UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Institut für Informatik
Arbeitsgruppe Softwaretechnik
Warburger Straße 100
33098 Paderborn

Konzeption und Java Codegenerierung von graphischen Constraints und Pfaden für einen Editor zur Erstellung von Konsistenzerhaltungsmechanismen

Studienarbeit
zur Erlangung des Grades
Bachelor of Computer Science
für den integrierten Studiengang Informatik

von
Christoph Klare
Ostheimer Straße 58
33034 Brakel

vorgelegt bei

Herrn Prof. Dr. Wilhelm Schäfer

und

Herrn Prof. Dr. Gregor Engels

Paderborn, den 1. Dezember 2003

Eidesstattliche Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe, und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Paderborn, den 1. Dezember 2003

Christoph Klare

Inhaltsverzeichnis

Kapitel 1: Einleitung

1.1 Motivation

Die Erstellung von Softwareprodukten läuft heutzutage in einem langen Entwicklungsprozess ab und besteht aus mehreren Phasen. Häufig arbeiten an diesem Prozess Personen aus verschieden Fachbereichen und benutzen dabei unterschiedliche Spezifikationssprachen mit deren Hilfe sie Dokumente zur Beschreibung des zu erstellenden Produkts anfertigen. Durch diese Spezifizierung des Produkts aus den unterschiedlichen Sichtweisen der Entwickler können sich Teile der Spezifikationen überschneiden.

Um am Ende ein möglichst fehlerfreies Produkt zu erhalten, ist es aber erforderlich die einzelnen Spezifikationen zueinander konsistent zu halten. Eine manuelle Überwachung der Konsistenz dieser Dokumente ist jedoch nicht praktikabel, da dies einen erheblichen Aufwand für die Entwickler bedeuten würde. Außerdem würde es den Modellierungsprozess, der meistens parallel von mehreren Entwicklern durchgeführt wird, behindern. Daher spielt für die Softwareentwicklung die dokumentübergreifende Konsistenzsicherung und die Integration der einzelnen Spezifikationssprachen bzw. Dokumente eine wichtige Rolle. Ein Beispiel für die Integration verschiedener Spezifikationssprachen ist die Unified Modeling Language (UML) [UML03]. Sie enthält u. a. Use Case Diagramme, Klassendiagramme, Aktivitätendiagramme, Kollaborationsdiagramme und Sequenzdiagramme. Das Konsistenzproblem ist in Ansätzen wie UML jedoch nicht gelöst, da immer noch Informationen redundant und inkonsistent modelliert werden können.

Zur Erstellung von Softwareprodukten werden viele verschiedene Spezifikationssprachen verwendet. Die in den diversen Sprachen erstellten Dokumente müssen dann miteinander integriert werden.

Ein Ansatz für die Überwachung der Konsistenz zwischen verschiedenen Diagrammarten stellt der in [Wag01] entwickelte Konsistenzerhaltungsmechanismus dar. In diesem Fall wurde ein Konsistenzmanagement-System für das CASE[1] Tool FUJABA[2] erstellt, das die Konsistenz zwischen zwei voneinander verschiedenen Diagrammarten überprüft und falls möglich, Inkonsistenzen automatisch behebt. Falls sie nicht automatisch behoben werden können, werden sie dem Benutzer angezeigt, da dieser zur Behebung der Inkonsistenz zusätzliche Angaben machen muss. Auf diese Weise kann z.B. die Konsistenz zwischen einem UML-Diagramm und einem Diagramm der Specification and Description Language (SDL) gewährleistet werden. Die SDL ist eine von der International Telecommunication Union (ITU) standardisierte Spezifikationssprache und wird vor allem im Bereich der Telekommunikation eingesetzt. Sie dient zur Beschreibung des Systemverhaltens und der Kommunikation in verteilten und eingebetteten Systemen. Für die Spezifikation der Kommunikationsstruktur werden in SDL sogenannte

[1] Computer Aided Software Engineering
[2] From UML to Java And Back Again, http://www.fujaba.de

Blockdiagramme eingesetzt. Diese können auf UML-Klassendiagramme abgebildet werden. Anschließend kann die Konsistenz zwischen den beiden Diagrammarten erhalten werden.

Ein anderes Anwendungsgebiet für konsistenzerhaltende Mechanismen ist der Bereich der Dokumentation von Datenbanken. Viele Datenbanken sind oft jahrelang gewachsen und das ursprüngliche konzeptionelle Schema ist selten noch konsistent mit dem physikalischen Schema. Diese Inkonsistenz tritt auf, weil in vielen Fällen nur das physikalische Schema vom Datenbankentwickler angepasst wird und diese Änderungen häufig auch mangels Zeit nicht im konzeptionellen Schema nachgetragen werden. Das konzeptionelle Schema ist jedoch für die Modernisierung alter Datenbanken, sogenannter Legacy Datenbanksysteme, von hoher Bedeutung.

Ein weiteres Beispiel, in dem das konzeptionelle Schema benötigt wird, ist die Migration von relationalen Datenbanken zu objektorientierten Datenbanken. Die Migration besteht aus drei Schritten: Im ersten Schritt, der Analyse, werden die Informationen aus der alten Datenbank wiedergewonnen. Danach folgt die Schemamigration, die das relationale Schema der alten Datenbank in ein objektorientiertes Schema transformiert. Zuletzt erfolgt die Applikationsmigration, die sich aus der Daten- und der Anfragemigration zusammensetzt. Die Datenmigration erstellt die Daten für das objektorientierte Schema und die Anfragemigration übersetzt die relationalen in objektorientierte Datenbankanfragen. Bei der Schemamigration ist es ebenfalls wichtig, die Konsistenz zwischen den verschiedenen Schemata zu erhalten. Ein Beispiel für einen Mechanismus, der dieses leistet, findet sich in [Wad98].

Für die Entwicklung und Erstellung von konsistenzerhaltenden Mechanismen ist eine Entwicklungsumgebung von Vorteil, die diese Mechanismen mit Hilfe von Graphen modelliert und aus diesen Graphen den Quellcode einer Programmiersprache erzeugen kann. Dazu muss dieser Editor umfangreiche Funktionen bieten, die zur Modellierung notwendig sind.

1.2 Ziel der Arbeit

Das Ziel dieser Studienarbeit ist die Realisierung von zusätzlichen Funktionen für einen Editor, der zur Erstellung von graphischen Regeln zur Konsistenzüberprüfung und -erhaltung verwendet wird. Dieser Editor bietet außerdem die Möglichkeit, aus diesen Regeln Java Code zu generieren. Er wird um folgende Funktionen erweitert:

1. Der Editor wird um die Angabe von graphischen Constraints erweitert. Graphische Constraints sind durch einen Graphen modellierte Restriktionen für Teile der im Editor spezifizierten Regeln.

2. Die Modellierung von graphischen Pfadausdrücken wird in den Editor eingebettet. Diese bieten ein zusätzliches Instrument zur Erstellung kompakter und übersichtlicher Regeln, da sie eine Abkürzung von graphischen Konstrukten darstellen.

3. Die Generierung von Regelsätzen aus den im Editor definierten Diagrammen wird austauschbar und frei wählbar realisiert.

4. Alle neu eingeführten Elemente werden bei der Java Codegenerierung berücksichtigt.

1.3 Aufbau der Arbeit

In Kapitel 2 werden zunächst die für diese Arbeit wichtigen Grundlagen erläutert. Dazu wird zuerst die oben bereits erwähnte Entwicklungsumgebung FUJABA vorgestellt, auf die der hier erweiterte Editor aufsetzt und deren Konzepte er weiterverwendet. Anschließend wird auf das Konzept der Tripel Graph Grammatiken eingegangen, die zur Modellierung von Konsistenzerhaltungsmechanismen verwendet werden können.

In Kapitel 3 wird der existierende Ansatz des Editors, in dem Tripel Graph Grammatik Regeln erstellt werden können, erläutert. Anschließend werden die Konzepte für die einzelnen Erweiterungen des Editors vorgestellt.

In Kapitel 4 wird darauf eingegangen, wie aus den erstellten Diagrammen im Editor Code erzeugt werden kann. Dabei wird auch erläutert, wie die Erweiterungen in die Codegenerierung mit einbezogen werden.

In Kapitel 5 wird anhand einer Beispielsitzung gezeigt, wie die Erstellung eines Diagramms mit allen neu implementierten Funktionen im Editor erfolgt. Anschließend wird vorgeführt, wie aus der erstellten Regel Code generiert wird.

In Kapitel 6 werden die erzielten Ergebnisse der Arbeit zusammengefasst. Dann folgt ein Ausblick auf mögliche zukünftige Ergänzungen des Editors.

Kapitel 2: Grundlagen

2.1 Entwicklungsumgebung (FUJABA)

Das für die Umsetzung der Ziele dieser Studienarbeit genutzte und erweiterte CASE-Tool ist das in der AG Softwaretechnik entwickelte Programm FUJABA. Dieses Programm ist seit 1997, als die erste Version von FUJABA im Rahmen einer Diplomarbeit erstellt wurde [FNT98], kontinuierlich weiterentwickelt worden. Zur Zeit ist es in der Version 4.0 erhältlich und heißt nun FUJABA Tool Suite. Im Folgenden ist mit FUJABA diese Version gemeint.

FUJABA ermöglicht mit Hilfe von diversen UML-Diagrammarten die Spezifikation neuer Software. Aus den Diagrammen kann nach ihrer Erstellung automatisch Code erzeugt werden. FUJABA unterstützt daher eine auf Graphen basierte Modellierung und Implementierung von Software. Außerdem ist es auch möglich, aus bereits vorhandenem Quellcode Diagramme zu extrahieren und so die Struktur des Codes zu visualisieren.

Für FUJABA können sogenannte Plugins erstellt werden, die die FUJABA Architektur nutzen und neue Funktionalitäten bereitstellen [vgl. BGN03]. Ein Beispiel für ein Plugin ist Dobs[1], ein graphischer Debugger, mit dem die in FUJABA spezifizierten Systeme graphisch simuliert werden können. Ein anderes Beispiel ist der in dieser Studienarbeit erweiterte TGG Editor, der in Kapitel 3 vorgestellt wird.

Einige der in FUJABA verwendeten UML-Diagrammarten wurden angepasst, um eine visuelle Programmiersprache zu erschaffen. Da in UML verschiedene Diagramme, wie z.B. Sequenzdiagramme und Kollaborationsdiagramme, für die Beschreibung von Methodenrümpfen genutzt werden können, ist es schwierig eine eindeutige Beschreibung der Abläufe in einer Methode für die Codegenerierung zu erhalten. Deshalb verwendet FUJABA Diagramme des sogenannten Story Driven Modeling (SDM).

SDM bezeichnet eine auf UML basierte Modellierungssprache, die an der Universität Paderborn in der AG Softwaretechnik entwickelt wurde. Im Gegensatz zu UML integriert SDM die in den einzelnen Entwicklungsphasen erstellten Diagramme miteinander und bietet so die Möglichkeit, nicht nur den statischen Teil der Software zu spezifizieren, sondern auch dynamische Veränderungen in den Objektstrukturen zu modellieren. Dadurch ermöglicht es eine Codegenerierung für Methodenrümpfe. Für eine ausführliche Beschreibung von SDM sei hier auf [FNT98] verwiesen. Im Folgenden soll nur ein Einblick in einen Teil von SDM, die Storydiagramme, gewährt werden, da sie für diese Arbeit von Bedeutung sind.

Zu jeder Methode, die man in einem Klassendiagramm in FUJABA deklariert, wird genau ein Storydiagramm angelegt, das die Funktionalität der Methode beschreibt und

[1] Dynamic Object Browsing System

dadurch implementiert. Aus diesem Story Diagramm kann deshalb der Code für die Methode automatisch erstellt werden. Auf diese Weise kann man in FUJABA nicht nur aus statischen Klassendiagrammen Code erzeugen, sondern auch die Dynamik von Software in Diagrammen modellieren und aus diesen Code generieren.

Storydiagramme sind eine Komposition aus UML-Aktivitätendiagrammen und Story Pattern. Story Pattern basieren wiederum auf UML-Kollaborationsdiagrammen und können Objektstrukturen und die Veränderungen, die an diesen Strukturen vorgenommen werden sollen, beschreiben. Die Aktivitäten der Storydiagramme werden mit Hilfe von Story Pattern spezifiziert.

Um dynamische Veränderungen in der Objektstruktur zu modellieren, basieren Story Pattern auf Graph Grammatiken. Graph Grammatiken enthalten eine Menge von Regeln, die spezifizieren, wie die durch sie beschriebene Graphstruktur erstellt werden kann. Jede Regel besteht aus einer linken und einer rechten Regelseite. Die linke Regelseite beschreibt ein Muster von Knoten und Kanten, das in einem Graphen gefunden werden muss. Ist dies der Fall, kann der Graph nach der Beschreibung der rechten Regelseite verändert werden.

Im Story Pattern stellen die Objekte die Knoten und die Links die Kanten dar. Wie auch Graph Grammatiken enthalten Story Pattern eine linke Seite, die als Muster einen Graphen vorgibt. Wird ein solcher Teilgraph, auch Ausgangsgraph genannt, in einer Objektstruktur gefunden, wird die rechte Seite des Story Pattern ausgeführt.

