den Regeln aufgerufen. Gibt es mehrere Regeln, die angewendet werden können, muss der Benutzer entscheiden, welche Regel ausgeführt werden soll. Steht die auszuführende Regel fest, wird auf ihr die Methode 'apply(ASGElement):Boolean' aufgerufen. Somit erhält die Regel die Kontrolle und kann ihr Programm ausführen. Tritt ein Fehler oder eine Ausnahme in der Regel auf, wird diese vom Transformator gefangen und dem Regelanwender mitgeteilt.

Anmerkung

Der oben beschriebene Mechanismus ist mit dem von Programmiersprachen, wie Java oder C/C++ vergleichbar. In diesen Programmiersprachen gibt es definierte Einsprungsfunktionen, über die ein in der entsprechenden Sprache geschriebenes kompiliertes Programm ausgeführt werden kann. In Java ist dies die Methode 'public static void main(String[] args)' und in C/C++ 'int main()' oder 'int main(int argc, char* argv[])'. Eine von SPin definierte Einsprungsfunktion ist die Methode 'public boolean apply(ASGElement)'.

5.2.4 Hilfskonstrukte

SPin stellt diverse Hilfskonstrukte zur Verfügung, die für interne Zwecke eingesetzt werden. Auch anderen Entwicklern ist es möglich diese Funktionalität z.B. in Transformationsregeln zu verwenden. SPin stellt dafür eine API bereit. Zu diesen Hilfskonstrukten zählen drei Fabriken, ein Synchronisations-Modul, ein Methoden-Änderungsmechanismus, eine allgemeine Hilfsklasse, sowie ein Mechanismus der es gestattet mit Benutzern zu kommunizieren.

Fabriken

Die von SPin zur Verfügung gestellten Fabriken sind in 5.10 dargestellt. Methoden und Attribute sind aus Platzgründen verborgen. Eine vollständige Abbildung der Fabriken findet sich in Anhang A.1. Jede der Fabriken hat die Aufgabe Elemente eines Metamodells zu erzeugen und Hilfskonstrukte zur Verfügung zu stellen, die die Bearbeitung der erzeugten Elemente erleichtern. Die Fabriken reduzieren den Aufwand für die Erzeugung eines Elements auf einen Funktionsaufruf. Dies hat für Regelentwickler den Vorteil, dass der zu implementierende Code für die Erzeugung eines Elements auf ein Minimum beschränkt wird. Transformationsregeln bleiben übersichtlich und die Erzeugungsaufrufe weisen ein bestimmtes Muster auf. Jede Fabrik ist als *Singleton* implementiert. So existiert nur jeweils ein Fabrik-Objekt, dessen Funktionalität über einen globalen Zugriffspunkt nutzbar ist.

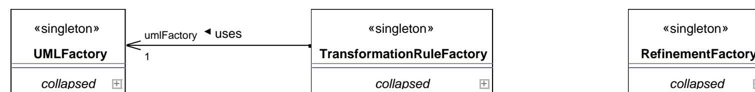


Abbildung 5.10: *Fabriken von SPin, die für interne Zwecke genutzt werden und einem Regelentwickler einfache Schnittstellen für die Erzeugung neuer Elemente bieten.*

UMLFactory Fujaba bietet zur Zeit keine zentrale Instanz, die Elemente von UML Klassendiagrammen oder Storydiagrammen erzeugt. Eine solche Instanz wird es erst in der kommenden Version 5 geben. Eine solche Instanz würde die Erstellung von Transformationsregeln jedoch sehr vereinfachen. Aus diesem Grund habe ich eine Fabrik implementiert, die das Erzeugen der wichtigsten Elemente für Klassen- und Storydiagramme übernimmt. Desweiteren bietet die *UMLFactory* die Möglichkeit, erzeugte Elemente eines Klassendiagramms mit Hilfe des Synchronisations-Moduls (siehe 5.2.4) abzugleichen. Die „wichtigsten Elemente" für Klassendiagramme sind:

> Klassendiagramme, Kommentare, Pakete, Klassen, Attribute, Methoden, Generalisierungen, Assoziationen, Typen

und für Storydiagramme:

> Storydiagramme, Story- und Statement-Aktivitäten, Transitionen, Objekte, Links und Attribut-Wert-Paare.

TransformationRuleFactory Die *TransformationRuleFactory* bietet neben der Hauptaufgabe des Erzeugens einer Transformationsregel auch update-Funktionen für generierbare Konstrukte einer Transformationsregel. Dies sind die Methoden 'isApplicable(ASGElement):Boolean' und 'getAnnotation():RAnnotation', sowie Import-Anweisungen, die in der generierten Java-Implementierung einer Transformationsregel benötigt werden. Für in Verfeinerungsregeln häufig eingesetzte Mustererkennungs-Teile besitzt die Fabrik Generierungsfunktionen. Diese erzeugen Konstrukte in einem *Story-Pattern*, die prüfen, ob ein RefinementElement bestimmte Parameter besitzt, ob ein RefinementLink zu einem bestimmten Typ von *ASGElement* vorhanden oder die Annotation einem bestimmten Typ von *ASGElement* zugeordnet ist.

In Zusammenarbeit mit dem Synchronisations-Modul erzeugt diese Fabrik zudem die von SPin zur Regelentwicklung mit dem SDM benötigten Metamodelle und legt diese in UML Klassendiagrammen ab.

RefinementFactory Die *RefinementFactory* ermöglicht es, die in Abschnitt 5.2.1 eingeführten Elemente des Annotierungs-Metamodells zu erzeugen.

Synchronisations-Modul

Für die Erstellung des Mustererkennungsteils von Transformationsregeln wird ein Klassendiagramm benötigt, welches die Strukturen des Fujaba UML- und des SPin Annotierungs-Metamodells enthält. Nur so wird das SDM für die Erstellung

von Transformationsregeln praktikabel. Sind jene vorhanden, können Mustererkennungs- und Verwaltungsprozesse mit Hilfe des SDM durchgeführt werden. Wie in 17 bereits angesprochen, stellt sich die Frage, wie man ein Klassendiagramm erhält, das die Strukturen des Metamodells der Objekte beinhaltet, die im SDM bearbeitet werden sollen.

Zunächst kann davon ausgegangen werden, dass im Falle eines Regelentwicklers, der das SDM nutzen möchte, eine Java-Implementierung der zu verwendenden Funktionalität existiert. Sowohl Fujaba als auch SPin sind in Java implementiert. Der Quellcode beider Programme steht zur Einsicht bereit. Die Implementierung beider ist in Form von JAR-Archiven vorhanden, die den Java-Bytecode in Form von class-Dateien enthalten.

Es gibt im wesentlichen zwei Ansätze für die Lösung des oben genannten Problems. Die benötigten Strukturen können von Hand erzeugt oder mit Hilfe eines Parsers generiert werden.

Der erste Ansatz verlangt explizites Wissen über die Sprachen UML und Java, Kenntnisse in Objektorientierter Analyse und Design und praktische Erfahrung im Programmieren. Der Quellcode muss analysiert und die benötigten Strukturen in ein UML Klassendiagramm umgesetzt werden. Dieser Vorgang ist zeitintensiv, fehleranfällig und verlangt manuelle Nachbesserung, sobald sich Strukturen des Metamodells ändern. Er ist also nur wenig empfehlenswert. Das Vorgehen im zweiten Ansatz stimmt mit dem des ersten überein. Der Unterschied ist jedoch, dass ein Mechanismus die Aufgaben „Analyse der Strukturen“ und „Umsetzung in ein Diagramm“ übernimmt und automatisiert.

Ein solcher Mechanismus existiert bereits in Form des *JavaParser-Plug-in*s für Fujaba, das von der Universität Paderborn bereitgestellt wird. Dieser Mechanismus ist in der Lage Java-Quellcode zu analysieren und daraus eine UML-Repräsentation zu generieren. Das entstehende Modell enthält sämtliche strukturellen Informationen und die Implementierung der Struktur. Es kann mit Hilfe eines nachgeschalteten Struktur-Analysators abstrahiert werden. Dieser erkennt z.B. eine Struktur, die eine Assoziation repräsentiert, sofern diese einer Namenskonvention folgt und ersetzt sie durch eine UML Assoziation. Das so entstandene Modell kann im SDM eingesetzt werden.

Ein von mir entwickelter Mechanismus baut auf der Annahme auf, dass die Implementierung des Modells für das SDM unerheblich ist und nur die Struktur benötigt wird. Diesen Mechanismus bezeichne ich mit *Synchronisations-Modul*. Er ist in der Klasse *VMSynchronizer* implementiert (siehe Abbildung 5.11) und verwendet die Java-Mechanismen *ClassLoader* und *Reflection-API* [5]. Diese ermöglichen es, Java-Klassen zur Laufzeit zu laden und deren Struktur zu analysieren. Standardmäßig sind die Klassen der Java- und Fujaba-API erreichbar. Desweiteren kann auf Klassen von Fujaba-Plug-ins, sowie auf verwendete Bibliotheken zugegriffen wer-

[5] Paket *java.lang.reflect*

den. Zudem ist der Lademechanismus transparent, d.h. der *ClassLoader* findet die Klassen automatisch. Der Analyse-Prozess erfragt mit Hilfe der *Reflection-API* die strukturellen Daten der Java-Klasse und erzeugt anhand der Daten eine UML Repräsentation. Diese kann, ebenfalls mit Hilfe eines nachgeschalteten Struktur-Analysators, abstrahiert und so für das SDM brauchbar gemacht werden.

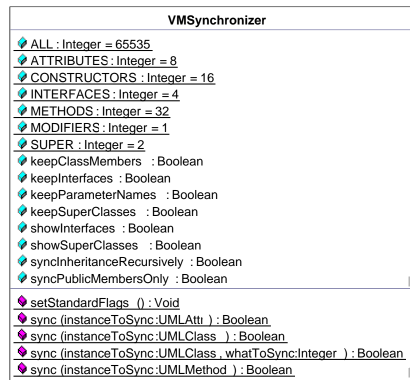


Abbildung 5.11: *Der Virtual-Machine–Synchronizer (VMSynchronizer) von SPin. Er ermöglicht die Synchronisation von UML Klassen mit korrespondierenden Java-Klassen.*

Der Vorteil dieses Mechanismus ist, dass er keinen Quellcode benötigt, sondern auf dem immer vorhandenen, aktuellen Bytecode arbeitet. Desweiteren ist die Schnittstelle zum Synchronisieren sehr einfach. Dem Synchronisations-Modul wird lediglich eine UML Klasse mit voll qualifiziertem Namen übergeben, der Rest erfolgt automatisiert. Abbildung 5.12 zeigt den Synchronisations-Prozess, der über das Userinterface ausgelöst wird und die UML Klasse *java.lang.Object* mit ihrem Java-Pendant abgleicht.

Ein Nachteil dieses Ansatzes ist, dass Parameter-Namen von Methoden mit der Reflection-API nicht rekonstruiert werden können. So können Parametern keine aussagekräftigen Namen zugewiesen werden. Dies ist für das SDM jedoch unerheblich. Methoden werden dort vom Entwickler nicht explizit genutzt, sondern nur die davon abstrahierten Attribute und Assoziationen.

Methoden-Modifizierer

Der *Methoden-Modifizierer* ist in der Klasse *UMLActivityDiagramModifier* implementiert (siehe Abbildung 5.13). Er verwendet einen von mir implementierten *Visitor* (*UMLActivityDiagramVisitor*), der Einsprungsfunktionen für jedes Element bietet, das regulär in einem *UMLActivityDiagram* enthalten sein kann[6]. Die Me-

[6] wobei *UMLActivityDiagramVisitor* in dieser Implementierungsstufe nicht alle regulären Elemente eines *UMLActivityDiagrams* unterstützt

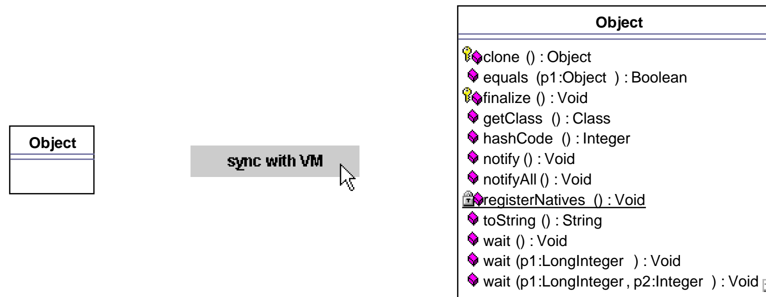


Abbildung 5.12: *Schema des Synchronisations-Prozesses: Auf der UML Klasse java. lang.Object (links) wird über ihr Kontextmenü der Synchronisationsvorgang ausgelöst. Es entsteht eine mit der Java-Klasse java.lang.Object synchrone UML Klasse (rechts).*

thode 'modify(UMLActivityDiagram, UMLActivityDiagramVisitor):Void' ruft für jedes reguläre Diagramm-Element die zugehörige Funktion eines Visitors auf, der von *UMLActivityDiagramVisitor* erbt. Neue Visitor-Klassen, die eine bestimmte Funktionalität zur Veränderung einer Methode bieten, können implementiert werden.[7]

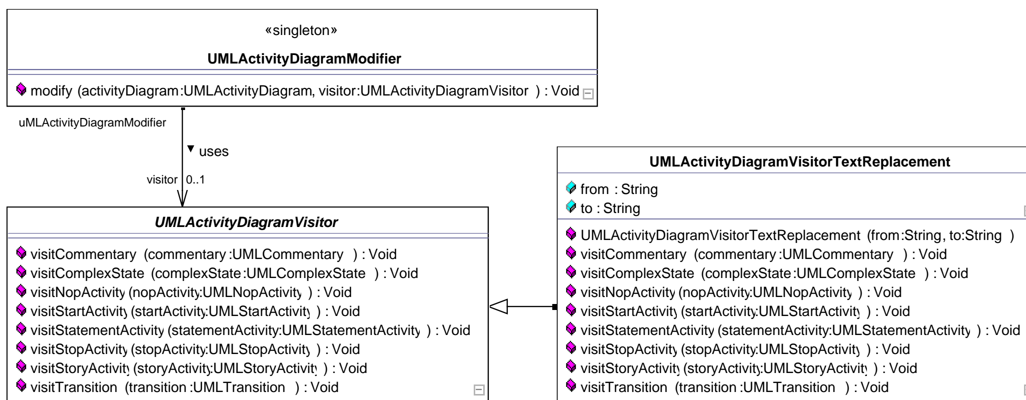


Abbildung 5.13: *Hilfskonstrukt von SPin: Methoden-Änderungsmechanismus*

Die Visitor-Implementierung *UMLActivityDiagramVisitorTextReplacement* führt eine einfache Textersetzung in Methoden durch. Es werden alle Vorkommnisse des Attribut-Werts „from“ durch den Wert von „to“ ersetzt und zwar in folgenden Teilen des Sourcecodes:

[7] z.B. von einem Regelentwickler

- in Kommentaren
- in Parameter-Namen
- im Code einer UMLStatementActivity (ohne Rücksicht auf die Semantik des Code-Fragmentes)
- im Rückgabewert der Methode
- im Namen eines *UMLObject*s
- im Ausdruck eines *UMLAttrExprPair*s
- im boole'schen Ausdruck eines Transition-Guards

Eine Transformationsregel, die den Methoden-Modifizierer verwendet, ist in Anhang B.6 dargestellt.