Alle Story Pattern sind von einem Rechteck mit abgerundeten Ecken eingerahmt. Es gibt jedoch auch sogenannte iterierte Story Pattern, die von zwei zueinander versetzten Rechtecken mit abgerundeten Ecken eingerahmt werden. Diese Story Pattern werden für alle Teilgraphen, die in der Objektstruktur gefunden wurden, angewendet. Ein einfaches Story Pattern wird hingegen nur für irgendeinen der gefundenen Teilgraphen ausgeführt, nämlich für den, der als erster gefunden wurde.

Die Abbildung 2.1 zeigt ein Storydiagramm, das eine Methode *example* für Objekte der Klasse *ClassA1* beschreibt und zwei Story Pattern enthält. In jedem Story Pattern sind Objekte enthalten, die über Links untereinander verbunden sein können. Die Objekte werden in Rechtecken dargestellt, die den Namen des Objekt und die zughörige Klasse enthalten, wenn sie noch ungebunden sind. Ein Beispiel für ein ungebundenes Objekt ist *a2* vom Typ *ClassA2* in Story Pattern *SP1*. Gebundene Objekte werden nur durch ihren Namen charakterisiert. Wie man in der Abbildung 2.1 sieht, ist das Objekt *a2* in dem StoryPattern *SP2* bereits gebunden. Optional können die Objekte noch Attribute mit Werten besitzen. Mit *this* wird analog zu Java das Objekt bezeichnet, auf dem die durch das Storydiagramm implementierte Methode, in diesem Fall *example*, aufgerufen wird. Für die Darstellung eines Mengen-Objekts, also eines Objekts, das mehrere Objekte des gleichen Typs enthalten kann, wird dieses von zwei zueinander versetzten Rechtecken umrahmt.

Außerdem gibt es noch Markierungen, die zusätzliche Eigenschaften von Objekten und Links beschreiben. Dies sind z.B. optionale Objekte und Links, die durch gestrichelte Linien gekennzeichnet werden. Sie sind, wie der Name schon sagt, nicht zwingend erforderlich für eine Ausführung des Story Patterns. Das Objekt *a6* in Story Pattern *SP2* ist z.B. optional. Es ist auch möglich negative Objekte oder Links in einem Story Pattern anzugeben. Diese dürfen dann in dem gefundenen Teilgraphen nicht enthalten sein, wenn ein Story Pattern ausgeführt werden soll. Negative Elemente des Graphen werden durch ein Kreuz durchgestrichen angezeigt. Ein Beispiel hierfür ist das Objekt *a5* in Story Pattern *SP2*.

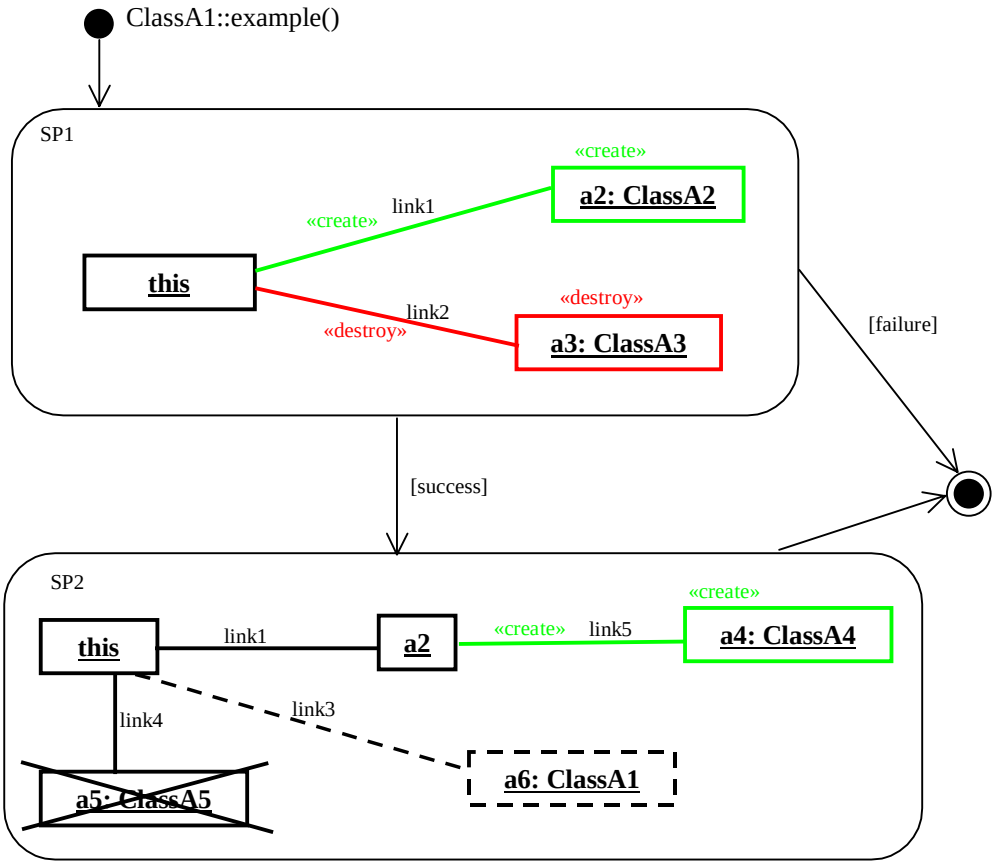

Abbildung 2.1: Beispiel für ein Storydiagramm

Aufgrund der besseren Übersichtlichkeit werden die linke und rechte Regelseite eines Story Patterns zusammengefasst. Um die Regelseiten trotzdem noch unterscheiden zu können, werden einige Objekte und Links mit Stereotypen versehen. Die linke Regelseite, die den zu suchenden Ausgangsgraphen beschreibt, besteht aus allen Objekten und Links ohne Stereotyp. Zusätzlich gehören noch die Objekte und Links mit dem Stereotyp *«destroy»* zu dieser Regelseite. Die rechte Regelseite beschreibt das Ergebnis nach der Anwendung der Regel. Dementsprechend gehören auch hier alle Objekte und Links

ohne Stereotyp zu dieser Seite und außerdem die Objekte und Links, die mit dem Stereotyp *«create»* markiert sind. Dies sind die neu zu erstellenden Objekte und Links. In Abbildung 2.1 ist das zum Beispiel das Objekt *a2* in Story Pattern *SP1*. Mit dem Stereotyp *«destroy»* werden dagegen Objekte und Links gekennzeichnet, die bei Ausführung der Regel gelöscht werden sollen. Deshalb sind diese Elemente auch nicht in der rechten Regelseite enthalten. Ein Beispiel für ein so markiertes Objekt ist *a3* in *SP1*.

Der Teil, der in Storydiagrammen aus Aktivitätendiagrammen übernommen wurde, hat die gleiche Semantik wie in UML und dient zur Modellierung des Kontrollflusses. Es gibt immer eine Start- und eine Stopaktivität. Wie in Abbildung 2.1 zu sehen ist, wird die Startaktivität durch einen schwarz gefüllten Kreis dargestellt. Dort in der Nähe steht auch der Name der Methode, die mit dem Storydiagramm beschrieben wird. Die Stopaktivität ist dagegen ein schwarzgefüllter Kreis, der von einem zweiten Kreis umschlossen wird. Aus ihr führen keine Transitionen heraus. In Storydiagrammen kann es auch Verzweigungen und Schleifen geben. Ein Beispiel für ein Verzweigung zeigt das Story Pattern *SP1* in Abbildung 2.1. Kann dieses Story Pattern angewendet werden (*success*), gibt es eine Transition zum Story Pattern *SP2*. Falls es nicht ausgeführt werden kann (*failure*), führt eine Transition zur Stopaktivität. Für jede Aktivität in einem Storydiagramm existiert ein Story Pattern. Ist ein Objekt in einem Story Pattern gebunden worden, so kann es in den nachfolgenden Story Pattern erneut genutzt werden.

2.2 Tripel Graph Grammatiken

Um die Konsistenz zwischen zwei unterschiedlichen Dokumenten, die sich als Graph darstellen lassen, herzustellen bzw. zu erhalten, haben sich die in [Sch94] und [Lef95] eingeführten Tripel Graph Grammatiken (TGG) als ein geeignetes Instrument herausgestellt.

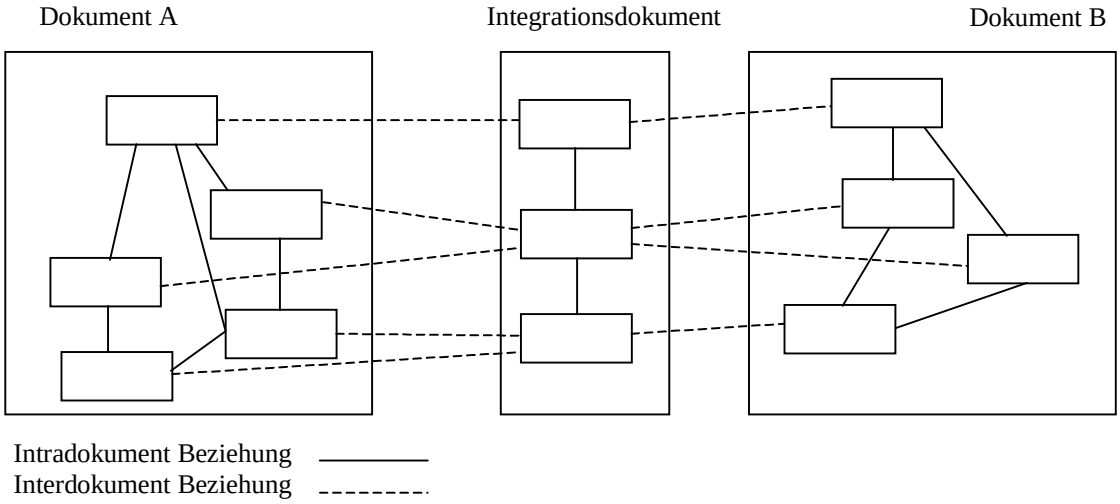

Abbildung 2.2: Integration durch Tripel Graph Grammatiken

Der Ansatz beruht auf den in [Pra71] vorgestellten Pair Graph Grammatiken, die um eine sogenannte Integrationsstruktur erweitert werden. Die korrespondierenden Elemente der beiden Dokumente werden nicht direkt miteinander verknüpft, sondern mit einem Element des sogenannten Integrationsdokuments. Der Name Tripel Graph Grammatiken betont die Wichtigkeit dieses zusätzlichen Korrespondenzgraphen, mit dessen Hilfe sich beide Dokumente aufeinander abbilden lassen. Die Transformation von einem Dokument in ein anderes Dokument ist somit ebenfalls möglich. Die im Abschnitt 1.1 schon erwähnten Diplomarbeiten [Wag01] und [Wad98] verwenden Tripel Graph Grammatiken für ihre Ansätze zur Konsistenzerhaltung. Eine schematische Darstellung der beiden Dokumente und ihrer Beziehungen zueinander über den Korrespondenzgraphen zeigt die Abbildung 2.2. Dort sieht man, dass jedes Element von Dokument *A* und *B* mit einem Element des Integrationsdokuments verbunden ist und keine direkten Beziehungen zwischen Dokument *A* und *B* bestehen.

Der Vorteil von Tripel Graph Grammatiken liegt in der Wiederverwendung der einmal definierten Abbildungsstrukturen. Da sowohl das linke als auch das rechte Dokument nur die sogenannten Map-Knoten des Integrationsdokuments kennen, ist es jederzeit möglich eines von beiden gegen ein drittes Dokument, das auch über Kanten mit den Map-Knoten des Integrationsdokuments verbunden ist, auszutauschen, ohne die Verbindungen des verbliebenen Dokuments zum Integrationsdokument zu verändern. Wären beide Dokumente jedoch direkt miteinander verbunden, so müsste eine komplett neue Abbildung von dem einen in das andere Dokument erstellt werden.

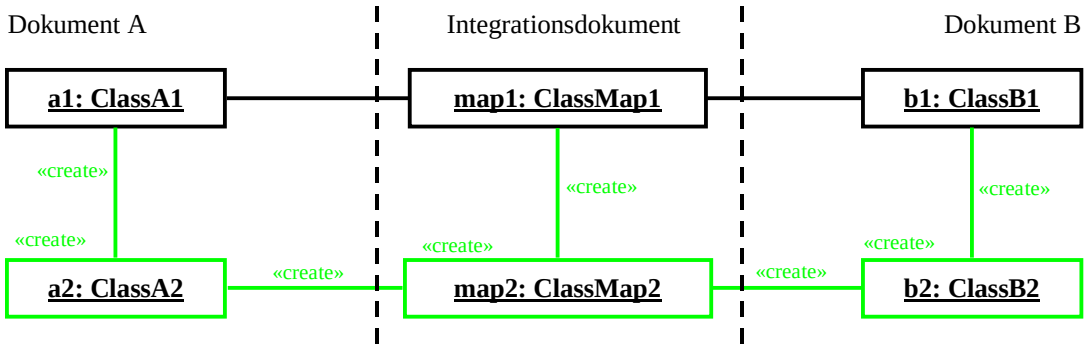

Abbildung 2.3: Beispiel für eine TGG-Regel

Eine Tripel Graph Grammatik besteht aus einem Satz von TGG-Regeln. Eine TGG-Regel besteht aus Objekten der beiden Dokumente *A* und *B* und Objekten des Integrationsdokuments. Sie modelliert einen konsistenten Zustand zwischen den Objekten der Dokumente *A* und *B*. In der TGG-Regel ist sowohl ein Ausgangs- als auch ein Erweiterungsgraph modelliert. Wird der Ausgangsgraph in einer Objektstruktur gefunden, so lässt sich die Regel anwenden, d.h. der Ausgangsgraph wird durch den Erweiterungsgraphen vervollständigt. Ein Beispiel für eine TGG-Regel zeigt die Abbildung 2.3. Der Ausgangsgraph wird durch die schwarzen Objekte und Links ohne Stereotyp beschrie-

ben. Die Objekte und Links des Erweiterungsgraphen sind mit dem Stereotyp *«create»* gekennzeichnet. Die gestrichelten Linien gehören nicht zur Notation sondern dienen lediglich der übersichtlichen Trennung der drei Dokumentstrukturen. Durch TGG-Regeln lassen sich die drei Dokumente in der in Abbildung 2.2 dargestellten Struktur parallel aufbauen.

Aus einer TGG-Regel können drei Graphersetzungsregeln zur Integration und Konsistenzerhaltung abgeleitet werden. Die Bezeichnung der im Folgenden vorgestellten Regeln basiert auf [LS96].

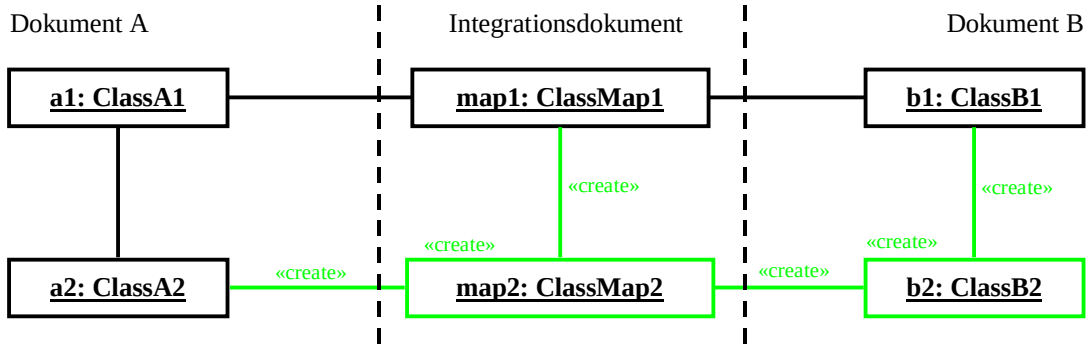

Abbildung 2.4: Vorwärts-Regel

Die Vorwärtsregel wird genutzt, um zu Objekten des linken Dokuments (Dokument *A*) die entsprechenden Objekte des Integrationsdokuments und des rechten Dokuments (Dokument *B*) zu erzeugen und diese mit Links zu verbinden. Der Ausgangsgraph für diese Regel umfasst daher den Ausgangsgraphen der ursprünglichen TGG-Regel und alle zu erstellenden Objekte und Links des linken Dokuments. In Abbildung 2.4 ist die aus der TGG-Regel in Abbildung 2.3 abgeleitete Vorwärts-Regel zu sehen. Diese erzeugt zu dem Objekt *a2* die Objekte *map2* und *b2* sowie die entsprechenden Links.

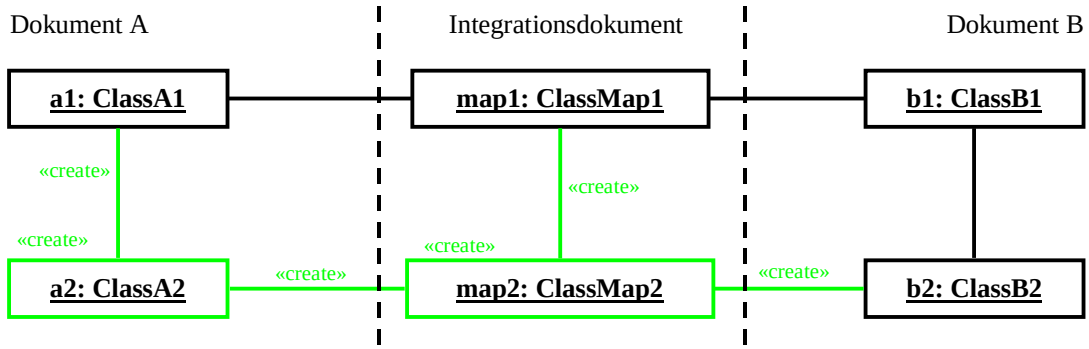

Abbildung 2.5: Rückwärts-Regel

Die zweite Graphersetzungsregel ist die Rückwärts-Regel. Im Gegensatz zur Vorwärts-Regel erzeugt sie zu Objekten des rechten Dokuments *B* die entsprechenden Objekte und Links des Dokuments *A* und des Integrationsdokuments. Der Ausgangsgraph dieser Regel beinhaltet deshalb alle Objekte und Links des Ausgangsgraphen der ursprünglichen TGG-Regel und zusätzlich alle Objekte und Links des rechten Dokuments. In Abbildung 2.5 sieht man die Rückwärts-Regel, die zu dem Objekt *b2* des rechten Dokuments die zugehörigen Objekte der anderen Dokumente erzeugt. Außerdem erzeugt sie noch die Links in und zwischen den einzelnen Dokumenten.

Eine weitere Graphersetzungsregel, die aus einer TGG-Regel abgeleitet werden kann, ist die Konsistenz-Regel. Sie dient zur Konsistenzprüfung zwischen den beiden Dokumenten *A* und *B*. Im Ausgangsgraphen sind sowohl alle Objekte und Links des linken als auch des rechten Dokuments enthalten. Außerdem sind darin die Objekte des Integrationsdokuments enthalten, die dem Ausgangsgraphen der TGG-Regel angehören. Wird der Ausgangsgraph in der vorhandenen Objektstruktur gefunden, erzeugt diese Regel die fehlenden Objekte des Integrationsdokuments und die Links zu dem linken und dem rechten Dokument. Abbildung 2.6 zeigt die aus der TGG-Regel in Abbildung 2.3 abgeleitete Konsistenz-Regel, die zu den schon vorhandenen Objekten *a2* und *b2* das Objekt *map2* des Integrationsdokuments erzeugt.

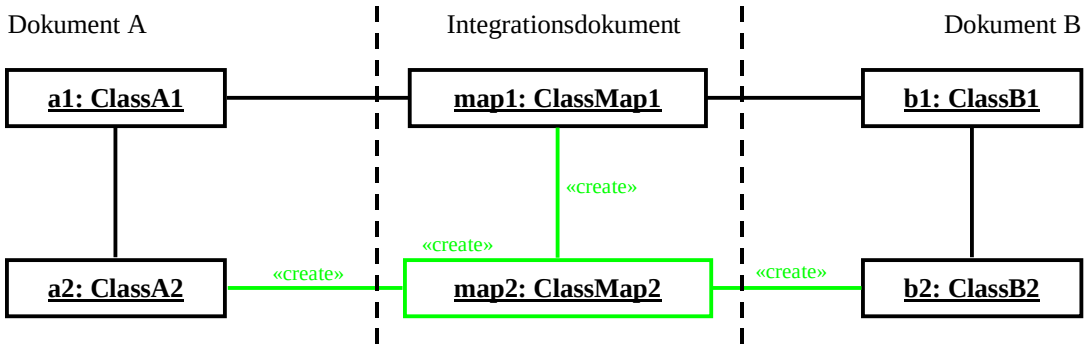

Abbildung 2.6: Konsistenz-Regel

Diese drei Graphersetzungsregeln können schließlich in FUJABA als Storydiagramme modelliert werden. Aus diesen lässt sich wiederum, wie in Abschnitt 2.1 bereits erwähnt, Code generieren. Die Umsetzung der Regeln in Storydiagramme und die Codegenerierung werden in Kapitel 4 noch genauer erklärt.

Kapitel 3: Der TGG Editor

3.1 Existierender Ansatz

Der zu Beginn dieser Arbeit vorhandene Editor für Tripel Graph Grammatiken ist ein Plugin für die im letzten Kapitel vorgestellte Entwicklungsumgebung FUJABA. Mit dem TGG Editor wird die Erstellung von TGG Diagrammen (gleichbedeutend mit TGG-Regeln) in FUJABA möglich.

3.1.1 Aufbau und Einbindung des TGG Editors in FUJABA

Da der TGG Editor ein Plugin ist, kann er nicht allein gestartet werden sondern nur zusammen mit FUJABA. Sein Aufbau richtet sich deshalb nach den Anforderungen, die FUJABA an ein Plugin stellt [vgl. BGN03]. Jedes Plugin muss die Schnittstelle *PluginInterface* implementieren, damit der Plugin-Manager die Plugins beim Start von FUJABA einbinden kann. Die Hauptklasse des TGGEditors *TGGEditorPlugin* erbt, wie in Abbildung 3.1 zu sehen, von der abstrakten Klasse *AbstractPlugin*, die wiederum das Interface *PluginInterface* implementiert. Deshalb kann der Pluginmechanismus den Editor beim Start von FUJABA initialisieren und einbinden.

Abbildung 3.1: Vererbungshierarchie der Klasse TGGEditorPlugin

Der Plugin-Manager lädt sofort nach dem Start von FUJABA alle Plugins aus einem bestimmten Verzeichnis, das im Standardfall den Namen *plugin* hat. In diesem Verzeichnis sind Unterverzeichnisse für die unterschiedlichen Plugins. Auch der TGG Editor ist dort in einem Verzeichnis namens *tgg* enthalten. In diesem Verzeichnis befindet sich der Quellcode des TGG Editors in einem JAR-Archiv. Außerdem enthält das Verzeichnis eine Datei *Plugin.xml*, die allgemeine Informationen zum Editor angibt, und die Datei *stable.xml*, die den Aufbau der einzelnen Menüs in der Benutzungsschnittstelle des Plugins festlegt. Auf diese Weise können Plugins die Benutzungsschnittstelle von FUJABA erweitern. Den in der XML-Datei angegebenen Menüeinträgen wird jeweils eine von der Klasse *AbstractAction* aus dem Java-Swing-Package abgeleitete Klasse zugeordnet. Diese implementiert die Methode *actionPerformed* aus der Oberklasse zur Behandlung von Action-Events, die ausgelöst werden, wenn ein bestimmter Menüeintrag durch den Benutzer angeklickt wurde.

Die interne Verwaltung der einzelnen Diagramme und ihrer Elemente geschieht in FUJABA mit dem FUJABA-MetaModell, das auch abstrakter Syntaxgraph (ASG) genannt wird. Die Elemente des TGG Editors, wie Objekte und Links, stellen eine Spezialisie-

rung von Teilen der UML-Diagramme dar und erben daher die Funktionalität von in FUJABA bereits existenten Klassen des ASG.

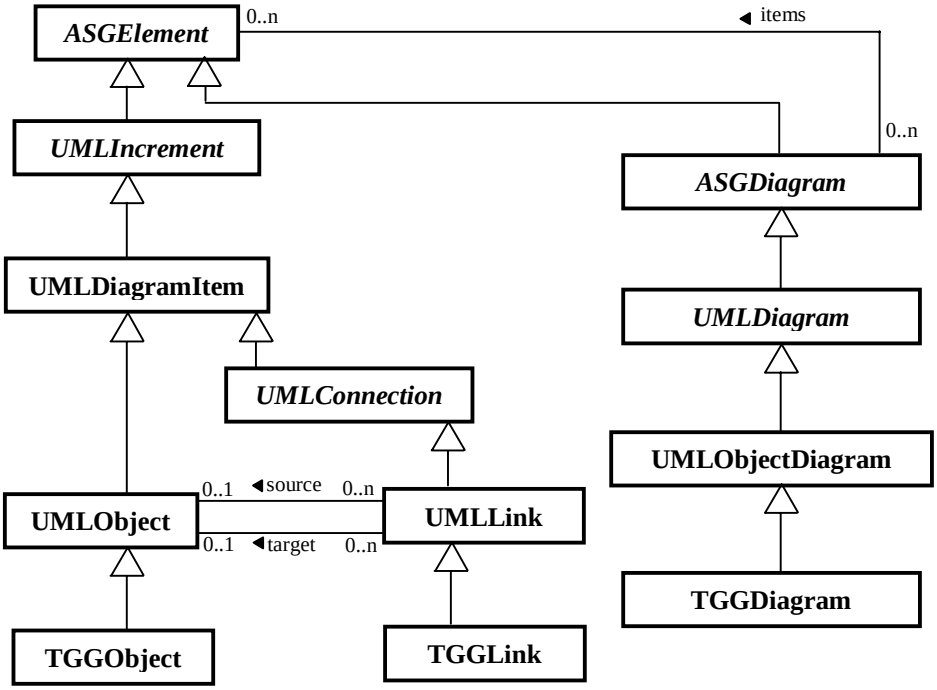

Abbildung 3.2: Metamodell des TGG Editors und von FUJABA

Abbildung 3.2 zeigt einen Auschnitt des FUJABA-Metamodells. Die Klasse *ASGElement* stellt die Wurzelklasse im ASG dar. Von ihr erbt die Klasse *ASGDiagram*, die ASGElemente enthalten kann. Diese Struktur ist angelehnt an das Entwurfsmuster *Composite* [vgl. GHJV95]. Allerdings kann im Gegensatz zum *Composite Pattern* eine Instanz von *ASGElement* eine Assoziation zu mehr als einer Instanz der Klasse *ASGDiagram* besitzen. Beim Entwurfsmuster *Composite* ist aber nur eine Aggregation vorgesehen, die jedes *ASGElement* genau einem *ASGDiagram* zuordnet.