SPinHelper

Die in Abbildung 5.14 dargestellte Klasse *SPinHelper* bietet hilfreiche Funktionen, die von einem Regelentwickler häufig benötigt werden. Eine ausführliche Dokumentation ist unter [14] zu finden.

```
SPinHelper
LEFT_AGG : String = KindComboBox.LEFT_AGG
LEFT_COMP : String = KindComboBox.LEFT_COMP
LEFT_REF : String = KindComboBox.LEFT_REF
NORMAL : String = KindComboBox.NORMAL
RIGHT_AGG : String = KindComboBox.RIGHT_AGG
RIGHT_COMP : String = KindComboBox.RIGHT_COMP
RIGHT_REF : String = KindComboBox.RIGHT_REF

addUnimplementedMethods (umlClass :UMLClass ) : Void
changeTypeInAllTypeUsers (from:UMLType, to:UMLType ) : Void
cleanTypeList (typeList:UMLTypeList ) : Void
createFullClassName (packageName :String, className :String ) : String
firstCharOfClassNameToUpperCase (fullQualifiedClassName :String ) : String
firstCharToLowerCase (aString:String ) : String
firstCharToUpperCase (aString:String ) : String
getFullMethodName (javaConstructor :Constructor ) : String
getFullMethodName (javaMethod:Method ) : String
getFullNameOfType (umlType:UMLType ) : String
getJavaMethodDeclaration (umlMethod :UMLMethod, includeModifiers :Boolean , includeResultType :Boolean , includeParameterNames :Boolean ) : String
getJavaNameOfType (umlType:UMLType ) : String
getMainFSAObject (element :ASGElement ) : FSAObject
getNameOfType (javaType:Class ) : String
getNameWithoutPackage (fullClassName :String ) : String
getPackageWithoutName (fullClassName :String ) : String
getUMLNameOfType (javaClass :Class ) : String
getUMLNameOfType (typeName :String ) : String
instanceOf (umlClass :UMLClass , fullClassName :String, justCompareParentClasses :Boolean ) : Boolean
instanceOf (umlClass :UMLClass , javaClass :Class , justCompareParentClasses :Boolean ) : Boolean
isImplemented (umlMethod :UMLMethod ) : Boolean
isTypeInUse (umlType:UMLType ) : Boolean
iteratorOfAllChildClasses (umlClass :UMLClass ) : Iterator
iteratorOfAllParentClasses (umlClass :UMLClass ) : Iterator
loadClassByName (fullJavaClassName :String, loader :ClassLoader ) : Class
newJavaInstance (fullJavaClassName :String, loader :ClassLoader ) : Object
printFSAHierachy (asgElement :ASGElement ) : Void
serialCopy (objectToCopy:Object ) : Object
setAssociationKind (umlAssoc :UMLAssoc, newKind :String ) : Void
vectorOfAllChildClasses (umlClass :UMLClass ) : Vector
vectorOfAllParentClasses (umlClass :UMLClass ) : Vector
```

Abbildung 5.14: *Hilfskonstrukt von SPin: eine Klasse, die Regelentwicklern wichtige Funktionalität bereitstellt*

UserMessage

Oft ist es hilfreich, mit dem Benutzer zu kommunizieren, ihm Informationen über Vorgänge mitzuteilen oder auf Unerwünschtes hinzuweisen. Mit der Klasse *User-Message* (siehe Abbildung 5.15) bietet SPin Regelentwicklern dafür eine zentrale Schnittstelle. Neben der Ausgabe von Nachrichten ist es möglich dem Benutzer einfache Ja/Nein-Fragen zu stellen, auf die eine Regel während ihrer Ausführung reagieren kann.

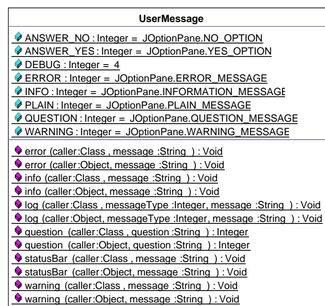


Abbildung 5.15: *Hilfskonstrukt von SPin: ein Mechanismus, der es gestattet Benutzer Meldungen anzuzeigen, oder ihnen Fragen zu stellen*

5.3 Userinterface

Ziel dieses Abschnitts ist es, den Leser mit den Prinzipien der Benutzungsschnittstelle von SPin vertraut zu machen.

Funktionalität von SPin wird über Aktionen aufgerufen, die durch (Kontext-) Menü-Einträge oder Knöpfe ausgelöst werden. SPin-eigene Elemente besitzen eigene Kontextmenüs. An Kontextmenüs fremder Elemente wird ein Submenu Namens „SPin“ angehängt, über das kontextbezogene Funktionalität von SPin genutzt werden kann.

Die SPin-Toolbar (siehe Abbildung 5.16) ist in jedem Diagramm vorhanden. Mit ihrer Hilfe können folgende Aktionen ausgelöst werden:

- Anzeigen des Annotierungs-Editors,
- Expandieren/Kollabieren von Annotationen,
- Anzeigen der Regel-Bibliothek,
- Anwenden von Transformationsregeln,

- Anwenden von Abstraktionsregeln,
- Anwenden von Verfeinerungsregeln und
- Anwenden von Verfeinerungsregeln eines bestimmten Typs.

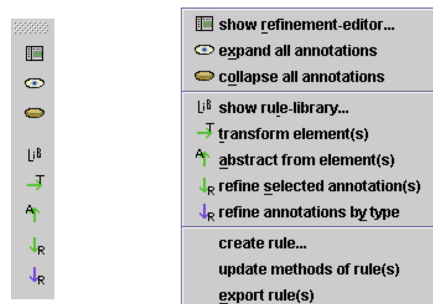


Abbildung 5.16: *Die Toolbar von SPin (links) und das SPin-Kontextmenü für UML Klassendiagramme (rechts)*

Transformationsregeln

Über das erweiterte Kontextmenü für UML Klassendiagramme hat man die Möglichkeit eine Transformationsregel zu erstellen. Diese wird als UML Klasse in das Klassendiagramm eingefügt. Ein Dialog, mit dessen Hilfe eine neue Regel erzeugt werden kann, ist in Abbildung 5.17 dargestellt.

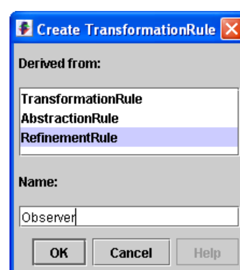


Abbildung 5.17: *Ein Dialog, mit dessen Hilfe eine neue Regel erzeugt werden kann.*

UML Klassen, die eine Transformationsregel implementieren, können exportiert und automatisch der Regel-Bibliothek hinzugefügt werden.

Im Implementierungsdiagramm einer Regel (Storydiagramm der Methode 'apply(ASGElement):Boolean') sind bestimmte Teile der Mustererkennung mit Unterstützung von SPin generierbar. So können oft genutzte Mustererkennungsstrukturen erzeugt werden. Derzeit sind dies Muster, die prüfen, ob

- einem Element des Annotierungs-Metamodells ein Parameter mit festgelegtem Namen und Typ zugeordnet ist,
- eine Annotierung an ein Element bestimmten Typs gebunden ist,
- ein Link existiert, der eine Annotation mit einem Element bestimmten Typs verbindet, welches einen festgelegten Rollennamen trägt.

Annotierungs-Editor

Der Annotierungs-Editor (siehe Abbildung 5.18) ermöglicht es, Annotierungen in ein Diagramm einzufügen und zu bearbeiten. So können Name und Ziel der Annotierung geändert, sowie ausgehende Links und Parameter editiert werden. Das Kontextmenü einer Annotierung gestattet es zudem, das Implementierungsdiagramm der korrespondierenden Verfeinerungsregel zu öffnen, sofern dieses im Projekt vorhanden ist. Dies hilft bei der Entwicklung von Verfeinerungsregeln, da nicht mehr manuell nach der Implementierung der korrespondierenden Regel gesucht werden muss.

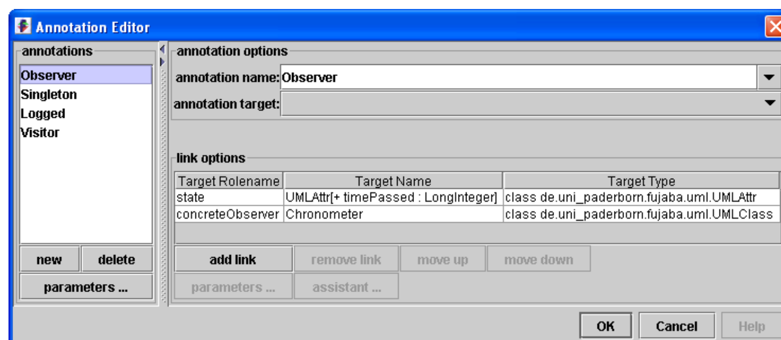


Abbildung 5.18: *Ein Editor zum Erstellen und Bearbeiten von Annotationen, Links und Parametern*

In der Architektur von SPin ist ein Assistent vorgesehen, der Hilfestellung bei der Erzeugung von Annotationen geben soll. Jede Annotation besitzt eine korrespondierende Verfeinerungsregel. Sofern diese in der Regel-Bibliothek existiert, wertet der Assistent den Rückgabewert der Methode 'getAnnotation():RAnnotation' der Klasse *RefinementRule* aus. Dieser beschreibt, welche Struktur eine Annotation

besitzen muss, damit der Mustererkennungsteil der Verfeinerungsregel erfolgreich ist. Der Assistent vergleicht die gerade bearbeitete Annotation mit der benötigten Struktur anhand der folgenden Fragen.

- Muss die Annotation einem Element zugeordnet sein? Wenn ja, von welchem Typ muss das Element sein?
- Wieviele Links müssen von ihr ausgehen?
- Welcher Rollenname muss den Links zugeordnet werden?
- Welcher Link muss die Annotierung mit welchem Element-Typ verbinden?
- Was für Parameter müssen Annotation und Links zugeordnet sein, wie heissen diese und von welchem Typ müssen sie sein?

Stimmen die benötigte und tatsächliche Struktur nicht überein, weist der Assistent den Benutzer darauf hin und leitet ihn an, die Annotierung so zu bearbeiten, dass sie mit der benötigten Struktur übereinstimmt.

Transformation

Der Transformationsvorgang wird über die Aktionen „transform elements“, „abstract from elements“, „refine selected annotations“ oder „refine annotations by type“ gestartet. Im Hintergrund wird einer der drei von SPin bereitgestellten Transformatoren gestartet, der eine ausführbare Transformationsregel auf den selektierten Elementen anwendet.

Synchronisations-Modul

UML Klassen können über ihr Kontextmenü mit einer korrespondierenden Java-Klasse synchronisiert werden.

Einstellungen

Über den Menüpunkt Options → Plug-ins Preferences → Spin gelangt man in die Grundeinstellungen von SPin (siehe Abbildung 5.19). Hier können der Pfad der Regel-Bibliothek, die Farben für Annotierungen, Links und deren Schrift geändert werden. Weiterhin kann man festlegen, ob das von SPin für die Bearbeitung von Transformationsregeln benötigte Metamodell dem Projekt automatisch hinzugefügt werden soll, sobald eine neue Regel erstellt wird.

Im Menüpunkt Tools → SPin kann das benötigte Metamodell für Transformationsregeln dem Projekt manuell hinzugefügt werden. Existiert es bereits, wird

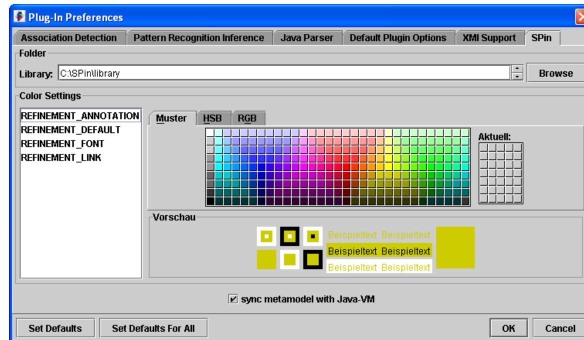


Abbildung 5.19: *Einstellungsmöglichkeiten für SPin*

es mit der aktuellen Implementierung synchronisiert. Dies ist hilfreich, falls eines der Metamodelle geändert oder vom Regelentwickler unbeabsichtigt Elemente der generierten Metamodelle gelöscht oder modifiziert worden sind.

5.4 Implementierungsdetails

Es folgen einige Details, die bei der Implementierung von SPin hilfreich waren.

5.4.1 Fujaba Quellen

Die Entwicklerversion der Fujaba Implementierung kann dem Fujaba-CVS-Server entnommen werden. Dessen Addresse ist unter `http://uther.uni-paderborn.de/projects/fujaba/` erhältlich. Derzeit aktuell ist die Fujaba Version 5, die im Hauptzweig entwickelt wird. Für die derzeitige Version von SPin[8] ist jedoch die Fujaba Entwickler-Version 4 erforderlich. Diese kann durch Angabe des CVS-Tags 'Version_4_Maintainance' vom CVS-Server angefordert werden.

Anleitungen zum Erstellen eines Plug-ins und Informationen über Techniken, die in Fujaba eingesetzt werden, sind auf der Fujaba-Homepage [20] zu finden.

5.4.2 Überblick über Klassen von SPin

Hier sollen einige Daten zur Implementierung von SPin genannt werden, damit ihre Größenordnung in etwa abgeschätzt werden kann. Detailliertere Informationen über die API von SPin ist aus Platzgründen nicht in dieser Arbeit enthalten. Eine

[8] für die aktuellste Version siehe [14]

Darstellung der SPin-API in Form einer Java-Dokumentation findet man unter [14].

SPin besteht derzeit aus 78 Klassen. 29 dieser Klassen sind Aktionsklassen, über die der Benutzer die Funktionalität von SPin nutzen kann. Die drei wichtigsten Pakete sind:

- *de.tud.SPin.metamodel*
- *de.tud.SPin.transformation*
- *de.tud.SPin.util*

Ein Überblick über Vererbungs- und Abhängigkeitsbeziehungen zwischen diesen Paketen und Fujaba ist in Abbildung 5.20 dargestellt.

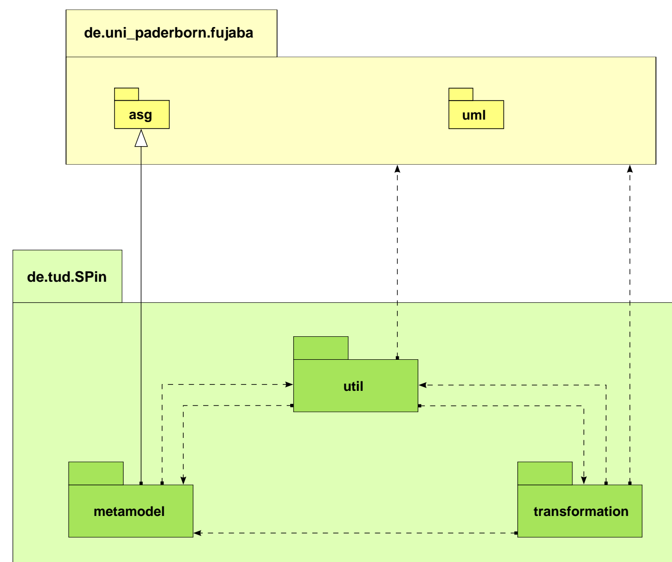


Abbildung 5.20: *Überblick über Vererbungs- und Abhängigkeitsbeziehungen zwischen SPin und Fujaba*

Paket *de.tud.SPin.metamodel* enthält das Datenmodell des Annotierungs-Metamodells und dessen Visualisierung. In ihm sind 11 Klassen enthalten. In Paket *de.tud.SPin.transformation* befinden sich 8 Klassen, die Transformationsregeln, Transformator und Bibliothek implementieren. Für Regelentwickler dürfte das Paket *de.tud.SPin.util* von größtem Nutzen sein. In ihm finden sich 14 Hilfsklassen, die von SPin bereitgestellt werden. Sie stellen Funktionen zur Verfügung, die in Transformationsregeln häufig benötigt werden. Diese können in eigenen Regeln verwendet werden.

5.4.3 Debuggen von Transformationsregeln

Ein Regelentwickler wird des öfteren die Implementierung einer Regel testen wollen. Aus diesem Grund soll hier anhand der Entwicklungsumgebung Eclipse exemplarisch erläutert werden, wie dies möglich ist.

Läuft SPin im Debug-Modus, lässt es den Java-Sourcecode einer Regel von Fujaba generieren, verschiebt diesen anschließend in das Bibliotheksverzeichnis, kompiliert den Sourcecode und lädt den so entstandenen Java-Bytecode als ausführbare Regel in die SPin Regel-Bibliothek. Quellcode und zugehöriger Bytecode liegen nach diesem Vorgang im selben Verzeichnis. Um den Source-Code einer Regel debuggen zu können, muss ein Java-Projekt angelegt werden, welches im Java-Build-Path das Bibliotheksverzeichnis als Quellordner enthält. Da Eclipse die .java-Dateien der Regeln jedoch jedes mal neu kompiliert, sobald Regeln im Source-Code geändert werden und die Option 'build automatically' aktiviert ist, muss das Ausgabeverzeichnis, in das Eclipse die Regel-Sourcen kompiliert, auf ein Verzeichnis umgestellt werden, welches nicht das library-Verzeichnis oder ein Unterverzeichnis davon ist.

Fujaba wird nun im Debug-Modus in Eclipse gestartet. Dabei muss darauf geachtet werden, dass als 'Source Lookup Path' in den Debug Einstellungen für den Fujaba Debug-Modus das Regel-Projekt angegeben ist. Nach jeder Änderung des Sourcecodes der Regel muss der Sourcecode in Eclipse „refreshed“ werden, da ansonsten der Sourcecode nicht mit dem Java-Bytecode der Regel übereinstimmt. Nun kann in die Regel ein Breakpoint eingefügt und die Regel so debuggt werden.

Kapitel 6

Ergebnisse

„Angenehm sind die erledigten Arbeiten“

– Cicero

Der ursprüngliche Gedanke hinter SPin ist die Realisierung von Architektur-Stratifikation. Dies ist jedoch nicht das einzige, wozu SPin eingesetzt werden kann. In den nachfolgenden Abschnitten werden Einsatzmöglichkeiten für SPin aufgezeigt und Transformationsregeln vorgestellt, die im Laufe dieser Arbeit entwickelt wurden.

6.1 Stratifizierung

Mit SPin ist es nun möglich Stratifikation auf UML und anderen Modellen anzuwenden. Dazu werden Transformationsregeln benötigt, die bestimmte Annotierungen verfeinern und Transformationsregeln, die Verfeinerungen zu Annotierungen abstrahieren. Dabei hat der Regelentwicklers dafür zu sorgen, dass diese Regeln auch das machen, was sie sollen. SPin hat darauf keinen Einfluss. Es bietet lediglich die Infrastruktur für die Navigation zwischen Abstraktionsebenen und unterstützt den Regelentwickler bei der Bearbeitung von Transformationsregeln.

6.2 Automatisierung

Das eingeführte Transformationssystem ist ein sehr mächtiges Konstrukt, da es Skript-Funktionalität und die Möglichkeiten einer Programmiersprache (Java) vereint. Die Erstellung von Transformationsregeln ist mit den in Abschnitt 5.2 eingeführten Verfahren nicht schwer und bietet ein hohes Potential. Das System er-

möglicht die Bearbeitung von Elementen eines abstrakten Syntaxgraphen, also z.B. ein UML Diagramm, zu automatisieren. So können in Transformationsregeln Makros definiert, Entwurfsmuster codiert oder parametrisierte Skripte abgearbeitet werden. Elemente können erzeugt, gelöscht oder geändert werden, ohne dabei Aktionen im konventionellen Userinterface des CASE-Tools zu tätigen. Dies kann beim Erzeugen eines Modells Zeit sparen, da die Umsetzung des Gewünschten automatisiert und vereinfacht wird. Es muss weniger Denkarbeit bei der Umsetzung aufgewendet werden, da das Gewünschte nur einmal in einer Regel implementiert wird. Die Schnittstelle zur Umsetzung wird zudem vereinfacht, denn es wird auf einer abstrakten Ebene modelliert. Die neue Schnittstelle enthält zusätzlich die Benutzungsschnittstelle von SPin (siehe Abschnitt 5.3), mit der Annotierungen bearbeitet und Transformationen ausgeführt werden können.

Voraussetzung für eine Automatisierung ist, dass die gewünschte Bearbeitung von Elementen automatisierbar ist, also gewisse Muster zu erkennen sind. Ferner muss es sich lohnen, den Automatisierungsprozess in eine Transformationsregel zu übersetzen. Prädestiniert sind beispielsweise Transformationsregeln, die Codefragmente generieren, welche das Aussehen eines Java-Swing-Dialogs implementieren. Java-Swing-Dialoge bestehen zumeist aus Knöpfen, Listen und Textfeldern, die in bestimmten Bereichen des Dialogs angeordnet sind. Der Java-Code, der diese Elemente erzeugt und anordnet, weist meistens bestimmte Muster auf, die unter Angabe von bestimmten Parametern generierbar sind.

Wissen über die Programmiersprache Java, die UML und Fujaba ist für die Erzeugung von Transformationsregeln, die komplexe Operationen auf UML Diagrammen durchführen sollen, nötig. Dies sollte für einen Modellierer jedoch keine große Hürde darstellen, da er es gewohnt sein dürfte neben seiner Modelliertätigkeit auch zu programmieren. So kann er nach einer kurzen Einarbeitungszeit den vollen Umfang des in dieser Arbeit eingeführten Transformationssystems nutzen, das auf seine Bedürfnisse angepasste Prozesse ausführt.

6.3 Hot-Plug-in Module

Der Transformationsmechanismus von SPin kann dazu verwendet werden Module in Form einer Transformationsregel zu implementieren. Diese Module sind während der Laufzeit änderbar. Neue Mechanismen stehen so ohne Programmneustart zur Verfügung. Ein CASE-Tool-Anwender kann z.B. neue Funktionalität entwickeln und diese auf ihn maßgeschneiderte Funktionalität sofort im CASE-Tool einsetzen.

So ist es möglich, eine Fabrik als Transformationsregel zu implementieren. Wäre die *TransformationRuleFactory* beispielsweise als Regel implementiert, könnte der Erzeugungsvorgang von Transformationsregeln während der Laufzeit geändert und an neue Bedürfnisse angepasst werden.

Desweiteren können Hilfskonstrukte, die bestimmte Vorgänge automatisieren, mit Transformationsregeln als *Hot-Plug-in Modul* implementiert werden (siehe auch 6.2, 6.5 und 6.6).

6.4 Evolution

Mit SPin ist es möglich, eine Verfeinerungsregel zu implementieren, die eine neue Regel generiert. Somit ist eine Verfeinerungsregel in der Lage, ihren Typ weiterzuentwickeln. Sie ist mit einem Werkzeug vergleichbar, das sich selbst verbessern kann. Die dafür benötigte Funktionalität wird von der *TransformationRuleFactory* bereitgestellt. Die Verfeinerungsregel ist durch das eingeführte Annotierungs-Metamodell über die Konstrukte *RefinementLink* und *RefinementParameter* zudem parametrisierbar.

6.5 Graphische Templates

Ein zusätzlicher Effekt von SPin ist die Einführung eines *graphischen Templatekonstrukts* (ähnlich dem textuellen Templatekonstrukt in C++). Ein Modellierer kann graphische Templates mit Hilfe der eingeführten Elemente *RefinementAnnotation*, *RefinementLink* und *RefinementParameter* erzeugen und in allen von Fujaba unterstützten Diagrammen anwenden. Dies soll anhand eines Beispiels verdeutlicht werden.

Ziel soll es sein, in einem UML Klassendiagramm UML Klassen zu erzeugen, die von *javax.swing.AbstractAction* erben und deren Implementierung ein bestimmtes Muster aufweist. Das Muster wird in ein parametrisierbares Framework übertragen, aus dem die UML Klasse generiert wird. Diese kann mit Hilfe des Export-Mechanismus von Fujaba als Java-Datei exportiert und die so entstandene Implementierung des Frameworks dann in ein Java-Projekt eingebunden werden.

Eine von *AbstractAction* erbende Klasse kann z.B. von Fujaba oder einem Fujaba Plug-in verwendet werden, um auf Benutzerinteraktion zu reagieren. Drückt ein Benutzer einen Knopf oder selektiert einen Menü-Eintrag[1] wird die Methode 'actionPerformed(ActionEvent):Void' der zugeordneten Aktion aufgerufen. Sie erhält als Parameter ein *ActionEvent*, aus dem die Quelle der Aktion extrahiert werden kann. Eine verschiedene Anzahl von Quellen ist dabei möglich, falls z.B. mehrere graphische Elemente eines Diagramms selektiert sind. Ebenso können die Quellen von verschiedenem Typ sein. Das Framework des graphischen Templates soll es ermöglichen, einen Typ-Filter für die Quelle des *ActionEvent*s anzugeben und auf eine verschiedene Anzahl von Auslösern zu reagieren. Desweiteren soll dem Framework eigener Programmcode zugefügt werden können.

[1] führt also eine Aktion aus

Das Framework benötigt aus diesem Grund die folgenden Parameter:

1. *packageName*,
2. *className*,
3. *typeToProcess*,
4. *storageItemVariableName* und
5. *codeToProcess*.

Die Parameter *packageName* und *className* legen den voll qualifizierten Klassennamen der zu erzeugenden UML Klasse fest. *typeToProcess* gibt den Typ der zu filternden Quellen des *ActionEvent*s an[2] und *storageItemVariableName* definiert den Namen der Variable die an ein Quell-Objekt gebunden werden soll. *codeToProcess* schließlich erlaubt es Code in das Framework einzubinden. Die Programmiersprache des Codes ist abhängig von der Programmiersprache aus der Fujaba in seinem Codegenerierungsprozess lauffähigen Code erzeugen soll. In unserem Beispiel ist dies die Programmiersprache Java. Der durch *codeToProcess* in das Framework eingefügte Code kann über den Wert des Parameters *storageItemVariableName* auf die Quell-Objekte von *ActionEvent* zugreifen und auf ihnen operieren. Aus Gründen der Einfachheit sollte *codeToProcess* keinen allzu langen Codeabschnitt enthalten. Stattdessen ist der Aufruf einer Funktion, die den eigentlich abzuarbeitenden Code enthält, zu empfehlen.

Mit Hilfe von SPin wird eine neue Transformationsregel vom Typ *RefinementRule* generiert, die den Namen *RRCreateAction* trägt. Die Methode 'apply(ASGElement):Boolean' der Klasse *RRCreateAction*, implementiert das oben beschriebene Framework. Das in Abbildung 6.1 dargestellte *Fujaba-StoryDiagram* ist die entsprechende Implementierung. Sie besteht aus vier Teilen:

Mustererkennung In einem UML Klassendiagramm wird nach einer Annotierung gesucht, die den Namen *CreateAction* trägt und die oben beschriebenen fünf Parameter enthält.

Generieren der Infrastruktur Eine UML Klasse wird erzeugt und dem aktuellen UML Klassendiagram hinzugefügt. Sie erhält den durch die Parameter *packageName* und *className* festgelegten Namen, erbt von *javax.swing. AbstractAction* und implementiert die Methode 'actionPerformed(ActionEvent):Void'.

Implementierungs-Template Dieser Teil erzeugt einen String, der in ein Implementierungsmuster variable Elemente einfügt (die Werte der Parameter *typeToProcess*, *storageItemVariableName* und *codeToProcess*).

[2] also der Quell-Objekte, die bearbeitet werden sollen

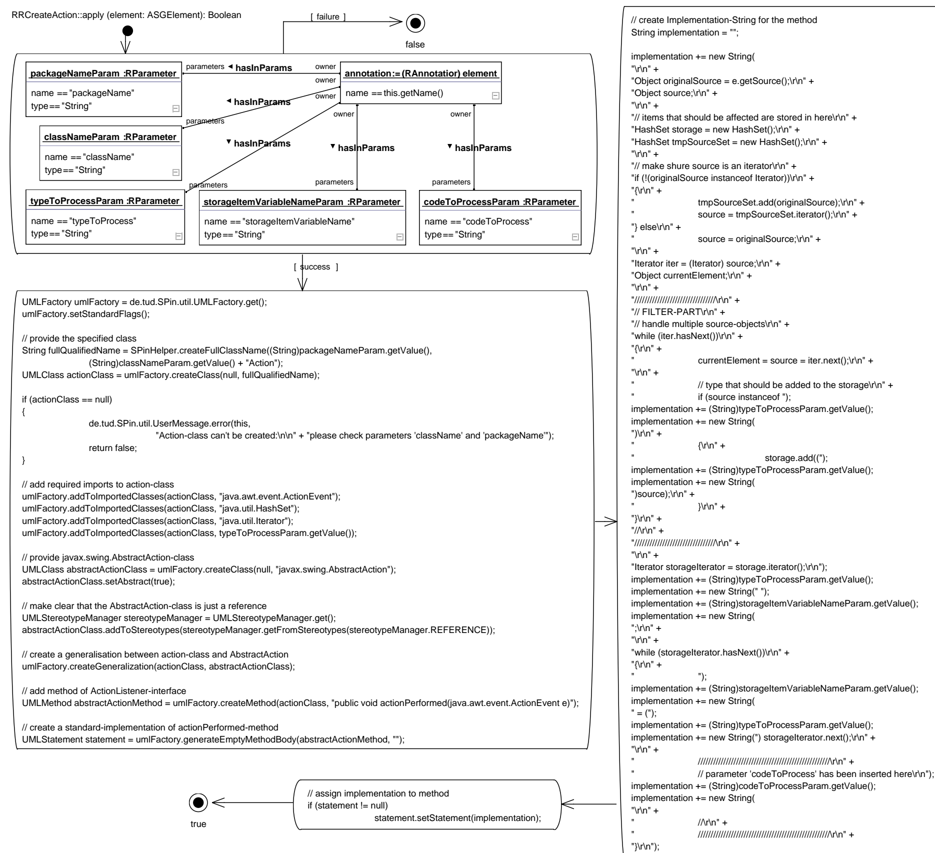


Abbildung 6.1: *Implementierung des Frameworks, das eine UML Klasse generiert, die von AbstractAction erbt. Dieser Klasse wird eine Methode zufügt, deren Implementierung ein bestimmtes Muster aufweist, das parametrisiert werden kann.*

Zuweisung des Implementierungs-Template an die Methode 'actionPerformed(ActionEvent):Void'.

Es ist zu beachten, dass die Annotierung, die im Transformationsprozess genutzt wird, am Ende nicht zerstört wird. Dies ist praktisch, wenn man mehrere Aktionen erstellen will. In diesem Fall muss nicht für jede Aktion eine eigene Annotation erzeugt und dieser die benötigten Parameter zugewiesen werden. Auf diese Weise kann man eine Annotation mehrfach transformieren. Vor dem nächsten Transformationsprozess sind nur die Werte der Parameter abzuändern.

Nachdem die Regel implementiert und der Transformationsbibliothek hinzugefügt worden ist, kann sie zum Einsatz kommen.

Daraufhin kann man in einem UML Klassendiagramm eine Annotierung mit Na-

men *CreateAction* erzeugen (siehe Abbildung 6.2).

Abbildung 6.2: *Repräsentation des graphischen Templates*

Nun werden dieser Annotierung die benötigten Parameter hinzugefügt (siehe Abbildung 6.3).

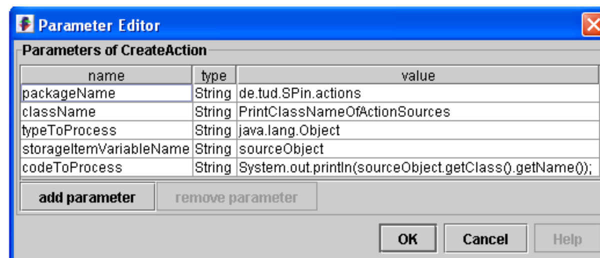


Abbildung 6.3: *Parametrisierung des graphischen Templates aus Abbildung 6.2*

Die Annotierung (das graphische Template) kann jetzt mit dem SPin-Transformator *TransformatorRefinement* transformiert werden. Das aus der Annotierung durch den Transformationsvorgang entstehende Konstrukt ist in Abbildung 6.4 dargestellt.

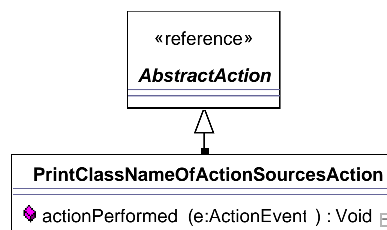


Abbildung 6.4: *Die Annotierung aus Abbildung 6.2 nach Anwenden der Regel aus Abbildung 6.1*

Die Implementierung der Methode 'actionPerformed(ActionEvent):Void' des parametrisierten Frameworks kann Abbildung 6.5 entnommen werden.

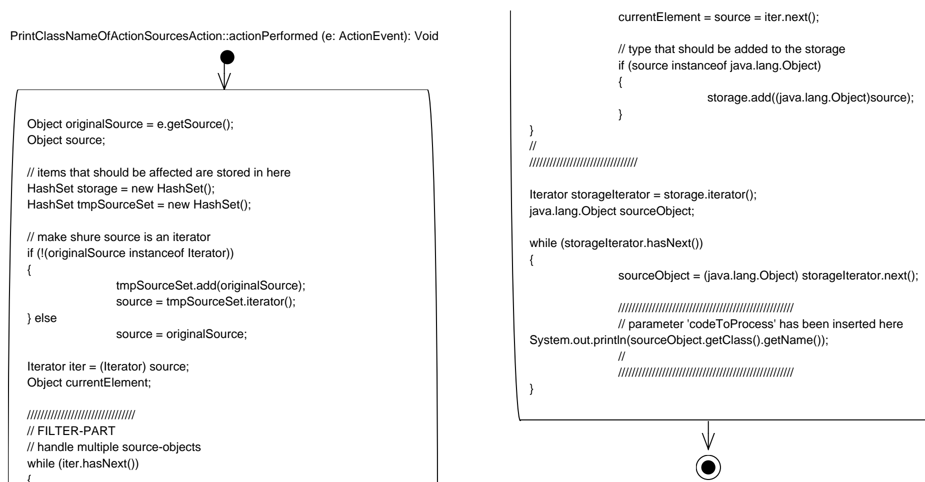


Abbildung 6.5: Generierte Implementierung der Methode 'actionPerformed(ActionEvent):Void'

6.6 Erstellte Transformationsregeln

Im Laufe der Entwicklung von SPin wurden einige Transformationsregeln erstellt, die sowohl für die Weiterentwicklung von SPin genutzt als auch für Wartungszwecke von Fujaba UML Projekten eingesetzt wurden. Diese sollen im Folgenden kurz vorgestellt werden. Die Implementierung der Regeln ist im Anhang dargestellt.

RRCreateAction ist in 6.5 ausführlich beschrieben.

RRDeleteMethods, siehe B.1
Löscht nach Rückfrage sämtliche Methoden einer UML Klasse inklusive deren Implementierung.