Von der Klasse *ASGDiagram* erbt die Klasse *UMLDiagram*, die die Oberklasse aller UML-Diagrammarten darstellt. Die Klasse *UMLObjectDiagram* ist ein Beispiel für eine dieser Diagrammarten.

Die Struktur der Dokumente, die mittels TGG miteinander integriert werden sollen, wird in FUJABA als Klassendiagramm modelliert. Die Dokumente bestehen dementsprechend aus einer Graphstruktur, die Instanzen dieser Klassen enthält, und können durch ein Objektdiagramm beschrieben werden. Da die TGG-Regeln sich auf die Struktur in diesen Dokumenten beziehen, sind die TGG-Diagramme spezielle Objektdiagramme. Deshalb erbt die Klasse *TGGDiagram* von der Klasse *UMLObjectDiagram*. Die TGG-Objekte eines TGG-Diagramms sind eine Spezialisierung von UML-Objekten. Daher erbt, wie in Abbildung 3.2 dargestellt, die Klasse *TGGObject* von der Klasse *UMLObject*. Diese gemeinsame Beziehung vereinfacht die Umwandlung von TGG-

Diagrammen in Storydiagramme, da in Storydiagrammen UML-Objekte benutzt werden. Alle anderen Klassen, die für die TGG-Diagrammelemente im Editor stehen, wie z.B. *TGGLink*, erben deshalb ebenfalls von einer passenden Klasse, die ein Element eines UML-Diagramms beschreibt und von *UMLDiagramItem* erbt, wie z.B. *UMLLink*.

Die Klassen des TGG Editors sind in verschiedenen Packages nach ihren jeweiligen Aufgaben zusammengefasst. In einem Package sind alle Dialoge, die zur Interaktion mit dem Benutzer dienen, enthalten. Ein anderes Package umfasst die oben schon erwähnten Actionklassen, deren Methoden für den Aufruf der Dialoge über das Menü sorgen. Alle Klassen des Metamodells des TGG-Editors, wie z.B. *TGGObject*, sind ebenfalls in einem Package zusammengefasst.

3.1.2 Bisherige Funktionen des TGG Editors

Wie schon oben kurz erwähnt wurde, ist es vor der Erstellung einer TGG-Regel notwendig, das Metamodell der zu integrierenden Dokumente und des Integrationsdokuments als Graphen zu beschreiben und in FUJABA mit Hilfe eines UML-Klassendiagramms abzubilden. Anschließend können auf der Grundlage dieses Klassendiagramms TGG-Diagramme erstellt werden, für die jeweils ein Name vergeben wird.

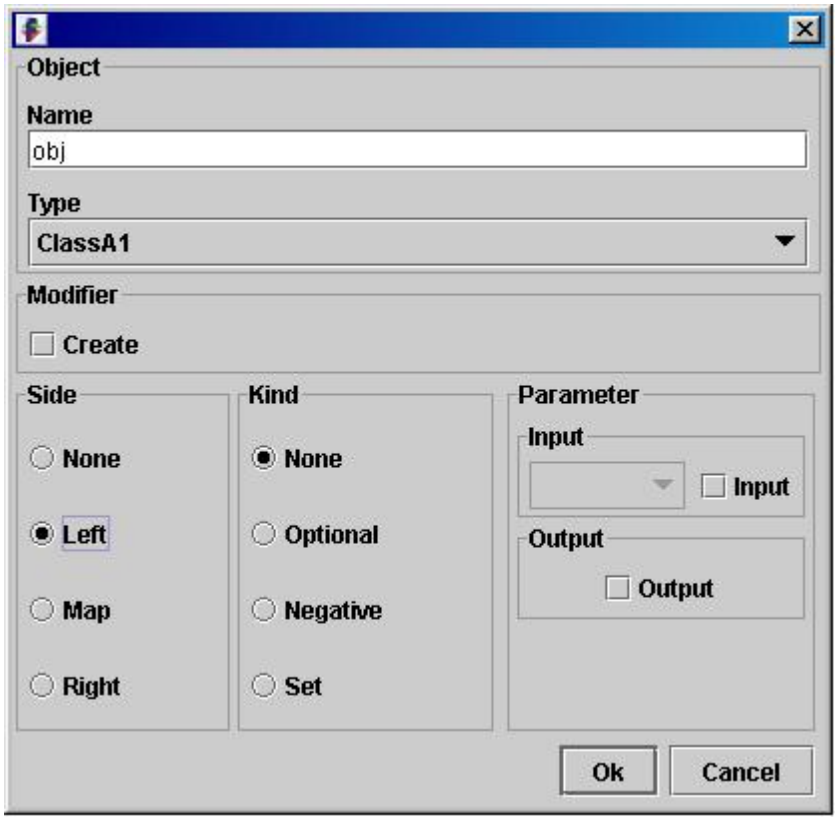

Abbildung 3.3: Erstellen oder Bearbeiten eines Objekts

Da in dem Klassendiagramm nicht angegeben werden kann, welche Klassen zu welchem Dokument gehören, wird bei der Erstellung von Objekten im TGG Editor die Zugehörigkeit zu einem Dokument explizit angegeben. Abbildung 3.3 zeigt den Dialog des Editors zum Erstellen bzw. Bearbeiten von Objekten im TGG Editor. Dort lässt sich ein Name für das Objekt vergeben und der Typ festlegen. Durch Aktivieren des Kästchens *create* kann man es als Objekt des Erweiterungsgraphen definieren. Das Objekt kann dann einem bestimmten Dokument zugeordnet werden, nämlich dem linken Dokument (*Left*), dem rechten Dokument (*Right*) oder dem Integrationsdokument (*Map*). Es ist auch möglich, es keinem Dokument zuzuordnen (*None*). Weiterhin kann man festlegen, ob das Objekt optional oder negativ sein soll oder eine Menge von Objekten repräsentieren soll (*Set*). Als letzte Einstellmöglichkeit beinhaltet der Dialog die Verwendung des Objekts als Input- oder Outputparameter. Da die TGG-Regeln später in Storydiagramme, die Methoden repräsentieren, transformiert werden, kann auf diese Weise angegeben werden, welche Objekte der Methode als Inputparameter übergeben werden sollen und welches Objekt als Outputparameter zurückgeliefert werden soll.

Neben Objekten können auch Links angelegt werden. In einem speziellen Dialog kann man das Quellenobjekt (*Source Object*), das Zielobjekt (*Target Object*) und die Art des Links (*Link Type*) festlegen. Für jeden Link wird auch ein Name vergeben. Außerdem kann man den Link mittels *create* als Teil des Erweiterungsgraphen kennzeichnen.

Alle Objekte und Links, die erstellt worden sind, können bearbeitet oder auch wieder gelöscht werden.

Die Vorschriften zur Generierung von Storydiagrammen aus dem TGG Editor sind im existierenden TGG Editor fest eingebaut und können nur durch direkte Änderungen im Quellcode des Editors und erneutes Kompilieren verändert werden. Es ist nicht möglich zwischen verschiedenen Arten der Generierung, wie sie z.B. in [Müh00] und [Wag01] genutzt werden, auszuwählen.

3.2 Konzeptionelle Erweiterungen

Zur Erstellung von TGG-Regeln mit Hilfe des TGG Editors sind dessen Möglichkeiten im bisher geschilderten Zustand noch beschränkt. Daher wurden im Rahmen dieser Studienarbeit die hier vorgestellten Erweiterungen in den Editor integriert.

3.2.1 Graphische Constraints

Die in Abschnitt 2.2 eingeführte Notation von Tripel Graph Grammatiken kann um die Angabe graphischer Constraints erweitert werden. Graphische Constraints dienen zur genaueren Beschreibung von Objekten des Ausgangsgraphen. Sie stellen eine zusätzliche Randbedingung für die Suche nach einem Muster, das dem Ausgangsgraphen entspricht, dar. Wie Abbildung 3.4 zeigt, wird ein graphischer Constraint durch einen Doppelpfeil auf ein Objekt definiert. An diesem Pfeil steht zunächst nur der Name des Constraints. Dieser Name steht für ein Storydiagramm, das die Restriktion für das Ob-

jekt beschreibt. Für eine erfolgreiche Ausführung der TGG-Regel muss die durch das Storydiagramm formulierte Bedingung wahr sein.

Constraints wurden auch in [Wad98] verwendet und stellten ein mächtiges zusätzliches Instrument für die Definition von TGG-Regeln dar. Allerdings wurden die Regeln in [Wad98] noch in PROGRES-Notation (PROgrammierte GRaphErsetzungsSysteme) beschrieben und in einem dafür passenden Editor erstellt. Nun gilt es, die Constraints auch bei der Erstellung von TGG-Regeln im TGG Editor einsetzen zu können.

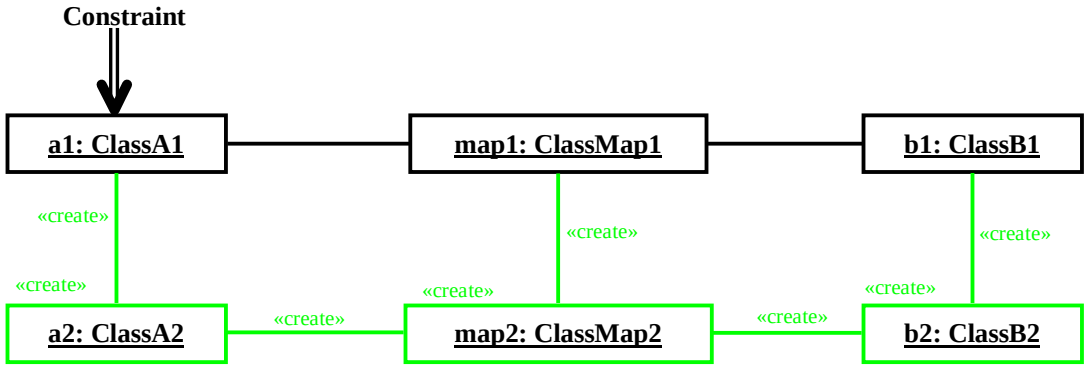

Abbildung 3.4: Notation eines graphischen Constraints

3.2.2 Side-Constraints

Außer graphischen Constraints können in TGG-Regeln auch sogenannte Side-Constraints benutzt werden. Sie sind wichtig, um die Konsistenz zwischen zwei Dokumenten in Bezug auf die Attribute ihrer Objekte sicherzustellen. Die Side-Constraints haben z.B. die Aufgabe, die Attributwerte von zwei verschiedenen Objekten zu vergleichen bzw. dem Attribut eines Objekts den Attributwert eines anderen Objekts zuzuweisen. Es ist aber z.B. auch möglich ein Attribut auf einen festen Wert zu setzen.

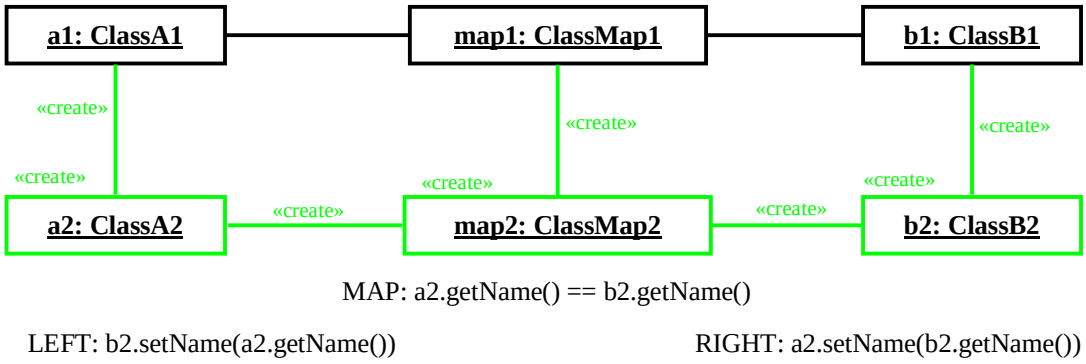

Abbildung 3.5: Einsatz von Side-Constraints

In Abbildung 3.5 ist ein Beispiel für den Einsatz von Side-Constraints zu sehen. Es können sowohl für das linke Dokument (*LEFT*), das rechte Dokument (*RIGHT*) oder das Integrationsdokument (*MAP*) Side-Constraints angegeben werden. Diese Unterscheidung ist wichtig für die spätere Regelgenerierung, da die verschiedenen Graphersetzungsregeln unterschiedliche Side-Constraints verwenden müssen. In diesem Beispiel ist für das linke Dokument angegeben, dass dem Attribut *name* von *b2* der Wert des Attributs *name* von *a2* zugewiesen werden soll. Für das rechte Dokument ist diese Zuweisung genau andersherum realisiert. Je nachdem, ob nun bei einer Graphersetzungsregel das Objekt *a2* oder das Objekt *b2* neu erzeugt wird, wird der jeweilige Side-Constraint ausgeführt. Falls nur das Objekt *map2* des Integrationsdokuments neu erzeugt wird, muss dafür dementsprechend gelten, dass die Attribute *name* von *a2* und von *b2* den gleichen Wert haben. Auf diese Weise erreicht man eine Konsistenz zwischen den Objekten des rechten und linken Dokuments in Bezug auf deren Attribute. Die Formulierung der Side-Constraints in Java Syntax hat den Vorteil, dass sie später bei der Codegenerierung direkt übernommen werden können. Für jede TGG-Regel können beliebig viele Side-Constraints definiert werden.