RRRepairUMLBaseTypes, siehe B.2
Diese Regel repariert einige der UML Basisdatentypen in Fujaba. Beispielsweise wird der Basisdatentyp *String* in der derzeitigen Implementierung von Fujaba (Version 4.3.1) überschrieben, falls eine Klasse mit Namen *String* in einem UML Klassendiagramm erzeugt wird. Dies hat schwerwiegende Folgen für das *Story Driven Modeling*, da ab sofort Attribute vom Datentyp *String* eines UMLObjects im SDM nicht mehr zur Attribut-Zuweisung oder -Assertion genutzt werden können.

RRCleanTypeList, siehe B.3
Räumt die Typen-Liste des aktuellen UML Projektes auf. Nützlich, falls beim Bearbeiten von UML Diagrammen viele UML Klassen erzeugt und wieder gelöscht wurden, weil diese nicht automatisch aus der Typen-Liste entfernt werden.

RRDetectAttributes und ARDetectAttributes, siehe B.4 und B.5 Zwei Regeln, die zeigen, dass man das gleiche Ergebnis auch mit unterschiedlichen Regeltypen erreichen kann. Beide Regeln abstrahieren is/set- und get/set-Methoden einer Klasse, die einer Namenskonvention folgen, zu einem Attribut der Klasse. Dieses Attribut kann im SDM genutzt werden.

Die AbstractionRule wird dabei so eingesetzt, dass sie alle Klassen des aktuellen Klassendiagramms abstrahiert. Die RefinementRule hingegen abstrahiert nur eine ausgewählte Klasse.

Verfeinerungsregeln können also dazu „missbraucht" werden, Strukturen zu abstrahieren. Dies wiederspricht der ursprünglichen Intention einer Verfeinerungsregel einen Sachverhalt zu konkretisieren.

RRMethodModification, siehe B.6
Führt eine einfache Textersetzung in Methoden durch. Verwendet dazu den in SPin enthaltenen *Methoden-Modifizierer* (siehe 5.2.4).

RRMethodNameModification, siehe B.7
Diese Regel ändert den Namen jeder Methode, die über einen Link „modify" an die Annotation gebunden ist. Der Link muss einen Parameter vom Typ *String* mit Namen „newName" aufweisen. Der Wert dieses Parameters wird als neuer Methodenname verwendet. Der alte Methodenname wird im Parameter gespeichert, so dass Änderungen rückgängig gemacht werden können.

RRSingleton, siehe B.8
Diese Regel fügt einer UML Klasse eine Implementierung des *Singleton-Pattern* hinzu. Durch dieses wird abgesichert, dass eine Klasse genau ein Exemplar besitzt und einen globalen Zugriffspunkt bietet. Aus dem entstehenden Modell kann eine lauffähige Java-Datei generiert werden. Eine detaillierte Beschreibung des Singleton-Pattern kann [11] entnommen werden.

In dieser Regel fehlt derzeit die Implementierung eines Mechanismus, der in sämtlichen Klassen, die die Singleton-Klasse verwenden, alle Erzeugungsaufrufe der Singleton-Klasse modifiziert. Erzeugungsaufrufe müssen durch den Aufruf der globalen Methode 'get():<Singleton-Klassenname>' ersetzt werden.

RRVisitor, siehe B.9
Die Regel fügt einem UML Klassendiagramm ein neues Interface hinzu, das für eine mit „element“ markierte UML Klasse und deren Kind-Klassen jeweils eine visit-Methode bereitstellt. Die „element“-Klasse und deren Kind-Klassen erhalten eine neue Methode „accept“. Diese wird so implementiert, dass sie die für sie zuständige „visit“-Methode des Besuchers aufruft. Optional können konkrete Besucher angegeben werden, die das generierte Interface implementieren sollen und denen die zu implementierenden Methoden des Interface hinzugefügt werden. Die Implementierung der Methoden muss der Modellierer von Hand erledigen. Durch das Visitor-Pattern wird es möglich, neue Operationen zu definieren, ohne die Klassen der zu bearbeitenden Elemente zu ändern. Eine detaillierte Beschreibung des Visitor-Pattern kann [11] entnommen werden.

RRObserver, siehe B.10
Diese Regel fügt einem UML Klassendiagramm die Basisklassen des Observer-Patterns hinzu. Die Namen dieser Klassen können durch Angabe der optionalen Parameter „subjectClassName“ und „observerClassName“ geändert werden. Eine der Regel zugehörige Annotation muss zwei Links enthalten. Diese binden die Klasse, die als konkreter Beobachter fungieren und das Attribut, dessen Zustand beobachtet werden soll in das Observer-Pattern ein. Das konkrete Subjekt ist die Klasse, in der das Attribut enthalten ist und muss nicht explizit angegeben werden. Ändert sich der Zustand einer Instanz des konkreten Subjekts, werden alle abhängigen Instanzen des konkreten Beobachters benachrichtigt und automatisch aktualisiert. Aus dem entstehenden Modell können lauffähige Java-Dateien generiert werden. Der Benachrichtungsaufruf 'notify():Void' muss von Hand ergänzt werden. Ebenso müssen konkrete Observer mittels der Methoden „attach“ und „detach“ bei einem konkreten Subjekt registriert bzw. abgemeldet werden.

Diese Implementierung ist eine Variante des Observer-Patterns, die es einem Observer ermöglicht mehr als ein Subjekt zu beobachten. Die Methode 'notifyObservers():Void' ist nicht in der abstrakten Subjekt-Klasse implementiert, sondern im konkreten Subjekt. Dies macht es möglich die Methode mit einer verfeinerten Implementierung zu versehen, die die 'update'-Methode nur auf den notwendigen Observern aufruft. Eine detaillierte Beschreibung des Observer-Pattern kann [11] entnommen werden.

RRLogged, siehe B.11
Diese Regel führt ein *interaction refinement* durch (siehe Kapitel 2). Ist eine Annotation Namens „Logged“ an eine UML Assoziation (*target*) gebunden, wird diese verfeinert. Optional kann eine Klasse angegeben werden, die als „Logger“ verwendet werden soll. Ansonsten wird eine neue Klasse *Logger* generiert. Anschließend werden Referenzen von an *target* gebundene Klassen auf die Logger-Klasse erzeugt. Dies geschieht jedoch nur, wenn eine solche Klasse von ihrer Partner-Klasse

nicht als Referenz genutzt wird[3]. Es wird außerdem keine zweite Referenz eingeführt, falls die Assoziationsenden auf die gleiche Klasse zeigen. Man beachte, dass ein generierter *Logger* als Singleton implementiert wird. Kenntlich gemacht wird dies durch Hinzufügen einer neuen Annotation „Singleton“, die an die Klasse *Logger* gebunden wird. Der Verfeinerungs-Transformator *TransformatorRefinement* wird anschließend dazu verwendet, diese Annotation automatisch zu transformieren[4]. Der Regelentwickler der Regel *RRLogged* nutzt so das „wohlbekannte“ Muster *Singleton* und muss es nicht mehr per Hand implementieren.

Mangels eines Quellcode-Modifizierers wurde das Generieren von Code in den Klassen, die an *target* gebunden sind, in dieser Regel nicht implementiert. Ein log-Aufruf müsste an jeder Stelle der Implementierung einer dieser Klassen erfolgen, an der mit der Partner-Klasse „gesprochen“ wird[5].

RRJavaApplication, siehe B.12
Diese Regel fügt einer an die Annotation gebundenen Klasse die Java-Einsprungsfunktion 'public static void main(String[] args)' hinzu.

Ein Beispiel für die Anwendung der Regeln RRSingleton, RRObserver, RRVisitor, RRLogged und RRJavaApplication findet sich in Kapitel 7.

---

[3] was bedeuten würde, dass die referenzierte Klasse nicht mit ihrer Partner-Klasse „spricht“

[4] dies funktioniert nur, wenn die korrespondierende Regel *RRSingleton* in der Regel-Bibliothek existiert

[5] „gesprochen“ wird, wenn Funktionen auf einem Attribut aufgerufen werden, das den Namen der Rolle der Partner-Klasse trägt

Kapitel 7

Fallbeispiel

In diesem Kapitel soll die Anwendung von SPin anhand eines Fallbeispiels verdeutlicht werden. Die Architektur des Beispiels ist in Abbildung 7.1 dargestellt. Diese zeigt das System auf der höchsten Abstraktionsebene.

Im angeführten Beispiel wird die Qualitätskontrolle einer Fabrik simuliert, die als Java-Anwendung ausgeführt werden kann. In der Simulation sind die Entwurfsmuster *Singleton*, *Observer* und *Visitor*, sowie der Aspekt *Logged* umgesetzt. Deren Realisierung wurde in 6.6 bereits vorgestellt.

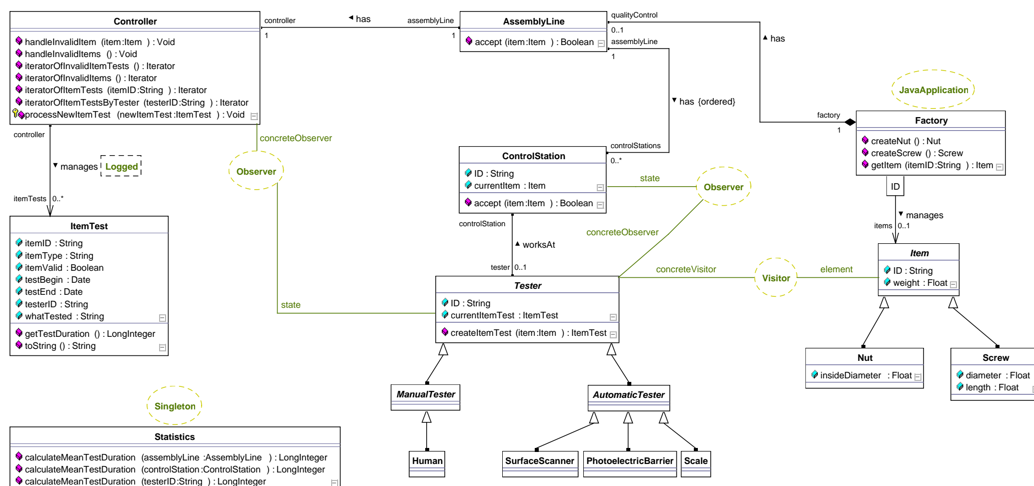


Abbildung 7.1: *Stratifikations-Beispiel: abstrakteste Ebene des Simulators*

Zunächst wird für das System eine Anforderungsanalyse erstellt. Anschließend wird es in der Sprache UML modelliert, mit Hilfe von SPin annotiert und daraufhin mit Hilfe der in 6.6 vorgestellten Regeln transformiert. Abschließend werden die

eigens spezifizierten Strukturen und sofern erforderlich die generierten Methoden in der Sprache „Java“ implementiert.

Anforderungsanalyse

Eine Fabrik (Klasse *Factory*) soll in der Lage sein Gegenstände (abstrakte Klasse *Item*) zu produzieren. Solange ein produzierter Gegenstand die Fabrik noch nicht verlassen hat, wird er von der Fabrik verwaltet. In diesem Beispiel handelt es sich um eine metallverarbeitende Fabrik, die Schrauben und Schraubenmuttern herstellt (Klassen *Screw* und *Nut*). Jeder Gegenstand hat eine eindeutige Kennung, über die man ihn identifizieren kann und besitzt bestimmte Kenngrößen, die in der Qualitätskontrolle geprüft werden sollen. Standardmäßig besitzt jeder Gegenstand ein bestimmtes Gewicht. Schrauben haben zusätzlich die Kenngrößen Durchmesser und Länge, Schraubenmuttern einen Innendurchmesser.

An eine Fabrik ist eine Qualitätskontrolle angeschlossen, die dafür zuständig ist fehlerhafte Produkte zu erkennen und diese auszusortieren. Realisiert wird die Qualitätskontrolle durch ein Fließband (Klasse *AssemblyLine*), das aus einer variablen Anzahl von Kontrollstationen (Klasse *ControlStation*) besteht. Diese können als Segmente des Fließbandes aufgefasst werden, an denen Tester „vorbeifahrende“ Gegenstände einer Qualitätskontrolle unterziehen. Ein Gegenstand durchläuft die Kontrollstationen in einer bestimmten Reihenfolge. Diese hängt von der Anordnung der Kontrollstationen am Fließband ab. Während der Simulation kann die Anordnung geändert werden um etwa zu prüfen, ob eine andere Anordnung effizienter sein kann. Die Tester können sowohl manuelle Tester als auch automatische Tester sein. In diesem Beispiel fallen Menschen (Klasse *Human*) in die Kategorie „manuelle Tester“. Konkrete Ausprägungen von automatischen Testern sind:

SurfaceScanner Eine Einheit, die in der Lage ist die Oberfläche von Gegenständen auf richtige Beschaffenheit zu testen. Eine Schraube beispielsweise würde daraufhin getestet werden, ob ihr Gewinde fehlerfrei ist.

PhotoelectricBarrier Ein Gerät mit einer Lichtschranke, das in der Lage ist zu bestimmen, ob ein Gegenstand die geforderte Länge aufweist.

Scale Ein Gerät mit einer Waage, das ermittelt, ob das Gewicht des Gegenstandes mit einem Sollwert übereinstimmt.

Ein Tester wird von seiner Kontrollstation benachrichtigt, sobald ein zu testender Gegenstand vorhanden ist. In diesem Falle beginnt der Tester mit der Überprüfung des Gegenstandes. Nach abgeschlossener Prüfung erstellt er einen Testbericht (Klasse *ItemTest*). Dieser enthält Informationen darüber, wann der Test begonnen und wann er abgeschlossen wurde, die ID des getesteten Gegenstandes, den

Typ des Gegenstandes, die ID des Testers, eine Beschreibung dessen, was getestet wurde und die Aussage, ob der getestete Gegenstand die Prüfung bestanden hat.

Das Fließband hat neben seinen Kontrollstationen eine Kontrolleinheit (Klasse *Controller*), die darauf wartet, dass die Tester neue Testberichte fertigstellen. Diese werden von der Kontrolleinheit verwaltet und ausgewertet. Zusätzlich wird der Zugriff auf die Testberichte dokumentiert. Desweiteren ist die Kontrollstation dazu bevollmächtigt, über die Behandlung von ungültigen Gegenständen zu entscheiden. Typischerweise werden solche Gegenstände recycled oder repariert. Auf diesen Prozess soll im Beispiel jedoch nicht eingegangen werden. Hier werden ungültige Gegenstände zerstört.

Die Klasse *Statistics* hat statistische Werte zu berechnen. Von ihr soll es nur eine Instanz geben, die in der Lage ist die durchschnittliche Dauer eines Tests zu berechnen, indem sie die in der Kontrolleinheit gespeicherten Testberichte auswertet.

Annotierung

Nachdem die Anforderungen an das System gestellt sind, wird das Modell in der Sprache UML realisiert. Anschließend werden Annotationen hinzugefügt, die anzeigen, welche Muster und Aspekte im Modell vorkommen. Das resultierende Diagramm ist in Abbildung 7.1 dargestellt. Deutlich zu erkennen sind die verwendeten Muster *Singleton*, *Observer*, *Visitor* und *JavaApplication* sowie der Aspekt *Logged*.

Die Annotation *JavaApplication* fügt der Klasse *Factory* eine Einsprungsfunktion für die Programmiersprache Java hinzu, damit das System als Simulation gestartet werden kann. Das *Observer-Pattern* kommt zweimal zum Einsatz. Zum einen beobachtet jeder Tester das Erscheinen eines neuen Gegenstandes in seiner Kontrollstation, zum anderen beobachtet die Kontrolleinheit des Fließbandes die Fertigstellung jedes Testberichts, der von einem Tester angefertigt wird. Zwischen Gegenständen und den konkreten Ausprägungen eines Testers kommt das *Visitor-Pattern* zum Einsatz. Jeder Tester soll in der Lage sein, jeden von der Fabrik produzierten Gegenstand zu kontrollieren und einen Testbericht darüber zu verfassen. Programmiertechnisch gesprochen „besucht“ der Tester den Gegenstand und kann ihn auf diese Weise einer Kontrolle unterziehen. Als Element des Pattern fungiert die Klasse *Item* und als Visitor die abstrakte Klasse *Tester*.

Wie in der Anforderungsanalyse bereits erwähnt, soll der Zugriff auf Testberichte dokumentiert werden. Für die Verwaltung von Testberichten ist die Klasse *Controller* zuständig. Sie greift auf diese über die Assoziation *manages* zu. Hier bietet es sich an, der Assoziation *manages* den Aspekt *Logged* zuzuweisen. Auf diese Weise wird jeder Zugriff der Kontrolleinheit auf Testberichte dokumentiert. Abschließend wird der Klasse *Statistics* das Enwurfsmuster *Singleton* zugewiesen, weil von ihr nur eine Instanz existieren soll.

Die Bindung der Annotationen *JavaApplication*, *Singleton* und *Logged* an die entsprechenden Diagrammelemente, ist in Abbildung 7.1 nicht explizit visualisiert, sondern wird durch die Nähe zum Element deutlich gemacht. *JavaApplication* ist an die Klasse *Factory*, *Singleton* an Klasse *Statistics* und *Logged* an die Assoziation *manages* gebunden. Die Parametrisierung von Annotationen durch Links kann Tabelle 7.1 entnommen werden.

Annotation	Link		
Name	Rollenname	verbindet mit Element	ElementTyp
Logged, JavaApplication, Singleton	- kein Link -		
Observer	concreteObserver state	Controller currentItemTest:ItemTest	UMLClass UMLAttr
Observer	concreteObserver state	Tester currentItem:Item	UMLClass UMLAttr
Visitor	element concreteVisitor	Item Tester	UMLClass UMLClass

Tabelle 7.1: *An Annotationen vorhandene Links*

Die beiden Observer-Annotationen besitzen je einen weiteren Parameter namens „observerClassName“. Dieser bewirkt, dass bei Auflösung der Annotation ein Observer-Interface mit bestimmtem Namen generiert wird. So erhält man differenziertere Observer-Typen die nur die nötigen 'update'-Methoden implementieren. Für den Observer *Controller* lautet der Wert des Parameters „TesterObserver“, für den *Tester* „ControlStationObserver“.

Auflösen von Annotationen

Nachdem das System nun auf der abstraktesten Ebene fertig modelliert ist, wird die Annotation *JavaApplication* verfeinert. Das Ergebnis dieses Vorgangs ist in Abbildung 7.2 dargestellt: der Klasse *Factory* wurde die Methode 'public static void main(String[] args)' hinzugefügt.

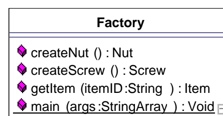


Abbildung 7.2: *Stratifikations-Beispiel: Änderungen nach Verfeinern der Annotation „JavaApplication“*

Jetzt werden die beiden Annotation *Observer* aufgelöst. Die Änderungen der be-

teiligten Klassen, die neu hinzugefügten Interfaces *ControlStationObserver* und *TesterObserver*, sowie die Klasse *Subject* sind in Abbildung 7.3 dargestellt. Es ist anzumerken, dass die Methode 'setTester(Tester):Boolean' in der Klasse *ControlStation* im Gegensatz zu anderen get/set-Methoden von Hand zu implementieren ist, da in ihr die 'attach'- und 'detach'-Methoden des Observer-Patterns aufgerufen werden sollen.

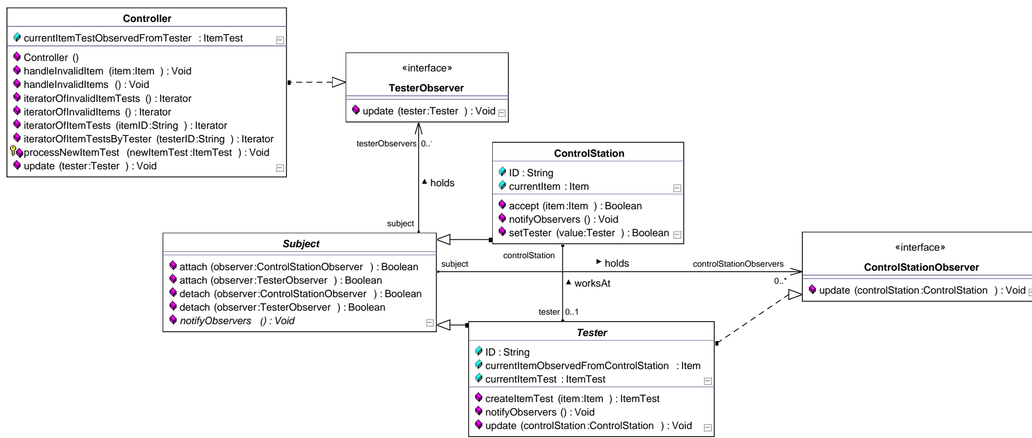


Abbildung 7.3: *Stratifikations-Beispiel: Änderungen nach Verfeinern der Annotation „Observer“*

Nun wird die Annotation *Visitor* verfeinert (siehe Abbildung 7.4). Es wird ein neues Interface *ItemVisitor* generiert, dem automatisch Methodendeklarationen für alle Item-Klassen[1] hinzugefügt werden. Die Methoden des Interfaces müssen anschließend per Hand den konkreten *Tester*-Klassen hinzugefügt[2] und implementiert werden. Sämtliche *Item*-Klassen erhalten die Methode 'accept(ItemVisitor): Void', die dem Visitor-Pattern entsprechend implementiert ist.

Nach Verfeinerung der Annotation *Singleton* (Abbildung 7.5) müssen Codestellen, an denen Instanzen von *Statistics* erzeugt wurden, an die Singleton-Implementierung angepasst werden.

Jetzt wird die noch ausstehende Logik der Simulation mit Hilfe von Fujaba implementiert. Dies sind die in Abbildung 7.1 spezifizierten Methoden und ein Teil der durch die Auflösung der Annotationen generierten Methoden, die Tabelle 7.2 zu entnehmen sind.

Nun wird der Ablauf der Simulation in der 'main'-Methode implementiert. Zum Simulationsstart wird eine Fabrik und deren Qualititätskontrolle erzeugt. Diese

[1] Klasse *Item* und alle davon erbenden Klassen

[2] bzw. durch einen von SPin zur Verfügung gestellten Automatisierungsprozess (siehe Klasse *SPinHelper*, Methode 'addUnimplementedMethods(UMLClass):Void')

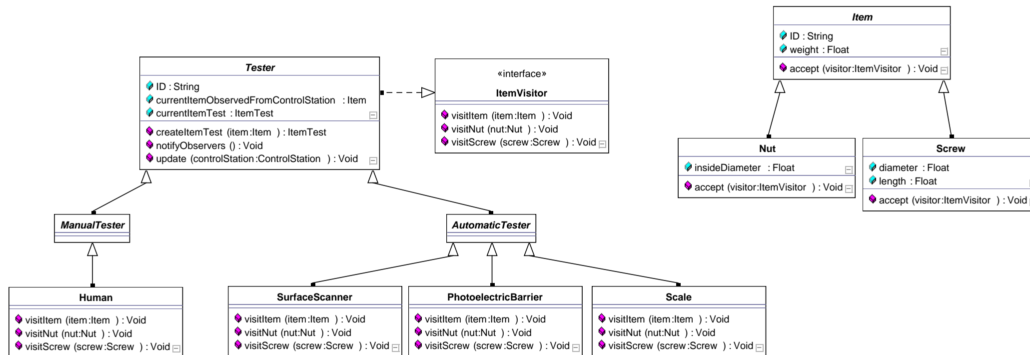


Abbildung 7.4: *Stratifikations-Beispiel: Änderungen nach Verfeinern der Annotation „Visitor“*

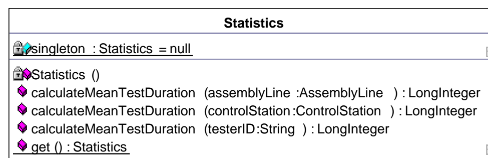


Abbildung 7.5: *Stratifikations-Beispiel: Änderungen nach Verfeinern der Annotation „Singleton“*

besteht aus einer Kontrolleinheit und mehreren Kontrollstationen mit je einem Tester. Die Simulation soll solange laufen, bis eine bestimmte Anzahl nicht defekter Gegenstände von der Fabrik produziert wurde. In jedem Simulationsschritt erzeugt die Fabrik entweder eine Schraubenmutter oder eine Schraube. Mit einer bestimmten Wahrscheinlichkeit ist der erzeugte Gegenstand defekt, es wird also von der Simulationslogik in bestimmten Fällen eine Kenngröße modifiziert. Nun „übergibt“ der Simulator den Gegenstand an die Qualitätskontrolle, die diesen nach dem spezifizierten Verfahren behandelt. Nach Abschluss der Simulation wird von der Klasse *Statistics* die mittlere Testdauer der Qualitätskontrolle und jeder Kontrollstation berechnet und ausgegeben.

Erst nach Abschluss der Implementierung wird die Annotation *Logged* aufgelöst (siehe Abbildung 7.6). Löst man sie früher auf, ergibt sich ein anderes Verhalten, sofern in der späteren Implementierung einer Methode das Rollenattribut *itemTests* verwendet wird.

Das durch diese Vorgänge entstandene UML-Modell ist die niedrigste Abstraktionsebene der Stratifikation, die die konkreteste Sicht auf das Modell der Architektur zeigt, wie die Abbildung 7.7 darstellt. Dieses Modell kann nun mit Hilfe des

Klasse	Methode	Implementieren
Factory	main(String[]):Void	Simulationsablauf
Tester Controller	update(ControlStation):Void update(Tester):Void	Aufruf von 'createItemTest(Item):ItemTest' Behandeln von Testberichten
Implementierungen von Tester	visitNut(Nut):Void visitScrew(Screw):Void	Erstellen eines Testberichts Erstellen eines Testberichts

Tabelle 7.2: *Generierte Methoden, die von Hand implementiert, bzw. modifiziert werden müssen.*

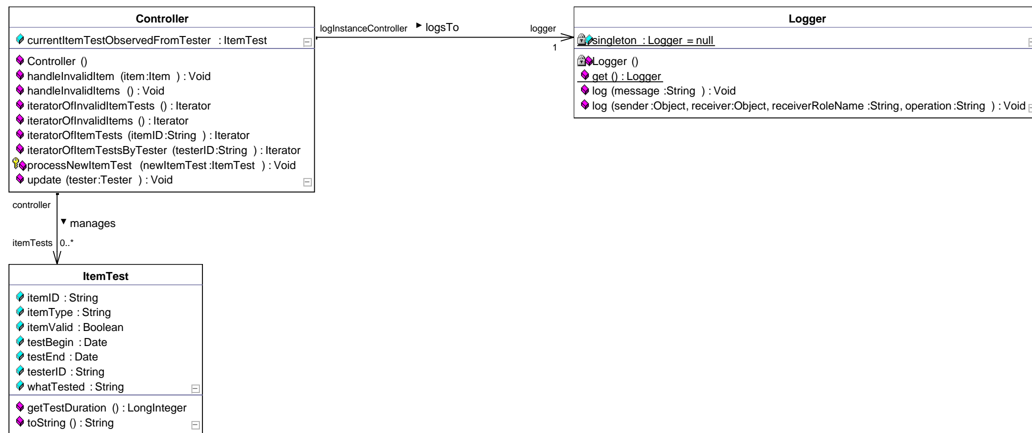


Abbildung 7.6: *Stratifikations-Beispiel: Änderungen nach Verfeinern der Annotation „Logged“*

Fujaba-Codegenerators in eine Java-Implementierung transformiert werden.

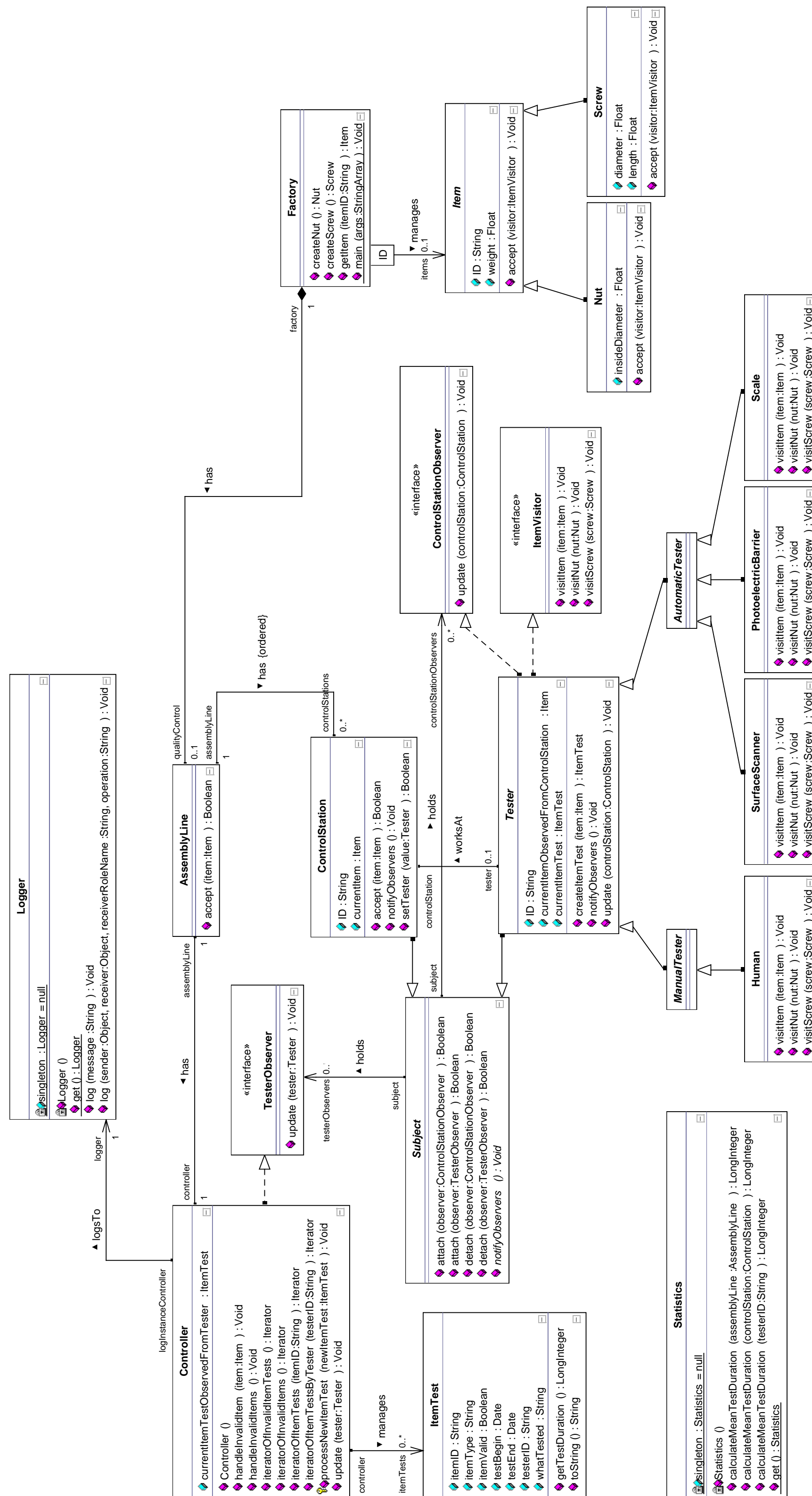


Abbildung 7.7: *Stratifikations-Beispiel: konkreteste Ebene des Simulator-Systems nach Auflösung aller Annotationen*

Kapitel 8

Ausblick

Mit SPin wird Softwareentwicklern die Möglichkeit geboten, ein Modell zu stratifizieren und zwischen verschiedenen Stratifikationsebenen zu wechseln. Stratifizierung ist eine spezielle Form der Modelltransformation, bei der es möglich sein muss, von einem konretisierten Modell zur abstrakteren Variante zurückzukehren und umgekehrt.

In Verfeinerungsregeln wird im Normalfall eine Modell-Modell- und Modell-Code-Transformation durchgeführt. Die Implementierung einer solchen wird mit Unterstützung durch die in SPin und Fujaba integrierten Hilfskonstrukte vereinfacht. Eine Mustererkennung in graphischen Modellen ist mit Hilfe des *Story Driven Modeling* von Fujaba einfach zu implementieren. Bei Transformation unterstützen die Hilfskonstrukte von SPin.

Zu untersuchen bleibt die Erzeugung der umgekehrten Richtung, also der Implementierung von Abstraktionsregeln. Diese sind im Normalfall komplexer als ihre Inverse, weil mehr Details in den Mustererkennungsprozess einbezogen werden müssen. Zudem sind in der Implementierung der Struktur oft textuelle Muster zu suchen, wofür das SDM nicht ausgelegt ist. Im Transformationsteil müssen diese Muster interpretiert und in eine abstrakte Form gebracht werden. Für diesen Fall bietet sich die Verwendung der Funktionalität der beiden Fujaba-Plug-ins *Java-Parser* und *Java-AST* (*Abstract Syntax Tree*) an. Mit deren Hilfe ist es möglich die textuelle Java-Implementierung einer Methode in einen Abstrakten Syntaxbaum zu parsen. Diese syntaktisch aufbereitete Implementierung kann nach Mustervorkommen untersucht und entsprechend modifiziert werden.

Weiterhin müssen Transformationsregeln zur Verfügung gestellt werden, die typische Muster und Aspekte verfeinern und wieder abstrahieren können. Der genannte Aspekt *Logging* ist nur *ein* Beispiel für einen zu generierenden Aspekt. Entwurfsmuster, die in der Softwareentwicklung zum Einsatz kommen, sind dafür prädestiniert in Transformationsregeln umgesetzt zu werden, was am Beispiel der Pattern *Singleton*, *Observer* und *Visitor* gezeigt wurde.