3.2.3 Graphische Pfadausdrücke

Um die Struktur der Objekte im Ausgangsgraphen besser und einfacher beschreiben zu können, muss eine Möglichkeit geschaffen werden, Pfade anzugeben. Auch dieses Konstrukt wurde in PROGRES-Notation in [Wad98] bereits verwendet und hat sich als nützlich erwiesen.

Der Vorteil bei der Nutzung von Pfadausdrücken ist die Einsparung von Zwischenknoten und Kanten, die sonst angegeben und eingezeichnet werden müssten. Auf diese Weise erhöht sich außerdem die Übersichtlichkeit komplexer TGG-Regeln.

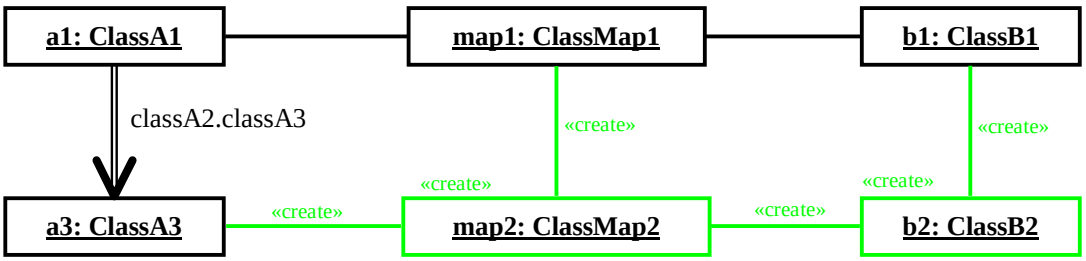

Abbildung 3.6: Notation eines graphischen Pfadausdrucks

Durch einen graphischen Pfadausdruck kann also festgelegt werden, dass ein bestimmtes Objekt mit einem oder einer Menge von Objekten einer bestimmten Klasse verbunden sein muss. Im Pfadausdruck wird außerdem noch spezifiziert, über welche Art von Objekten diese Verbindung bestehen muss. Dazu wird ein graphischer Pfadausdruck in dieser Arbeit durch die Rollennamen der Objekte, die sich auf dem Pfad befinden, spe-

zifiziert. Die Rollennamen werden im UML-Klassendiagramm für jede Seite einer Assoziation festgelegt und geben die Rolle an, die ein Objekt dieser Klasse innerhalb der Beziehung mit dem anderen Objekt spielt. Da in FUJABA als Standardwert für einen Rollennamen der Name der jeweiligen Klasse mit kleinem Anfangsbuchstaben verwendet wird, sind in den Beispielen dieser Arbeit die Rollen- und die Klassennamen identisch.

Im Pfadausdruck werden die jeweiligen Rollennamen durch Punkte voneinander abgetrennt. Im Beispiel in Abbildung 3.6 ist dies der Ausdruck *classA2.classA3*. Das doppelte Rechteck um das Objekt *a3* beschreibt, dass es sich hierbei auch um eine Menge von Objekten handeln kann.

3.2.4 Dynamische Generierung von Storydiagrammen

In Abschnitt 2.2 wurde ein TGG-Regelsatz vorgestellt, der die Vorwärts-, die Rückwärts- und die Konsistenzregel beinhaltet. Es existieren aber auch andere Regelsätze. So werden z.B. in [Wag01] folgende Regeln definiert: die Vorwärts-Regel, die Vorwärts-Lösch-Regel, die Vorwärts-Konsistenz-Regel, die Rückwärts-Regel, die Rückwärts-Lösch-Regel und die Rückwärts-Konsistenz-Regel. Zu jeder im TGG Editor spezifizierten TGG-Regel müssen in diesem Fall sechs statt drei Graphersetzungsregeln abgeleitet werden.

In [Müh00] wurde, wie schon in Abschnitt 3.1.2 erwähnt, eine feste Generierung von Storydiagrammen aus TGG-Regeln nach einem bestimmten Regelsatz implementiert. Da der TGG Editor jedoch flexibel eingesetzt werden soll, wird in dieser Arbeit eine austauschbare Generierung von Storydiagrammen aus TGG-Regeln in den Editor integriert. Dazu müssen die verschiedenen Regelsätze zur Laufzeit gegeneinander austauschbar sein. Jeder Regelsatz, auch Strategie genannt, enthält direkt die Vorschrift, wie aus einer TGG-Regel Storydiagramme entstehen.

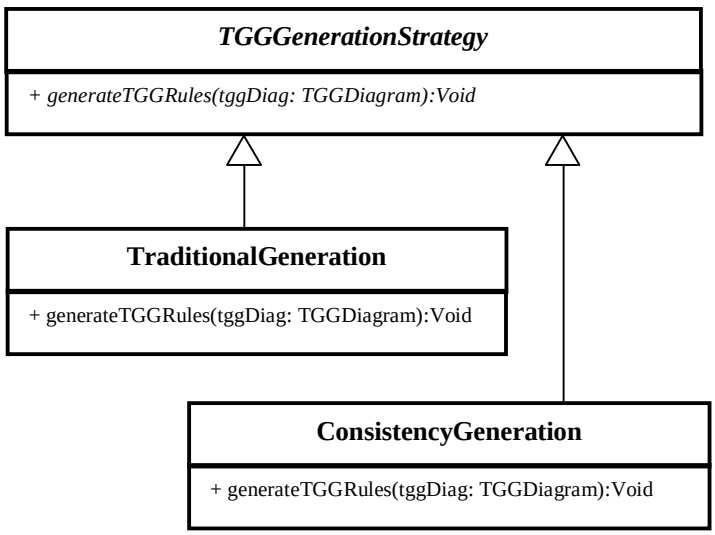

Abbildung 3.7: Die Klasse TGGGenerationStrategy und ihre Unterklassen

Um eine Gruppe unterschiedlicher Strategien zur Generierung von Storydiagrammen aus TGG-Regeln untereinander austauschbar zu machen, bietet sich das Entwurfsmuster *Strategy* [vgl. GHJV95] an. Wie in Abbildung 3.7 zu sehen ist, erben die einzelnen Strategien von der abstrakten Klasse *TGGGenerationStrategy*. Hier sind nur zwei Beispielstrategien aufgeführt. Es können natürlich beliebig viele verschiedene sein. Jede der Unterklassen implementiert die Methode *generateTGGRules.* Als Eingabeparameter bekommt diese Methode ein TGGDiagramm aus dem Storydiagramme anhand der vorgegebenen Strategie erstellt werden.

Über einen Dialog kann der Benutzer schließlich im Editor aus den verschiedenen implementierten Strategien eine auswählen und auf eine oder alle modellierten TGG-Regeln anwenden.

Kapitel 4: Codegenerierung

Für die Erzeugung von Java Code aus TGG-Regeln werden diese zunächst in Storydiagramme transformiert und dann mit Hilfe der Codegenerierung von FUJABA in Quelltext übersetzt. Dieses Kapitel beschreibt zunächst die Erstellung der Storydiagramme und erläutert dann die Codegenerierung von FUJABA. Außerdem geht es darauf ein, wie die im letzten Kapitel vorgestellten neuen TGG-Elemente in die Codegenerierung eingebunden werden.

4.1 Generierung von Storydiagrammen aus TGG-Regeln

Die Erstellung von Storydiagrammen kann, wie im letzen Kapitel bereits erläutert, auf verschiedene Arten geschehen. Die Strategie, an der die Generierung hier beispielhaft erklärt wird, ist die schon in [Wad98] benutzte traditionelle Strategie, die aus einer TGG-Regel eine Vorwärts-, eine Rückwärts- und eine Konsistenzregel erstellt.

Die Abbildung 4.1 zeigt den Dialog, aus dem der Benutzer eine Strategie auswählen kann. Aktiviert er zusätzlich das Kästchen *Generate all*, werden aus allen TGG-Regeln nach dem ausgewählten Regelsatz Storydiagramme erzeugt.

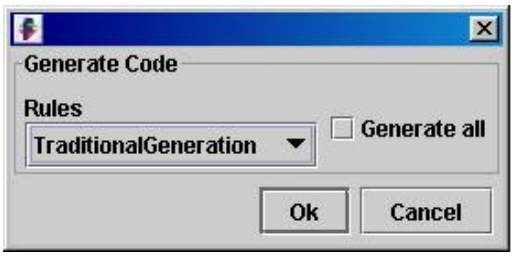

Abbildung 4.1: Dialog zur Auswahl der Generierungsstrategie

All diese Strategien implementieren die Schnittstellenmethode *generateTGGRules* der abstrakten Klasse *TGGGenerationStrategy*, die als Parameter ein TGG-Diagramm übergeben bekommt. Diese Methode enthält die Vorschrift, wie und wie viele Storydiagramme aus der vorgegebenen TGG-Regel erzeugt werden sollen und kann von Strategie zu Strategie sehr unterschiedlich sein. Im Folgenden soll nun die Methode *generateTGGRules* der Klasse *TraditionalGeneration*, die die oben schon genannte traditionelle Strategie umsetzt, näher erläutert werden.

Als erstes wird für die Transformation der TGG-Regeln ein neues Klassendiagramm *GeneratedMappingRules* in FUJABA erzeugt. Anschließend wird im Klassendiagramm eine neue Klasse *MappingRules* erstellt, deren drei Methoden jeweils die Vorwärts-, Rückwärts- und die Konsistenzregel repräsentieren.

Zu jeder Methode wird ein Storydiagramm erstellt, das nach dem in Abbildung 4.2 dargestellten Schema aufgebaut ist. Hier wird zunächst nur der Aufbau der Storydiagram-

me, die die Graphersetzungsregeln beschreiben, erläutert. Im folgenden Abschnitt 4.1.1 wird daraufhin näher auf die Erstellung der im Storydiagramm enthaltenen Story Pattern eingegangen.

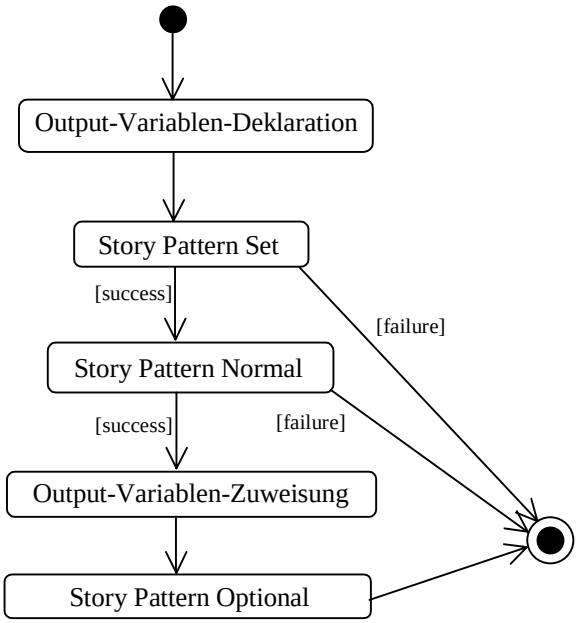

Abbildung 4.2: Schema eines generierten Storydiagramms (Traditional Generation)

Als erstes wird die Startaktivität des Storydiagramms erstellt. Falls ein Ausgabeparameter in der TGG-Regel angegeben wurde, wird er in der folgenden Aktivität deklariert und mit *null* initialisiert. Als nächstes wird das *Story Pattern Set* erzeugt. Dieses geschieht aber nur, wenn in der TGG-Regel Mengen-Objekte oder Constraints definiert wurden. Falls dieses Story Pattern angewendet werden kann, folgt anschließend das *Story Pattern Normal*, das auf Grundlage der TGG-Regel die entsprechenden Objekte und Links bindet sowie neue Objekte und Links erzeugt. Im anderen Fall (failure) wird der Endknoten des Story Pattern über eine Transition erreicht. Wenn in der TGG-Regel auch optionale Objekte bzw. Links vorhanden sind, folgt nach erfolgreicher Ausführung des *Story Pattern Normal* das *Story Pattern Optional*. Dort werden die Mengen-Objekte und alle optionalen Objekte und Links gebunden. Vorher wird noch in einer Aktivität dem Ausgabeparameter des Storydiagramms das im *Story Pattern Normal* gebundene Objekte zugewiesen. Nach der Ausführung des *Story Pattern Optional* folgt eine Transition zum Endknoten des Storydiagramms.

4.1.1 Erstellung der Story Pattern

Anhand der zugrunde liegenden TGG-Regel werden die Story Pattern erstellt. Dabei werden aus TGG-Objekten UML-Objekte und aus TGG-Links UML-Links. Je nach zu erzeugender Graphersetzungsregel werden alle Objekte eines Dokuments dem Ausgangsgraphen zugeordnet und die Stereotypen *«create»* entfernt. Wie in Abschnitt 2.2 erläutert, werden für die Vorwärtsregel alle mit *«create»* gekennzeichneten Objekte und

Links des linken Dokuments geändert in Objekte und Links des Ausgangsgraphen. Analog dazu werden demnach für die Rückwärtsregel alle Elemente des rechten Dokuments dem Ausgangsgraphen hinzugefügt. Für die Konsistenzregel geschieht dies sowohl für die Elemente des linken als auch für die Elemente des rechten Dokuments.

Das *Story Pattern Set* wird nur erstellt, wenn in der TGG-Regel Mengen-Objekte enthalten sind. Falls dies der Fall ist, wird hier überprüft, ob diese Mengen nicht leer sind. Dazu wird ein Graph aus allen Objekten des Ausgangsgraphen, die nicht optional sind, erstellt und alle Mengen-Objektknoten werden zu normalen Objektknoten. Bis auf die gebundenen Eingabeparameter werden alle anderen Objekte an Objektvariablen gebunden, deren Namen gegenüber den in der TGG-Regel angegebenen das Präfix *Check* vorausgeht. Diese Umbenennung wird hier vorgenommen, da erst im für jede Regel existierenden *Story Pattern Normal* alle Objekte an die Objektvariablen gebunden werden sollen.