Die von SPin genutzte Fujaba-Version 4.3 wird nach neuesten Angaben die letzte dieser Reihe sein. Für die kommende Version 5 ergeben sich grundlegende Änderungen. Um die Vorteile dieser Version nutzen zu können, muss SPin an diese angepasst werden. In Version 5 wird anstelle von Java 1.4 die Version 1.5 zum Einsatz kommen. Zudem wird es möglich sein in mehreren Projekten zu arbeiten, was die Erstellung von Regeln in einem Projekt und das Testen in anderen Projekten erleichtern wird. Desweiteren wird das ASG Metamodell durch ein anderes Modell ersetzt werden, das aller Voraussicht nach auf MOF (Meta Object Facility) basieren wird. Am Fachgebiet *Echtzeitsysteme* des Fachbereichs Elektrotechnik an der TU Darmstadt wird an einer Implementierung des *MOF2.0 Standards* für Fujaba gearbeitet [19]. Eine Anbindung von SPin an die Schnittstelle des neuen Metamodells wird sich auf die Architektur von SPin auswirken. Bis dahin erzeugte Regeln müssen an die neuen Schnittstellen angepasst werden.

Die Anbindung der Regel-Bibliothek an eine global erreichbare Datenbank, in der Transformationsregeln gespeichert sind, ist eine zusätzliche Erweiterungsmöglichkeit für SPin. Entwickler könnten so neue Regeln erzeugen und anderen Entwicklern zur Verfügung stellen, indem jene der Datenbank hinzufügt werden.

Gewisse Sicherheitsaspekte müssen hierbei jedoch bedacht werden:

- Der Quellcode von Regeln muss zur Einsicht bereitliegen und vor Freigabe durch eine vertrauenswürdige Instanz geprüft werden. Damit ist gewährleistet, dass kein unerwünschter Code bei Anwendung einer Regel ausgeführt wird.
- Regeln müssen vom Regelerzeuger signiert werden, damit nachvollziehbar ist, wer die Regel erzeugt hat.
- Die Integrität von Regeln muss gewährleistet sein. Die von der Datenbank übertragene Regel darf deshalb während des Übertragungsprozesses nicht verändert werden können.

Günstig wäre es, wenn aus einer Verfeinerungsregel eine Abstraktionsregel generierbar ist, da Verfeinerungsregeln im Allgemeinen einfacher zu erstellen sind, als Abstraktionsregeln. In der Praxis scheint dieser Vorgang jedoch wenig praktikabel. Viel aussichtsreicher ist es bi-direktionale Transformationen zu formulieren, wie dies mit den in 4.4 angesprochenen *Triple Graph Grammar*s möglich ist.

Weiterhin muss sowohl für die Navigation zwischen Abstraktionsebenen als auch für die Automatisierung von Regelanwendungen ein Mechanismus entwickelt werden, der feststellt, in welcher Reihenfolge Regeln ausgeführt werden sollen. Dies hängt im wesentlichen davon ab, ob Regeln den Mustererkennungsteil anderer Regeln beeinflussen.

Um SPin bei der Entwicklung eines stratifizierten Softwaresystems einsetzen zu können, müssen mehr Regeln zur Verfügung gestellt werden. Diese müssen das

Modell des Systems verfeinern und abstrahieren können. Typische Muster, die in Regeln codierbar sind, können durch Analyse bestehender Systeme gefunden werden. Zudem entstehen Muster bei der Entwicklung von neuen Systemen. Jene können analysiert, in neue Regeln gefasst und bei Bedarf angewendet werden. Weiterhin sollte ein größeres System testweise mit Unterstützung von SPin entwickelt werden, um herauszufinden, inwieweit die Entwicklung eines größeren Softwaresystems mittels Stratifikation möglich ist.

Literaturverzeichnis

[1] *The eclipse modeling framework (emf) overview*, 06 2004. http://download.eclipse.org/tools/emf/scripts/docs.php?doc=references/overview/EMF.html. 11

[2] C. Atkinson and T. Kühne, *Separation of Concerns through Stratified Architectures*, tech. report, International Workshop on Separations of Concerns and Aspect Oriented Programming, Cannes, France, June 2000. 4

[3] ——, *Stratified Component Frameworks*, Informatica, 25 (2001), pp. 393–401. 2, 5, 6, 20

[4] ——, *Aspect-Oriented Development with Stratified Frameworks*, IEEE Software, 20 (2003), pp. 81–89. 6

[5] C. Atkinson, T. Kühne, and C. Bunse, *Dimensions of Component-based Development*, tech. report, 4th International Workshop on Component-Oriented Programming (WCOP'99; in conjunction with ECOOP'99), Lisbon, Portugal, June 1999. 4

[6] Borland, *User Guide for Together® ControlCenter and Together Solo – Together Open API*, 6.1 ed., May 2003, ch. 45, pp. 755–774. 13

[7] Code Generation Network, *EMF interview with David Steinberg*. Internet. http://www.codegeneration.net/tiki-read_article.php?articleId=38. 11

[8] T. Erler and M. Ricken, *UML2 - GE-PACKT*, vol. 1, mitp-Verlag, 2005. 3

[9] M. Falkhausen, *Effektive Klassendiagramme für Java*. Internet. http://www.falkhausen.de/de/article/article.html. 2

[10] 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. 16, 18

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Entwurfsmuster*, Addison-Wesley, 1996. 4, 54, 55

[12] M. Jeckle. Internet. www.jeckle.de/umltools.html. 12

[13] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, *Aspect-Oriented Programming*, in Proceedings European Conference on Object-Oriented Programming, M. Akşit and S. Matsuoka, eds., Springer-Verlag, Berlin, Heidelberg, and New York, 1997, pp. 220–242. citeseer.ist.psu.edu/kiczales97aspectoriented.html. 6

[14] F. Klar, *Spin – homepage*. Internet. www.klarentwickelt.de/SPin/. 3, 39, 44, 45

[15] T. Kühne, *Automatisierte Softwareentwicklung mit Modellcompilern*, thema Forschung, 1/2003 (2003), pp. 116–122. 6

[16] oose.de GmbH. Internet. http://www.oose.de. 3, 12

[17] J. D. Poole, *Model-driven architecture: Vision, standards and emerging technologies*, (2001). Workshop on Metamodeling and Adaptive Object Models. 9

[18] A. Schürr, *Specification of graph translators with triple graph grammars*, Germany, June 1994, Herrschin, Springer Verlag. Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science. 19

[19] TUD - FB18 - Fachgebiet Echtzeitsysteme, *Mof2.0 plugin für fujaba*. Internet. http://gforge.es.e-technik.tu-darmstadt.de/projects/mofedit/. 66

[20] University of Paderborn, Germany, *Fujaba Tool Suite*. Internet. http://www.fujaba.de/. 3, 13, 44

[21] M. Wannemacher, *Das FPGA-Kochbuch*, Thomson Publishing, 1998, ch. 4 – Entwurf von FPGA-Schaltungen, pp. 69–92. 32

Index

Aktivitätsdiagramm, 16
Annotierungs-Metamodell, 21
AOP, 6
ASG, 14

Bibliotheks-Verzeichnis, 26

CASE-Tool, 1
  Fujaba, 3, 13, 14
  ArgoUML, 12
  EclipseUML, 12
  Together, 13
correspondence graph, 19

Ecore, 11
EMF, 11
EMOF, 11

Fabriken, 34
  RefinementFactory, 35
  TransformationRuleFactory, 35
  UMLFactory, 35
Forward Engineering, 12

graphisches Template, 49

Hilfskonstrukte, 34
  Methoden-Modifizierer, 37
  SPinHelper, 39
  Synchronisations-Modul, 36
  UserMessage, 40
hot spot, 6
Hot-Plug-in Modul, 49

Instanz, 8

LHS, 19

MDA, 9

meta, 8
Meta-Metamodell, 9
Metamodell, 8
MOF, 9, 66

OMG, 9

Plug-in, 11

Regel-Bibliothek, 26
Regeln
  Abstraktionsregeln, 28, 29
  Transformationsregeln, 25
  Verfeinerungsregeln, 28, 29
Reverse Engineering, 12
RHS, 19

SDM, 17
SPin, 3, 20
Story Driven Modeling, 13, 17
Storydiagramm, 16
Stratum, 4

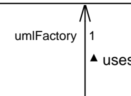
TGG, 19
Transformation, 4

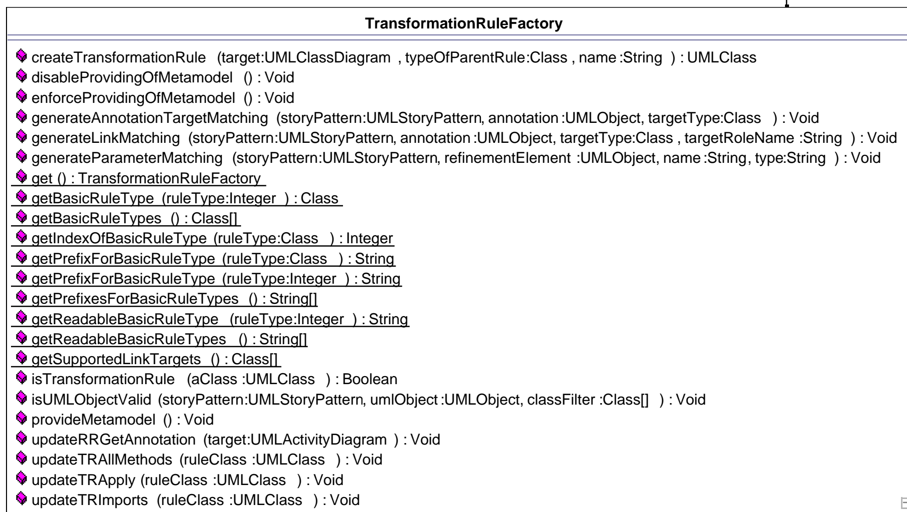
UML, 1
UML Modellierungs Framework, 9
UML2, 11

XMI, 10

Anhang A

Ausführliche Diagramme





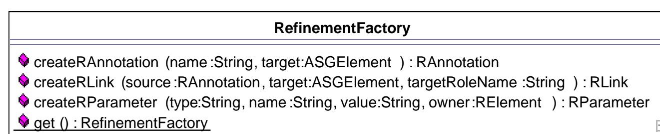


Abbildung A.1: *Fabriken von SPin, die für interne Zwecke genutzt werden und einem Regelentwickler einfache Schnittstellen für die Erzeugung neuer Elemente bieten (volles Detail).*

Anhang B

Transformationsregeln

Eine Beschreibung der hier abgebildeten Regeln kann Kapitel 6.6 entnommen werden.

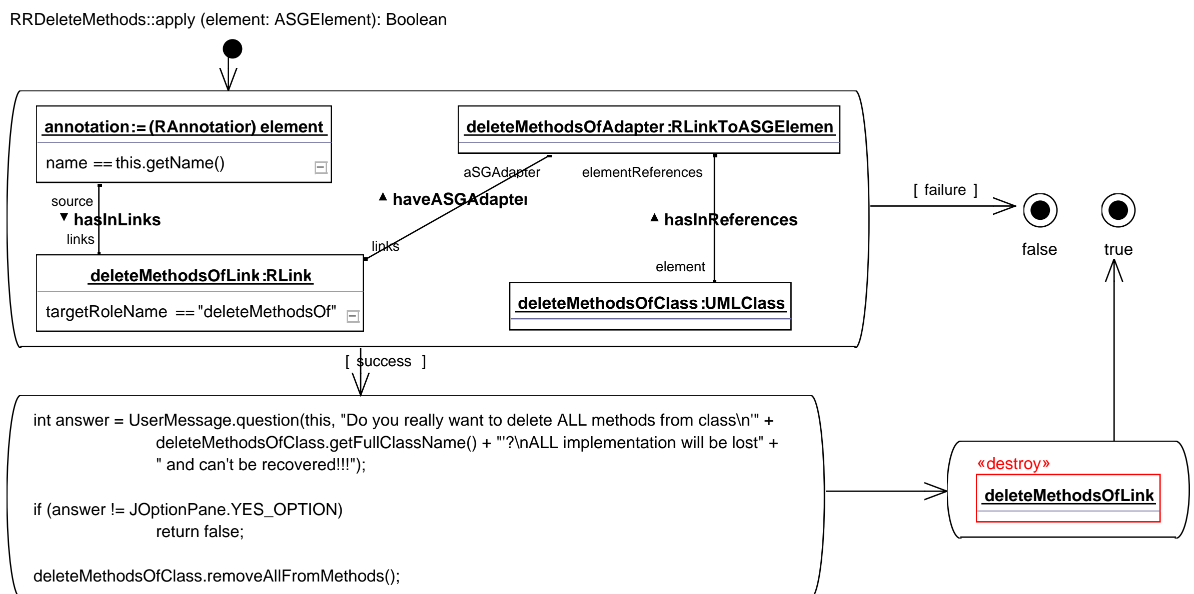


Abbildung B.1: *Eine Regel vom Typ RefinementRule die sämtliche Methoden einer UML Klasse löscht.*

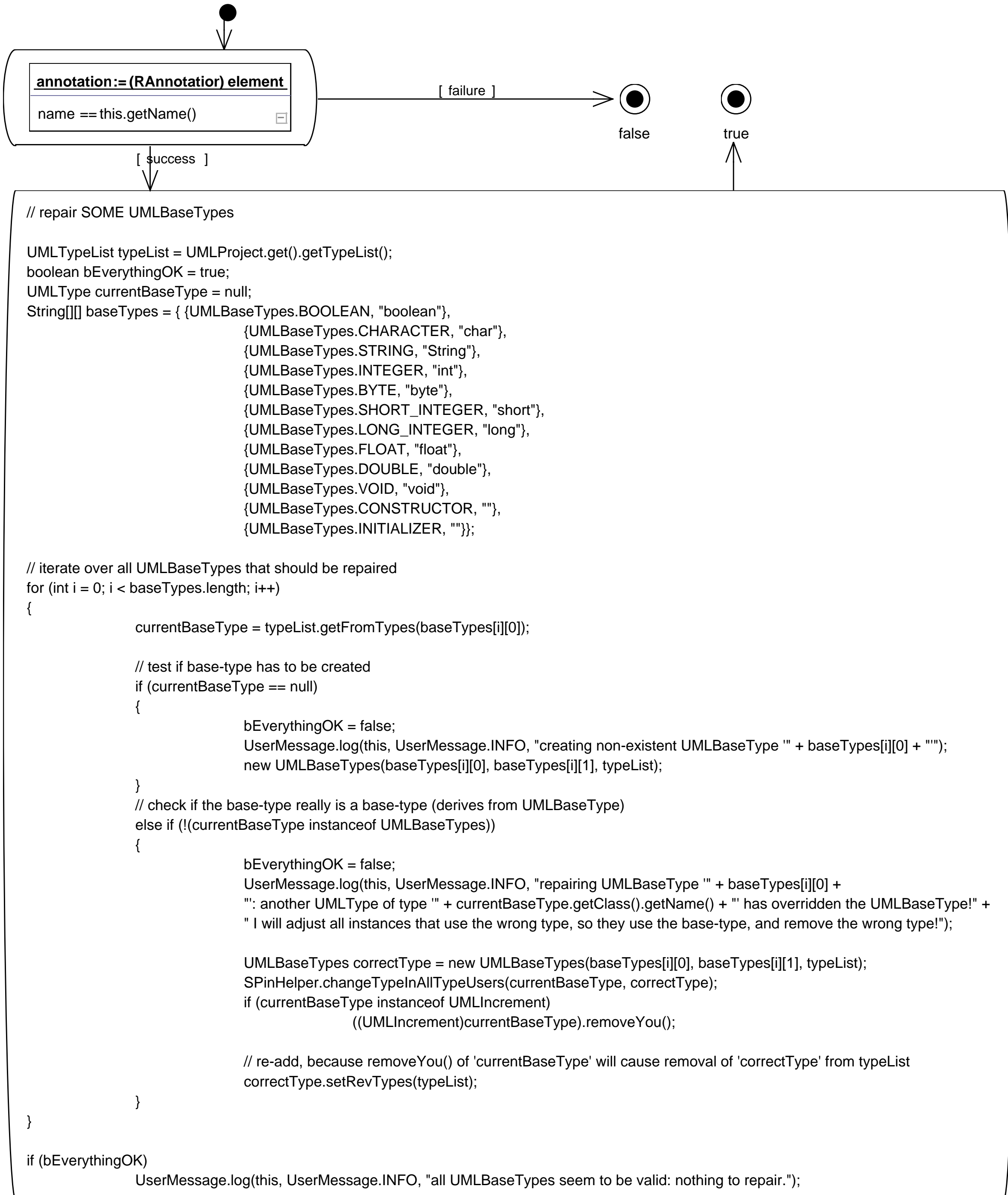


Abbildung B.2: *Eine Regel vom Typ RefinementRule zum Reparieren der UML Basistypen in Fujaba. Nützlich, falls diese korrumpiert wurden.*

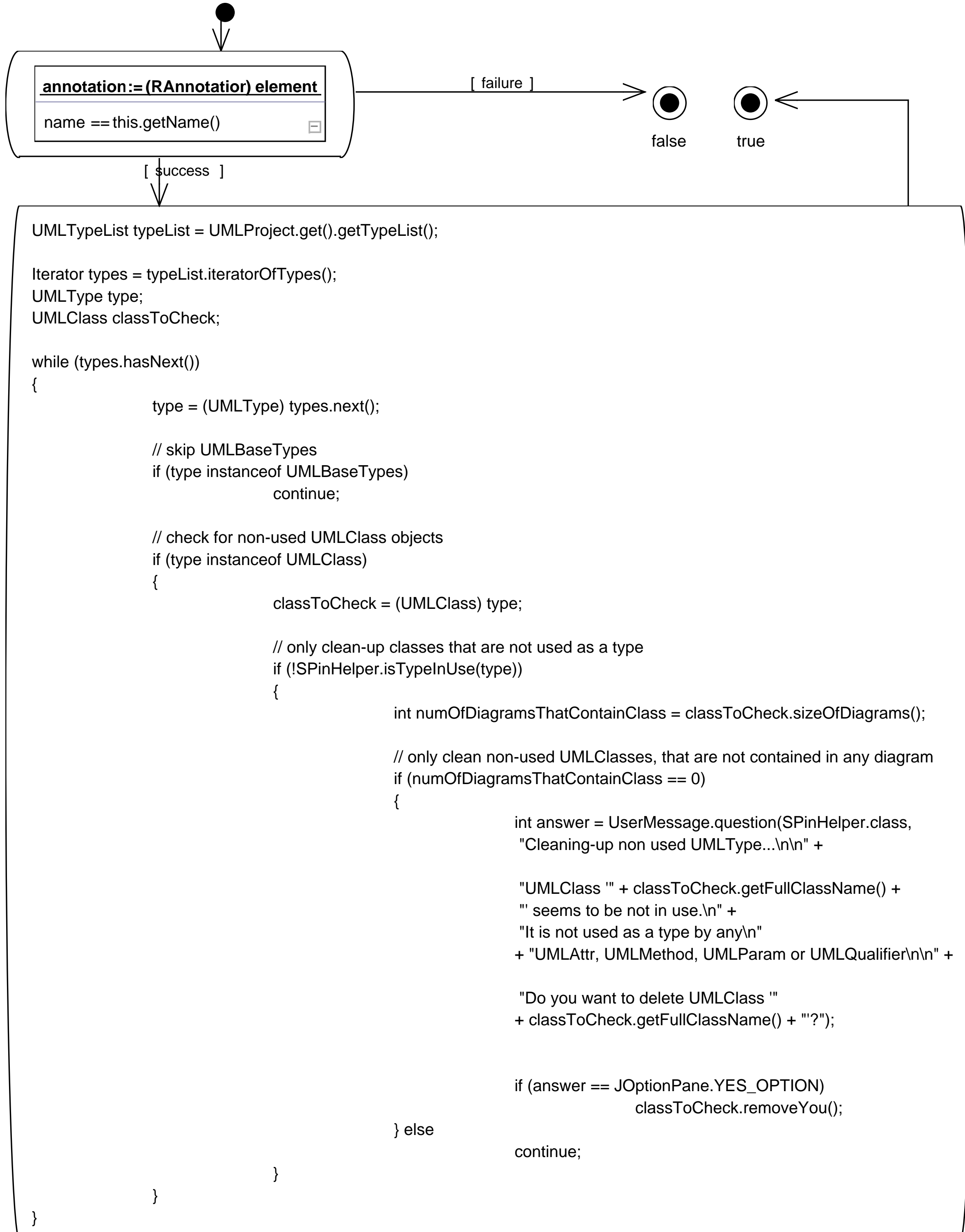


Abbildung B.3: *Eine Regel vom Typ RefinementRule zum Aufräumen der Typen-Liste eines UML Projektes in Fujaba. Nützlich, falls diese Typen enthält, die nicht mehr benötigt werden.*

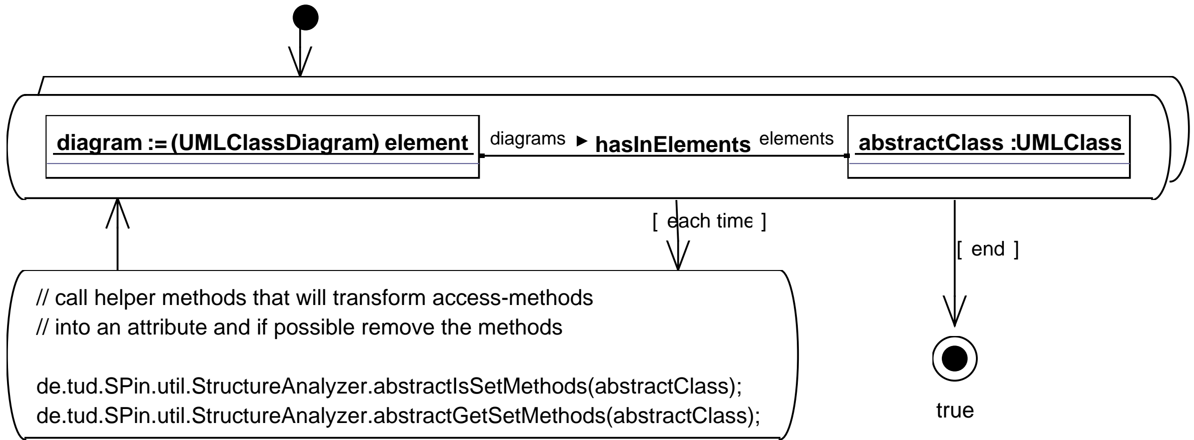


Abbildung B.4: *Eine Regel vom Typ AbstractionRule zum abstrahieren von get/set und is/set Methoden zu Attributen, für alle Klassen eines Klassendiagramms. Nützlich für das Story Driven Modeling.*

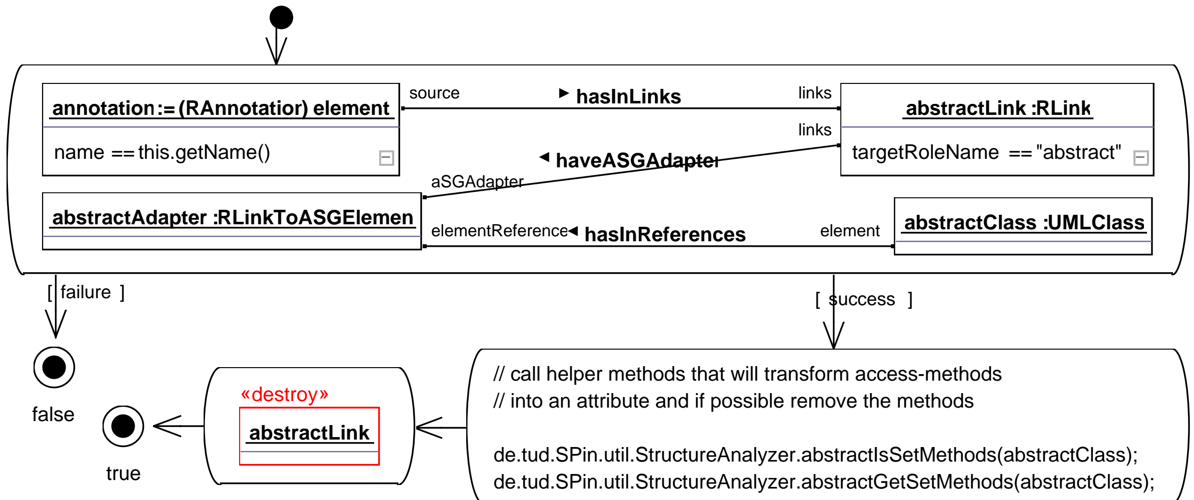


Abbildung B.5: *Eine Regel vom Typ RefinementRule zum Abstrahieren von get/set und is/set Methoden zu Attributen für eine Klasse, die über einen Link „abstract“ an eine Annotierung „DetectAttributes“ gekoppelt ist. Nützlich für das Story Driven Modeling.*

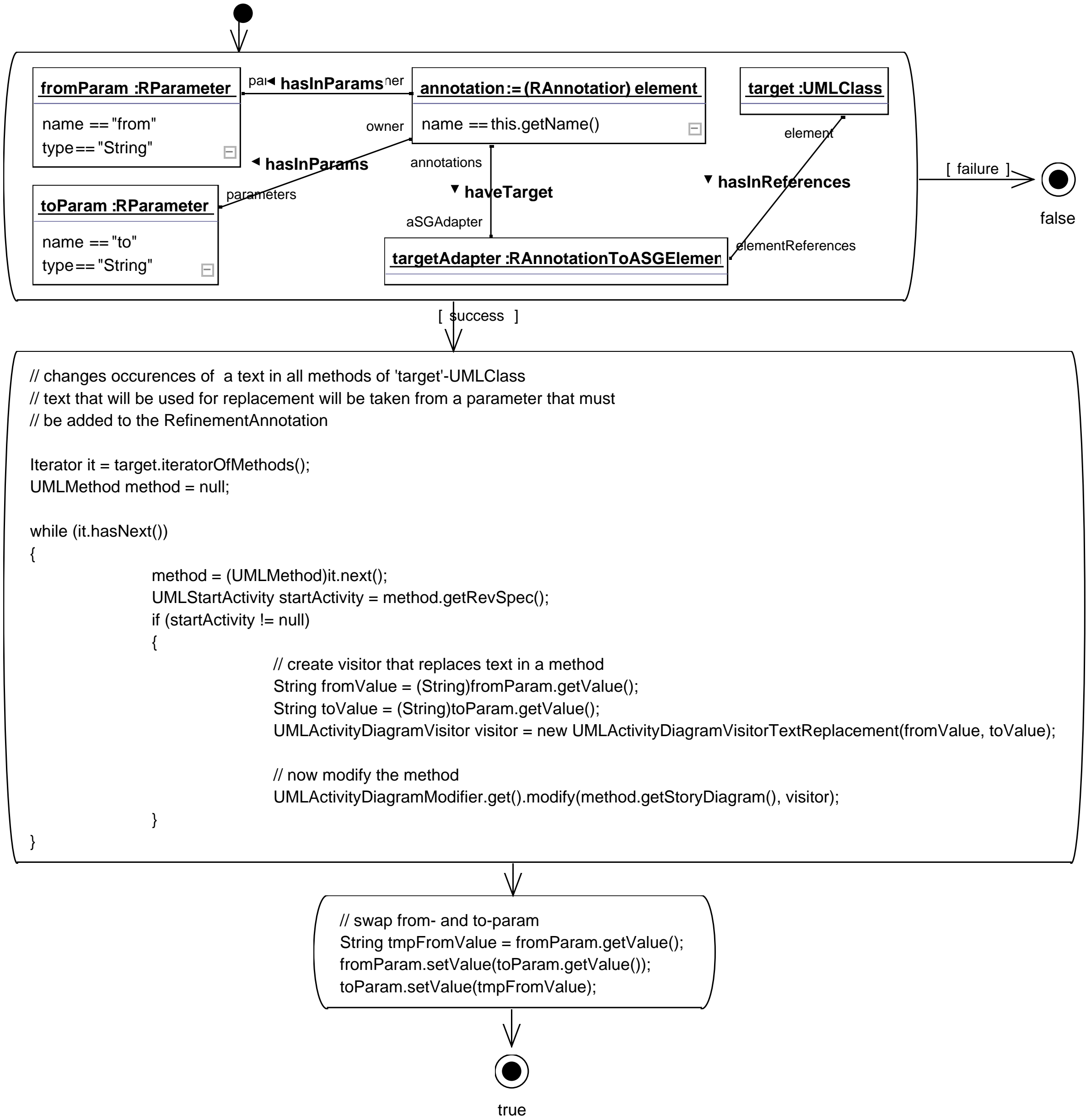


Abbildung B.6: *Eine Regel vom Typ RefinementRule, die eine Textersetzung in Methoden vornimmt.*

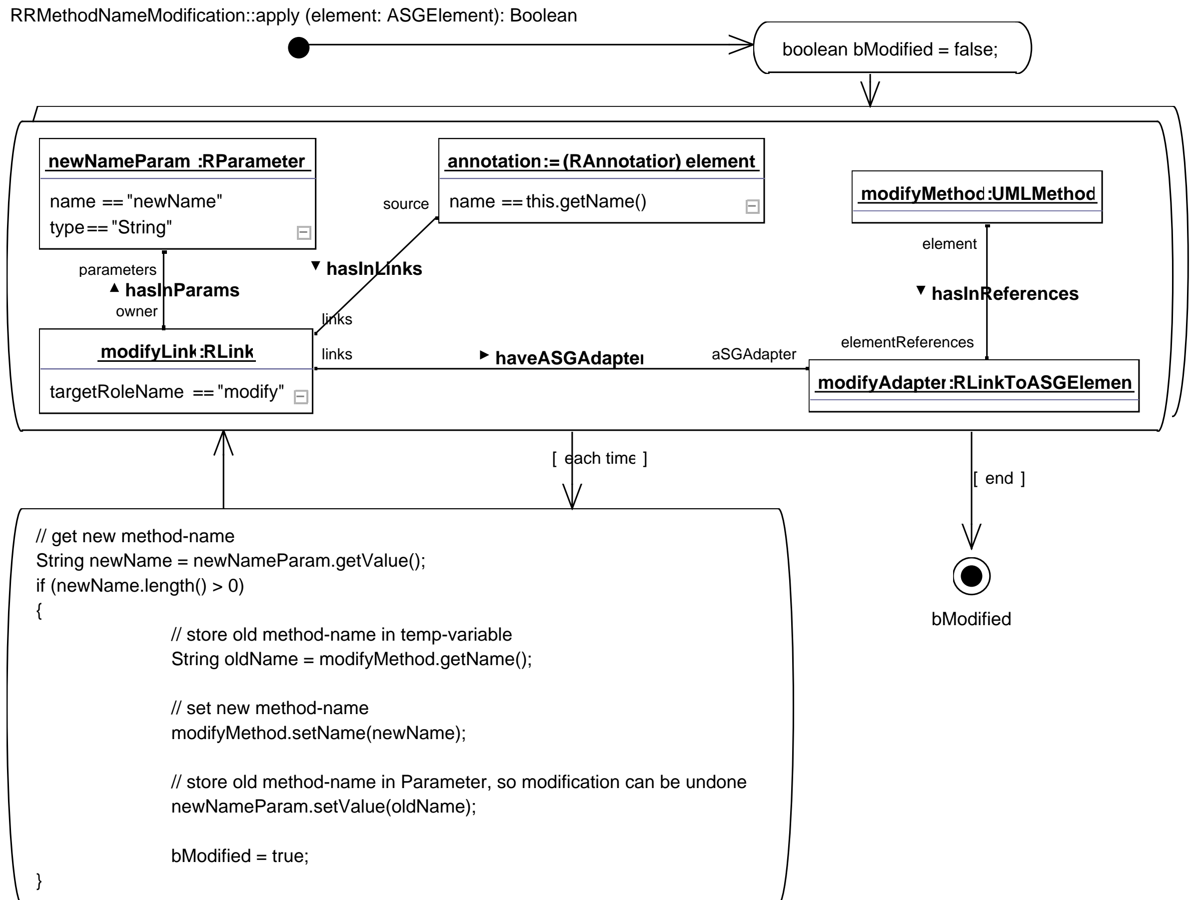


Abbildung B.7: *Eine Regel vom Typ RefinementRule, die den Namen einer Methode ändert. Für jeden Link, der die Annotation mit einer UML Methode verbindet und der einen Parameter vom Typ String mit Namen „newName“ trägt, wird der Methodenname geändert. Der alte Methodenname wird im Parameter gespeichert.*