Im *Story Pattern Normal* wird versucht, alle Objekte des Ausgangsgraphen zu binden und falls dies erfolgreich war, die restlichen in der TGG-Regel angegebenen Objekte und Links des Erweiterungsgraphen neu zu erzeugen. Auch hier werden keine optionalen Objektknoten berücksichtigt. Die Mengen-Objektknoten spielen in diesem Story Pattern ebenfalls keine Rolle. Bei erfolgreicher Anwendung des Story Pattern ist auch die Ausführung des Storydiagramms erfolgreich

Das *Story Pattern Optional* behandelt die in den vorangegangenen Story Pattern bisher ignorierten optionalen Objektknoten sowie die Mengen-Objektknoten. Es enthält den Objektgraphen mit den bereits gebundenen Objekten und zusätzlich die Mengen-Objektknoten und die optionalen Objektknoten, die hier, falls vorhanden, gebunden werden. Wenn dies erfolgreich war, werden auch die daraufhin zu erzeugenden Links neu angelegt. Da die optionalen Objekte für eine erfolgreiche Ausführung des Storydiagramms nicht unbedingt vorhanden sein müssen und im *Story Pattern Set* bereits geprüft wurde, ob die Mengen nicht leer sind, führt aus diesem Story Pattern nur eine Transition heraus.

Ein Beispiel für ein aus einer TGG-Regel erzeugtes Storydiagramm ist in Abbildung 4.4 zu sehen. Dabei handelt es sich um die Vorwärtsregel der in Abbildung 4.3 dargestellten TGG-Regel. Anhand dieses Storydiagramms soll nun die Einbindung der graphischen Constraints, der Side-Constraints und der graphischen Pfadausdrücke anschaulich dargestellt werden.

4.1.2 Einbindung von graphischen Constraints

Die Verarbeitung der Constraints erfolgt in dem *Story Pattern Set*. Dieses Story Pattern wird also nicht nur angelegt, wenn die vorliegende TGG-Regel Mengen-Objekte enthält, sondern auch, wenn sie Constraints enthält. Die Constraints werden bereits bei der Erstellung der TGG-Regel in FUJABA als Storydiagramm definiert. All diese Methoden, die einen Constraint beschreiben, befinden sich in der Klasse *ConstraintRules* und sind Klassenmethoden. Als Parameter bekommen sie das Objekt übergeben, für das der

Constraint erstellt wurde. Als Rückgabeparameter liefern sie einen boolschen Wert. Daher können zur Überprüfung der Constraints sogenannte UML-Constraints verwendet werden. Das sind textuelle boolsche Bedingungen, die in FUJABA innerhalb von Story Pattern angewendet werden können. Falls nicht alle angegebenen Bedingungen wahr sind, schlägt die Ausführung des Story Pattern und damit die Ausführung des Storydiagramms fehl.

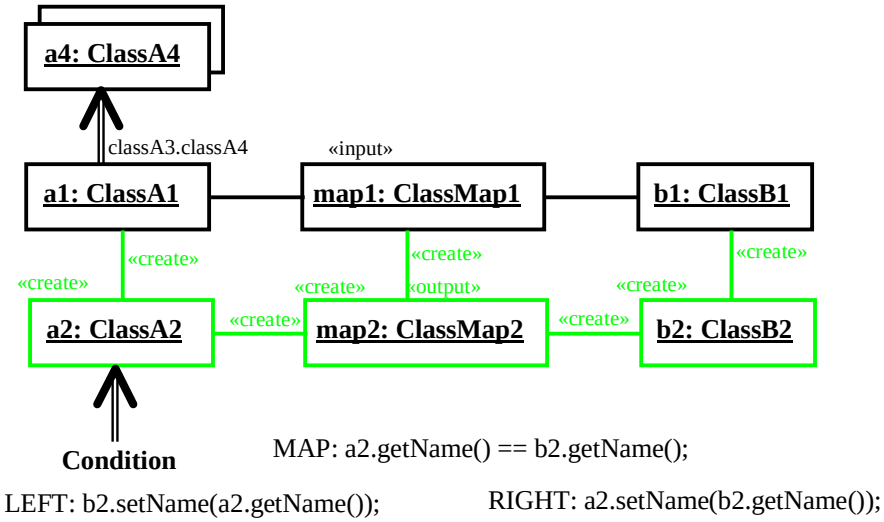

Abbildung 4.3: Beispiel für eine TGG-Regel

Da alle Objekte des Ausgangsgraphen der Regel im *Story Pattern Set* schon zu Testzwecken gebunden werden, können hier auch die Constraints für die einzelnen Objekte behandelt werden. Um einen Constraint zu überprüfen, wird in einem UML-Constraint ein Aufruf der Methode, die durch den Constraint definiert wird, für sein zugehöriges Objekt in Java-Syntax erstellt. Bei der späteren Java Codegenerierung kann dieser Aufruf dann direkt in den Code übernommen werden. In unserem Beispiel wird der Constraint *Condition* aus der TGG-Regel im Storydiagramm durch den in einem UML-Constraint definierten Aufruf *ConstraintRules.Condition(a2Check)* überprüft.

4.1.3 Einbindung von Side-Constraints

Im Gegensatz zu den Constraints erfolgt die Verarbeitung der Side-Constraints im *Story Pattern Normal*. Da innerhalb von UML-Constraints auch Zuweisungen gemacht werden können, wird bei der Einbindung der SideConstraints in die Storydiagramme ebenfalls auf diese zurückgegriffen.

Je nach erzeugter Regel muss bei den Side-Constraints jedoch unterschieden werden, ob sie im Storydiagramm enthalten sein müssen oder nicht. Durch die Angabe der Seite kann dies entschieden werden. Für die Vorwärtsregel werden demnach die mit *LEFT*, für die Rückwärtsregel die mit *RIGHT* und für die Konsistenzregel die mit *MAP* gekennzeichneten Side-Constraints ins zugehörige Storydiagramm übernommen.

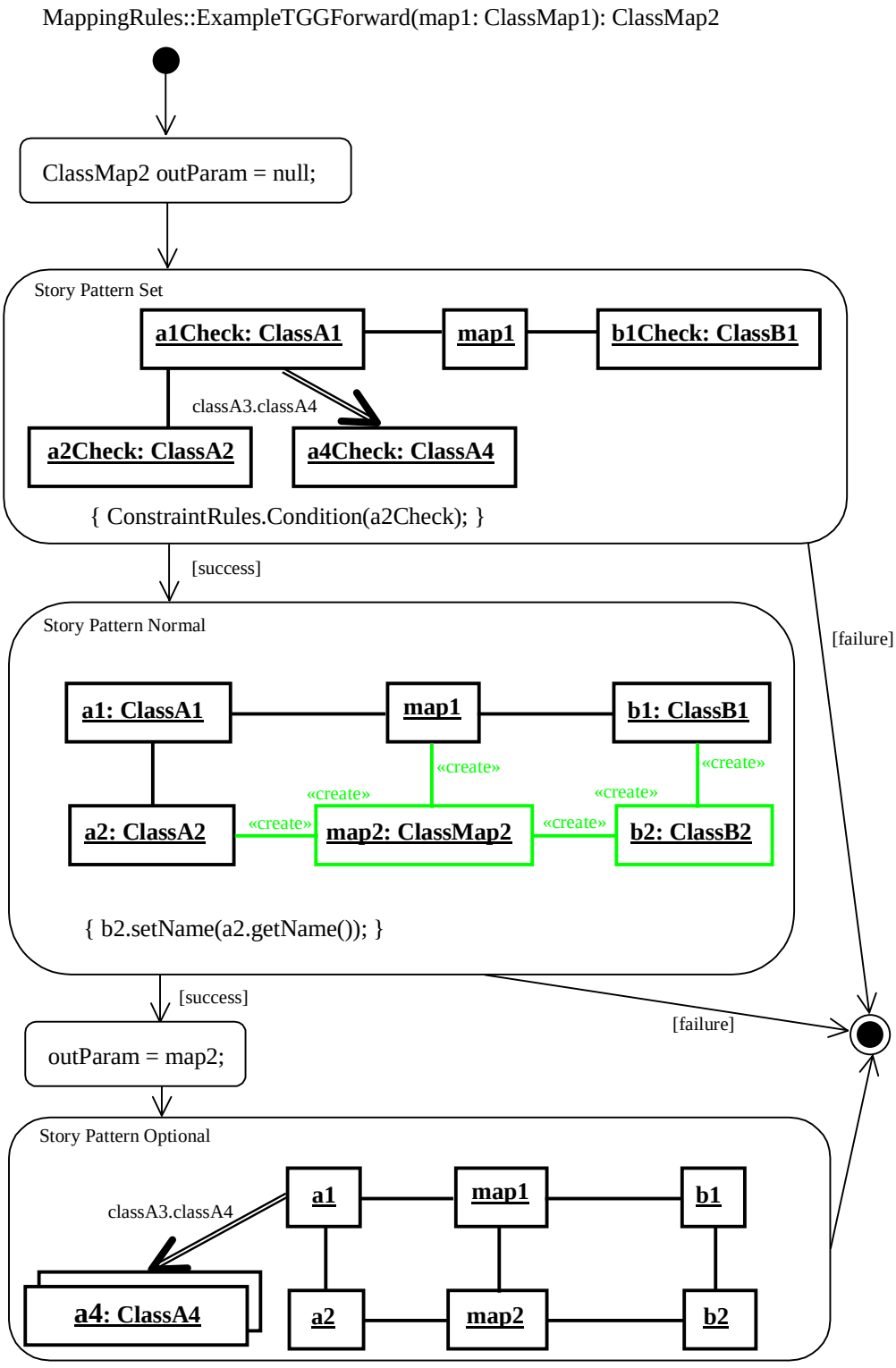

Abbildung 4.4: Storydiagramm der Vorwärtsregel

Da die Side-Constraints bereits in Java-Syntax vorliegen, können sie mit Hilfe der UML-Constraints ebenfalls bei der späteren Java Codegenerierung direkt übernommen werden.

Im Beispiel wird nur der Side-Constraint *LEFT* berücksichtigt, da es sich hier um eine Vorwärtsregel handelt. Für das im *Story Pattern Normal* erzeugte Objekt *b2* wird für das Attribut *name* der Wert des Attributs *name* von *a2* übernommen.

4.1.4 Einbindung der graphischen Pfadausdrücke

Die Möglichkeit der Angabe von graphischen Pfadausdrücken wurde im Rahmen dieser Studienarbeit nicht nur für TGG-Diagramme sondern auch für Storydiagramme geschaffen. Die in den Story Pattern verwendeten Pfade verlaufen zwischen UML-Objekten. In den TGG-Diagrammen verlaufen die Pfade zwischen TGG-Objekten, die eine Spezialisierung der UML-Objekte darstellen. Deshalb können auch die graphischen Pfadausdrücke im TGG-Diagramm durch Vererbung wiederverwendet werden. Aufgrund dieser Vererbungsbeziehungen können die Pfadausdrücke aus den TGG-Diagrammen bei der Generierung leicht in die Story Pattern übernommen werden.

In welches Story Pattern ein Pfadausdruck eingefügt wird, hängt von den TGG-Objekten ab, zwischen denen er definiert ist. Er wird in jedes Story Pattern eingefügt, in dem auch sein Start- und sein Zielobjekt vorhanden ist. In unserem Beispiel ist das Zielobjekt eine Menge und deshalb in *Story Pattern Normal* nicht vorhanden. Dort wird dann auch der Pfadausdruck nicht eingefügt. In die anderen Story Pattern wird er übernommen und verarbeitet.

Falls der angegebene Pfad nicht existiert, schlägt die Ausführung des jeweiligen Story Pattern fehl. Dies führt bei den Story Pattern *Set* und *Normal* dazu, dass die Ausführung des ganzen Storydiagramms fehlschlägt.

Die Übernahme der Pfadausdrücke in die Story Pattern erfolgt nur für Objekte, die dort dem Ausgangsgraphen angehören und nicht neu erzeugt werden müssen. Wird also ein Pfadausdruck in einer TGG-Regel für ein Startobjekt definiert, das mit *«create»* gekennzeichnet ist, so wird dieser nur in der Graphersetzungsregel berücksichtigt, in der dieses Objekt nicht mehr erzeugt werden muss. In unserem Beispiel wäre dies der Fall, wenn für *a2* ein Pfadausdruck definiert worden wäre. Er würde bei der Vorwärtsregel berücksichtigt, bei der Rückwärtsregel jedoch nicht.

4.2 Transformation der graphischen Pfadausdrücke in Java

Im Gegensatz zu den Constraints und den Side-Constraints können graphische Pfadausdrücke nicht direkt mit der bisher vorhandenen Codegenerierung in Java umgesetzt werden. Deshalb erfordert die Erweiterung der Storydiagramme um graphische Pfadausdrücke auch eine Erweiterung der Codegenerierung von FUJABA. Dazu wird im Folgenden zunächst der Mechanismus der Codegenerierung in FUJABA allgemein vorge-

stellt und anschließend die Einbindung der Generierung von Code für die Pfadausdrücke erläutert.