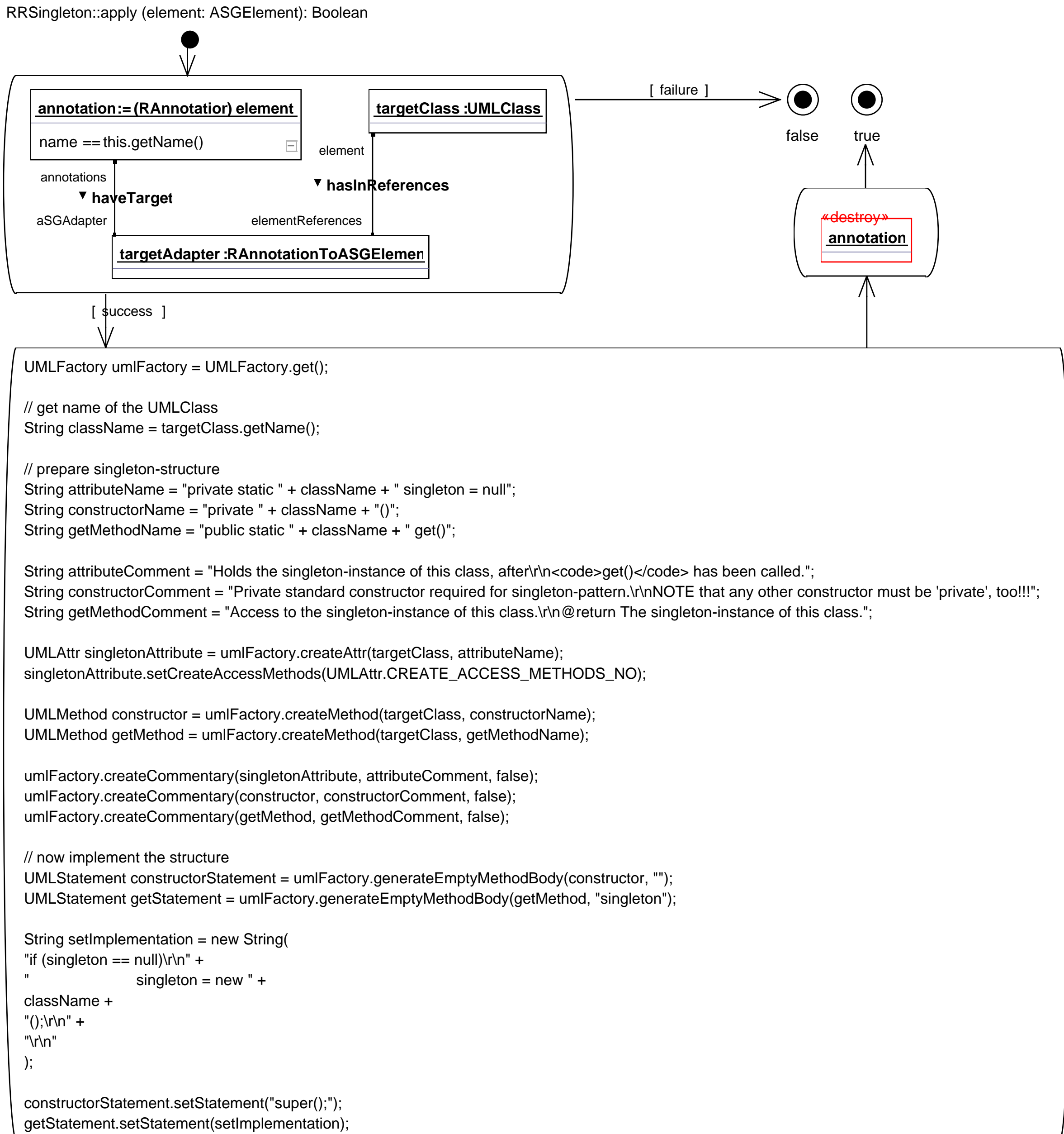


Abbildung B.8: *Eine Regel vom Typ RefinementRule, in der das Singleton-Pattern umgesetzt ist. Macht eine Klasse zu einem „Singleton“.*

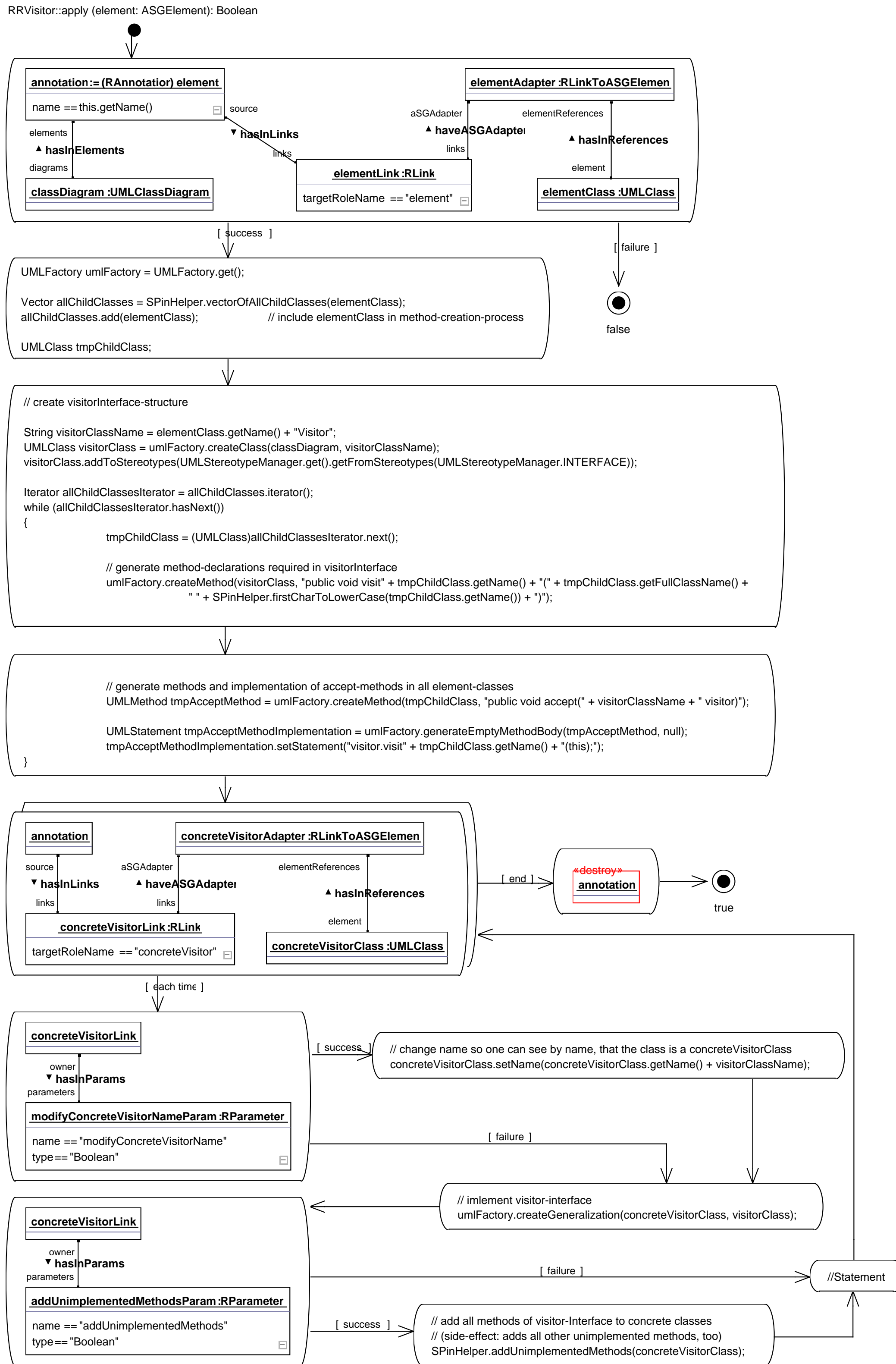


Abbildung B.9: *Eine Regel vom Typ RefinementRule, in der das Visitor-Pattern umgesetzt ist. Es ermöglicht neue Operationen zu definieren, ohne die Klassen der von ihr bearbeiteten Elemente zu verändern.*

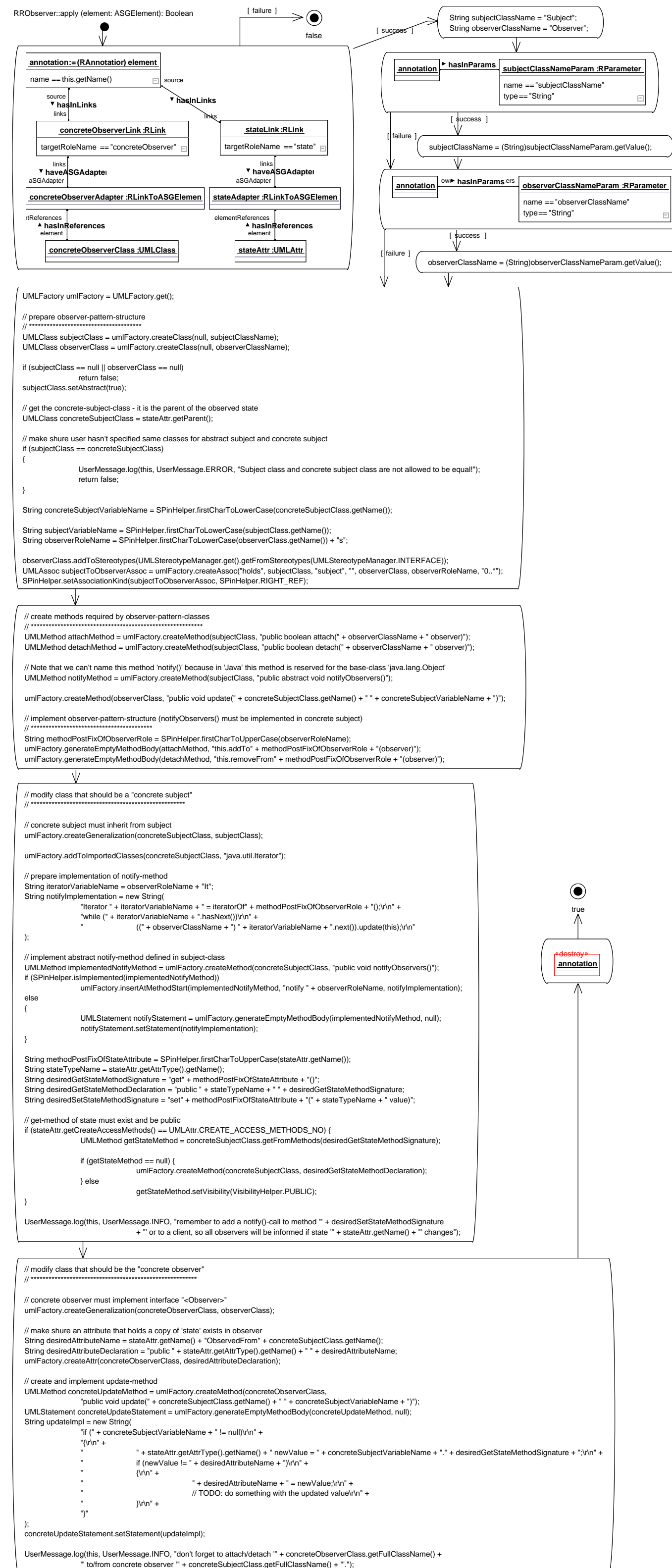


Abbildung B.10: *Eine Regel vom Typ RefinementRule, in der das Observer-Pattern umgesetzt ist. Einem konkreten Beobachter wird mitgeteilt, sobald sich der Zustand eines konkreten Subjekts ändert.*

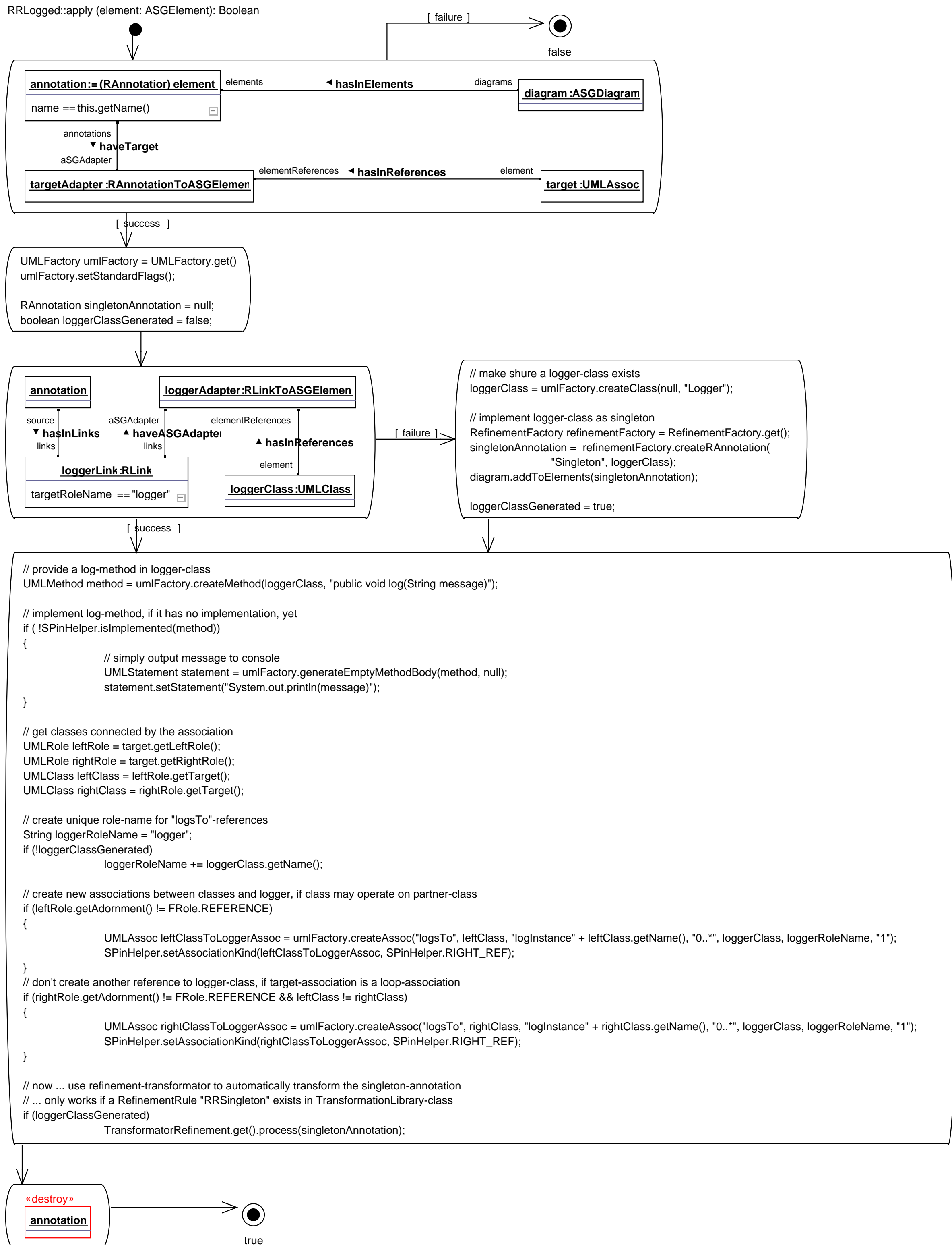


Abbildung B.11: *Eine Regel vom Typ RefinementRule zum Loggen der Kommunikation zwischen zwei Klassen.*

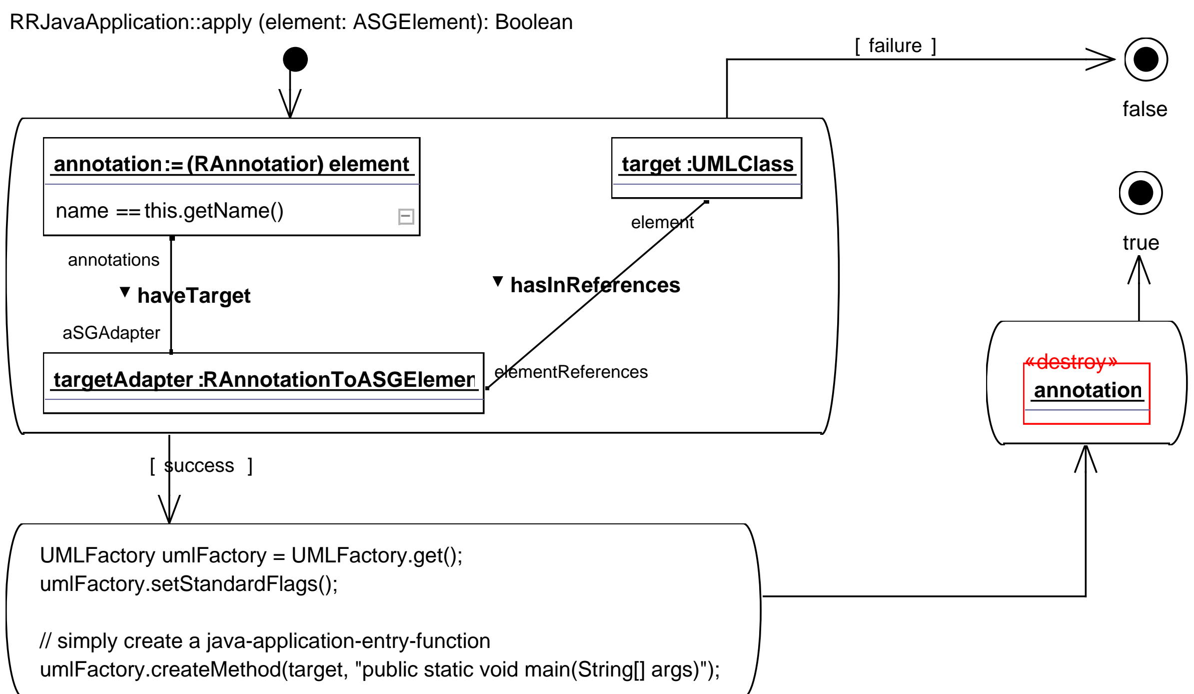


Abbildung B.12: *Eine Regel vom Typ RefinementRule, die einer Klasse die Java-Einsprungsfunktion 'public static void main(String[] args)' hinzufügt.*