4.2.1 Codegenerierung in FUJABA

Der in [Moa02] beschriebene Codegenerierungsmechanismus von FUJABA basiert im Wesentlichen auf fünf Klassen, von denen alle anderen Klassen erben. Sie heißen *CodeGenStrategy*, *CodeGenStrategyHandler*, *CodeGenVisitor*, *CodeGenFactory* und *CodeGenFunction*. Das Klassendiagramm in Abbildung 4.5 zeigt die Zusammenhänge zwischen diesen Klassen und einige ihrer Unterklassen. Eine vollständige Abbildung aller Klassen der FUJABA Codegenerierung wäre zu groß und zu unübersichtlich, deshalb beschränkt sich dieses Diagramm auf die wichtigsten Klassen. Die grauen Klassen stehen stellvertretend für die Unterklassen der jeweiligen Klasse und dienen hier nur als Beispiel.

Der Codegenerierungsmechanismus ist so generisch modelliert worden, um später einmal Diagramme in verschiedene Programmiersprachen übersetzen zu können. Zur Zeit ist dies jedoch nur für die Sprache Java realisiert. Für andere Sprachen ist die Codegenerierung noch in der Entwicklung.

Für jede zu generierende Programmiersprache existiert in FUJABA ein XML-Dokument. In unserem Fall ist dies die Datei *javatarget.xml*. Sie enthält u.a. die Namen der benötigten Unterklassen von *CodeGenStrategy* und *CodeGenFunction*.

Die singleton Klasse *CodeGenFactory* hat die Aufgabe, die jeweilige Codegenerierung zu initialisieren und nutzt einen XML-Parser, um die Informationen aus dem zur Sprache passenden XML-Dokument einzulesen. Auf diese Weise können Objekte von den dort angegebenen Klassen erstellt werden.

Die Klasse *CodeGenStrategyHandler* hat Unterklassen, die jeweils für eine Klasse des in Abschnitt 3.1 beschriebenen abstrakten Syntaxgraphen (ASG) von FUJABA zuständig sind. Jede der Unterklassen überschreibt die abstrakte Methode *isResponsible(ASGElement asgElement)*, die angibt, ob das übergebene ASGElement ein Objekt der Klasse ist, für die der jeweilige Handler verantwortlich ist. Alle Handler sind über Links miteinander verbunden und bilden eine Chain of Responsibility [vgl. GHJV95], wie in Abbildung 4.5 zu sehen ist. *CodeGenStrategy* besitzt einen Link auf das erste Element dieser Chain und übergibt das zu bearbeitende *ASGElement* an diesen *CodeGenStrategyHandler*. Ist der nicht für das Objekt zuständig, wird es zum nächsten Handler in der Chain weitergereicht. Ist ein *CodeGenStrategyHandler* verantwortlich für das Objekt, so kann er dafür die passenden Aktionen ausführen. Jeder Handler besitzt ein *continueChain* Flag, das angibt, ob das Objekt nach der Verarbeitung noch weitergereicht werden soll, oder ob die Ausführung hier endet. Dieses Flag ist aus Performanzgründen standardmäßig auf *false* gesetzt und wird nur auf *true* gesetzt, wenn mehrere Handler für ein Objekt Aktionen durchführen müssen. In der XML-Datei wird definiert, ob ein Flag auf *true* gesetzt werden muss. Ein Beispiel für so einen Fall ist der *UpdateImportOfFileOOHandler*. Da er ein Objekt vom Typ *UMLFile* verarbeitet und

der *UMLFileOOHandler* ebenfalls, muss sein *continueChain* Flag auf *true* gesetzt werden.

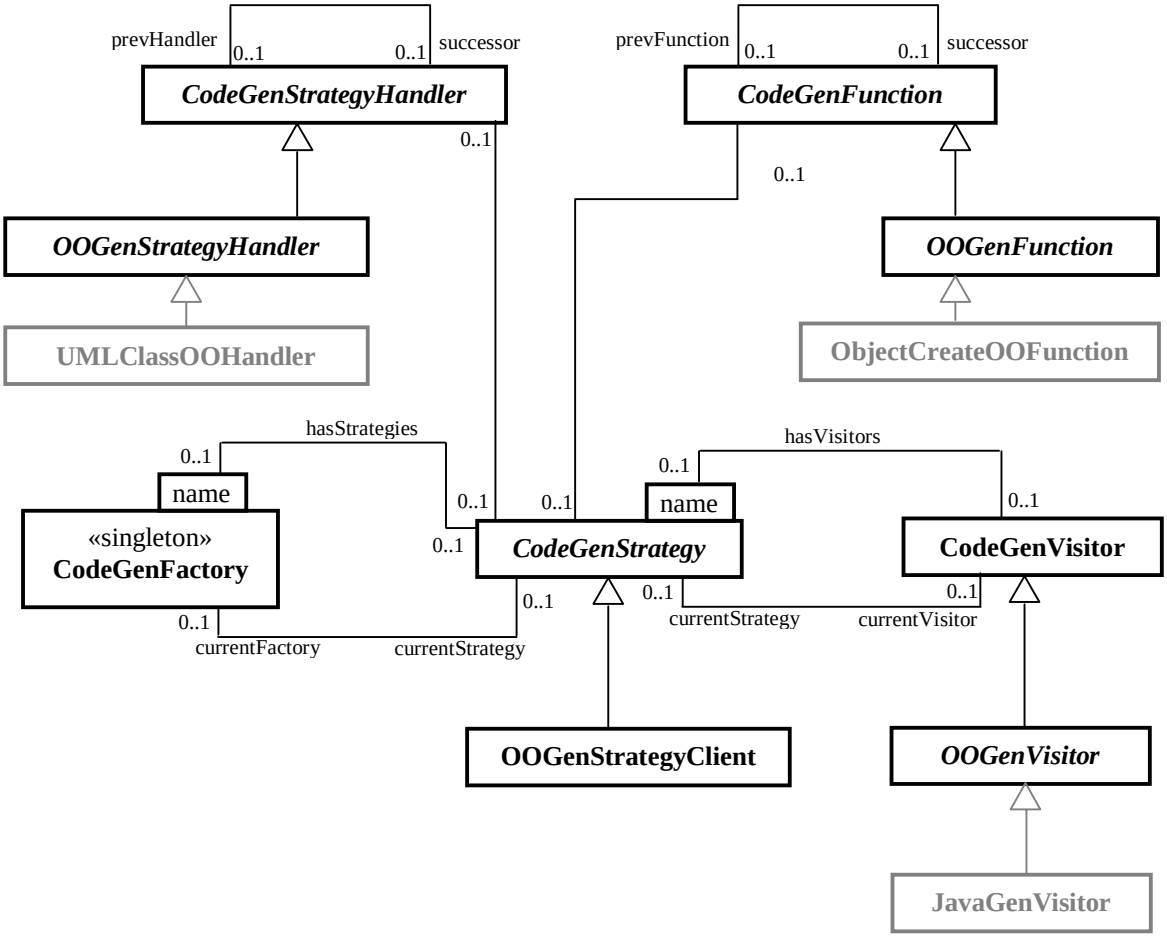

Abbildung 4.5: Codegenerierungsmechanismus

Der *CodeGenVisitor* hat die Aufgabe, die notwendigen Dateien für den erzeugten Code zu erstellen. Er muss außerdem einen Puffer bereitstellen, in dem der erzeugte Code zwischengespeichert wird. Die Klasse *CodeGenVisitor* enthält Methoden, die es erlauben Programmcode ohne spezielle Kenntnis der Syntax der Programmiersprache zu erstellen. Die Unterklassen der Klasse *CodeGenVisitor* überschreiben diese Methoden und erstellen das gewünschte Konstrukt in der Syntax der Programmiersprache. In unserem Fall ist dies die Klasse *JavaGenVistor*, die die Konstrukte in Java Code umsetzt. Ein Beispiel für so eine Methode ist *generateImportPackage (UMLPackage thePackage)*. Der *JavaGenVisitor* überschreibt diese Methode und erzeugt die syntaktisch korrekte Import-Anweisung für das Package.

In [Moa02] wird nur die Codegenerierung für Diagramme, die zur Beschreibung der statischen Struktur dienen, wie z. B. Klassendiagramme, vorgestellt. In dieser Arbeit ist

jedoch gerade die Codegenerierung für Diagramme wichtig, die dynamische Strukturen modellieren. Damit ist die Generierung von Code aus Storydiagrammen gemeint, da dort die Ergänzungen für die Behandlung der graphischen Pfadausdrücke vorgenommen werden müssen. Für diesen Zweck existiert die Klasse *CodeGenFunction*. Sie hat Unterklassen, die jeweils eine bestimmte Funktion für ein *ASGElement* implementieren. Es kann mehrere Funktionen für ein Element geben. Diese werden innerhalb der Handler über die *CodeGenStrategy* aufgerufen und können dann für das entsprechende Element ausgeführt werden.

Damit für ein *ASGElement*, für das ein Handler existiert, von mehreren *CodeGenFunctions* Code erzeugt werden kann, existiert für jeden Handler ein Flag *needToken*. Ist dieses auf *true* gesetzt, wird ein Objekt der Klasse *OOGenToken* zur Codegenerierung genutzt. Die Abbildung 4.6 zeigt den Zusammenhang zwischen *ASGElement* und *OOGenToken*. Auch hier wurden wieder aus Gründen der Übersichtlichkeit nur einige Beispielunterklassen in grau aufgeführt. Ein *OOGenToken* enthält eine Liste mit *OOStatements*. *OOStatements* sind allgemeine objektorientierte Sprachkonstrukte, die vom JavaGenVisitor in Java Code umgesetzt werden können. Um einem *OOGenToken* Code hinzuzufügen, kann das Token als Parameter übergeben und die entsprechenden *OOStatements* an seine Liste angehängt werden.

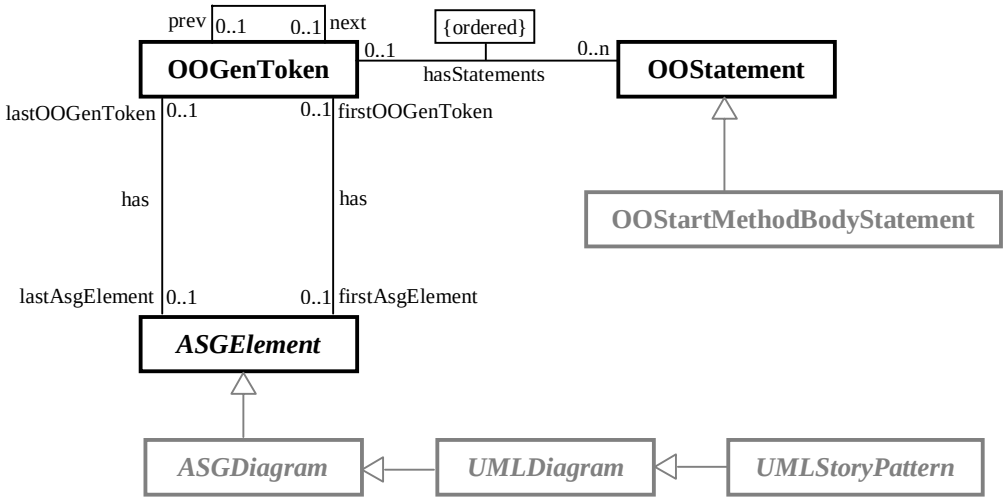

Abbildung 4.6: Beziehung zwischen ASGElement, OOGenToken und OOStatement

Wie im Klassendiagramm in Abbildung 4.5 zu sehen ist, erben alle aufgeführten Beispielklassen nicht direkt von den oben beschriebenen Klassen, sondern von Klassen mit dem Präfix *OOGen*. Dies sind Spezialisierungen der Klassen mit Präfix *CodeGen*, die zusätzliche Methoden für Codegenerierung in objektorientierte Sprachen bereitstellen. Sie sind ein weiteres Instrument zum möglichst generischen Aufbau der Codegenerierung in FUJABA.

4.2.2 Codegenerierung für graphische Pfadausdrücke

Da für die graphischen Pfadausdrücke ein neues *ASGElement*, nämlich *UMLPathExpr*, eingeführt worden ist, muss die Codegenerierung erweitert werden. Weil die Pfadausdrücke innerhalb von Story Pattern modelliert werden, muss der *UMLStoryPatternOOHandler* ergänzt werden. Dies ist ein Handler dessen *needToken* Flag auf *true* gesetzt ist.

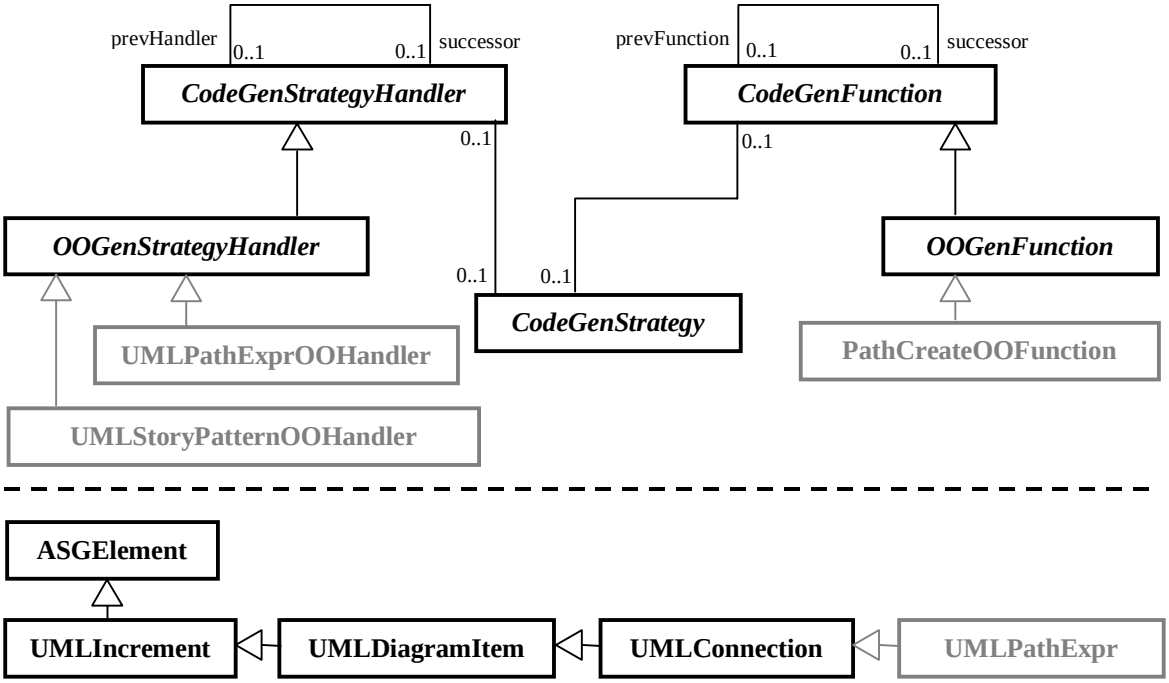

Abbildung 4.7: Einbindung von graphischen Pfadausdrücken in FUJABA

Die Klasse *UMLStoryPattern* erbt von *UMLDiagram* und verwaltet eine Reihe von Objekten der Klasse *ASGElement*, z.B. *UMLLink* und *UMLObject*. Für jedes dieser Objekte existiert ein Handler, der über den *OOGenStrategyClient* für die jeweiligen Objekte aufgerufen wird. Diese Handler bekommen auch das *OOGenToken* übergeben, dem sie mit *OOGenFunctions OOStatements* hinzufügen.

Für das *ASGElement UMLPathExpr* wurde im Rahmen dieser Arbeit die Klasse *UMLPathExprOOHandler* eingeführt, die für die Codegenerierung der graphischen Pfadausdrücke zuständig ist. Weiterhin wurde die Klasse *PathExprCreateOOFunction*, die von *OOGenFunction* erbt, erstellt. Sie wird innerhalb der Klasse *UMLPathExprOOHandler* genutzt, um dem übergebenen *OOGenToken* die entsprechenden *OOStatements* hinzuzufügen. Abbildung 4.7 zeigt, an welcher Stelle die hier beschriebenen neuen und veränderten Klassen (grau gekennzeichnet) in die Vererbungshierarchie von FUJABA eingebunden sind.

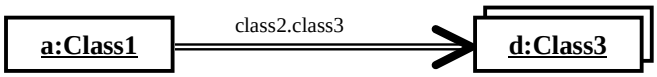

Abbildung 4.8: Beispiel für einen graphischen Pfadausdruck im Story Pattern

Mittels der beiden neuen Klassen *UMLPathExprOOHandler* und *PathExprCreateOO-Function* wird nun aus graphischen Pfadausdrücken Java Code erstellt. Aus dem Beispiel in Abbildung 4.8 kann so der in Abbildung 4.9 zu sehende Java Code erzeugt werden. Zur Erstellung des Pfadausdrucks in Java wird die Klasse *Path* benutzt. Sie stellt eine der Hilfsklassen dar, die in der Bibliothek *RuntimeTools.jar* enthalten sind, und wurde hier zur Erzeugung der Pfade wiederverwendet. Die Bibliothek ist im Fujaba-Installationspaket enthalten. Die Klasse *Path* implementiert das Java-Interface *Iterator*. Das Anlegen eines Pfades geschieht mit der Konstruktormethode *Path(Object start, String expr)*. Der erste Parameter *start* gibt das Startobjekt des Pfades an, der zweite Parameter *expr* enthält den Pfadausdruck. Die Syntax des hier erwarteten Pfadausdrucks ist identisch mit der in Abschnitt 3.2.3 beschriebenen Syntax. Die durch die Rollennamen im Pfadausdruck spezifizierten Nachbarobjekte werden beim Konstruktoraufruf durchlaufen. Mit dem Aufruf der Methode *next* auf der Instanz der Klasse *Path* können dann alle Objekte, die sich am Ende des Pfades befinden, innerhalb einer *while*-Schleife abgefragt werden.

```
try
{
  fujaba__Success = false ;

  // path expression
  d = new FHashSet ( ) ;
  fujaba__IterD = new Path (a, "class2.class3" ) ;
  while ( fujaba__IterD.hasNext () )
  {
    fujaba__TmpObjectClass3 = (Class3) fujaba__IterD.next () ;
    d.add (fujaba__TmpObjectClass3);
  } // while

         fujaba__Success = true ;
}
catch ( JavaSDMException fujaba__InternalException )
{
  fujaba__Success = false ;
}
```

Abbildung 4.9: Für einen graphischen Pfadausdruck erzeugter Java Code

Da Abbildung 4.9 nur ein Codeausschnitt ist, ist die Import-Anweisung des Packages *de.upb.tools.sdm* hier nicht zu sehen. Weil *d* eine Menge von Objekten im Diagramm in Abbildung 4.8 darstellt, ist die Variable *d* vom Typ *FHashSet*. *FHashSet* ist wie *Path* eine Hilfsklasse von FUJABA und entspricht einem *HashSet* in Java mit einigen zusätzlichen Methoden. Nachdem ein neues Objekt der Klasse *Path* mit den Parametern Startobjekt und Pfadausdruck erstellt worden ist, wird der erzeugte *Iterator* in einer *while*-

Schleife durchlaufen. Alle über den *Iterator* zugreifbaren Objekte werden dem *HashSet* hinzugefügt, nachdem eine Typkonvertierung in ihren ursprünglichen Typ, in diesem Fall *Class3*, vorgenommen wurde. Anschließend wird die boolsche Variable *fujaba_Success* auf *true* gesetzt. Kann jedoch das Pfadobjekt nicht erstellt werden, weil z.B. kein Objekt über den angegebenen Pfadausdruck erreicht werden kann, wird vorher eine *JavaSDMException* erzeugt und in dem *catch*-Block abgefangen. Dann schlägt die Ausführung des Story Pattern fehl, da die Variable *fujaba_Success* auf *false* gesetzt wird.

Kapitel 5: Prototypische Implementierung

Dieses Kapitel stellt den TGG Editor im Zusammenspiel mit FUJABA anhand eines Beispiels vor und zeigt den Einsatz der in den vorigen Kapiteln beschriebenen Erweiterungen.

Um den TGG Editor nutzen zu können, muss er als Plugin in FUJABA eingebunden werden. Anschließend wird ein neues Projekt erstellt. Innerhalb dieses Projekts wird ein neues Klassendiagramm erstellt, das die Struktur der zu integrierenden Dokumente abbildet. Das hier verwendete Beispiel ist von einer der in [Jah99] aufgestellten TGG-Regeln abgeleitet, die zur Schemamigration von einem relationalen Schema zu einem konzeptionellen Schema und umgekehrt dienen.

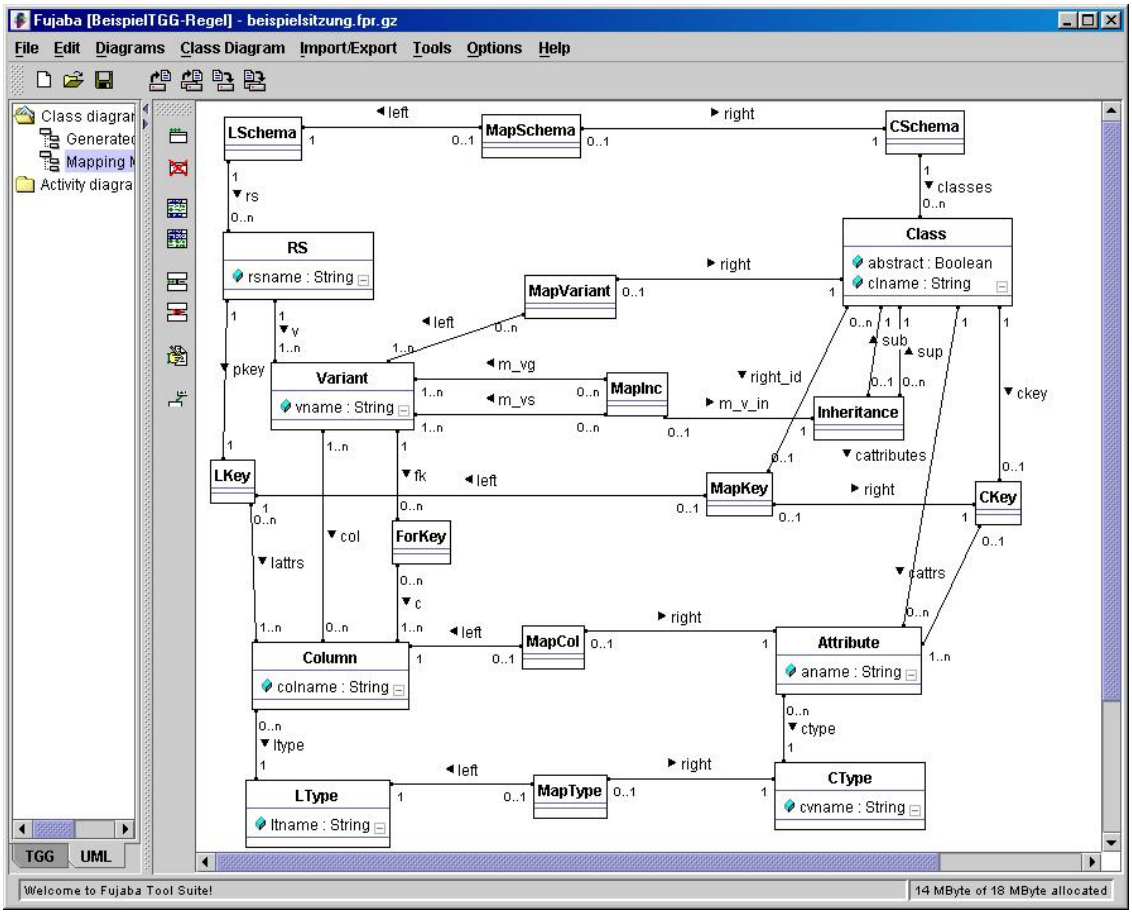

Abbildung 5.1: Dokumentenstruktur als Klassendiagramm in FUJABA

Das Klassendiagramm in Abbildung 5.1 zeigt die Struktur der drei Dokumente und ihre Beziehungen zueinander. Das relationale Schema, das in diesem Fall das logische Schema (Wurzelklasse *LSchema*) ist, und das konzeptionelle Schema (*CSchema*) werden durch das Integrationsdokument (*MapSchema*) aufeinander abgebildet. So können

z.B. Varianten einer Relation im relationalen Schema auf Klassen im konzeptionellen Schema abgebildet werden.

Nach der Erstellung des Klassendiagramms kann nun eine TGG-Regel erstellt werden. Dazu ruft man in FUJABA im Menü *Tools* unter dem Punkt *TGG Editor* den Unterpunkt *New TGG Diagram* auf, wie in Abbildung 5.2 zu sehen ist.

Abbildung 5.2: Erstellung einer neuen TGG-Regel

Nachdem man einen Namen für die Regel vergeben hat, wird automatisch der TGG Editor aufgerufen und man erhält eine neue Toolbar und ein neues Menü, das zur Erstellung von TGG-Diagrammelementen dient. Wie Abbildung 5.3 zeigt, können dort z.B. Objekte oder Constraints erstellt und gelöscht oder die Storydiagramme (Generate Code) generiert werden.

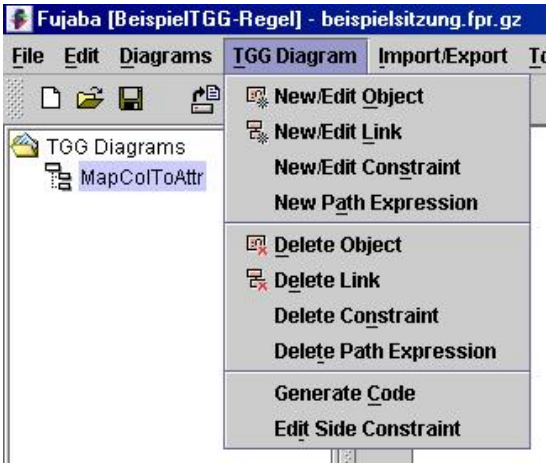

Abbildung 5.3: Menü zur Modellierung von TGG-Regeln

Im TGG Editor wird im Folgenden die TGG-Regel *MapColToAttr* (siehe Abbildung 5.4) erzeugt. Diese bildet die Attribute (*Columns*) des relationalen Schemas auf die Attribute des konzeptionellen Schemas ab. Abbildung 5.5 zeigt, wie die TGG-Regel *MapColToAttr* aussieht, wenn sie im TGG Editor modelliert wurde.

Die Erstellung von Objekten und Links wurde bereits in Abschnitt 3.1.2 erklärt. Deshalb soll hier darauf nicht mehr eingegangen werden. Die TGG-Regel *MapColToAttr* enthält Side-Constraints. Für die Erstellung ruft man im Menü *TGG Diagram* den Punkt

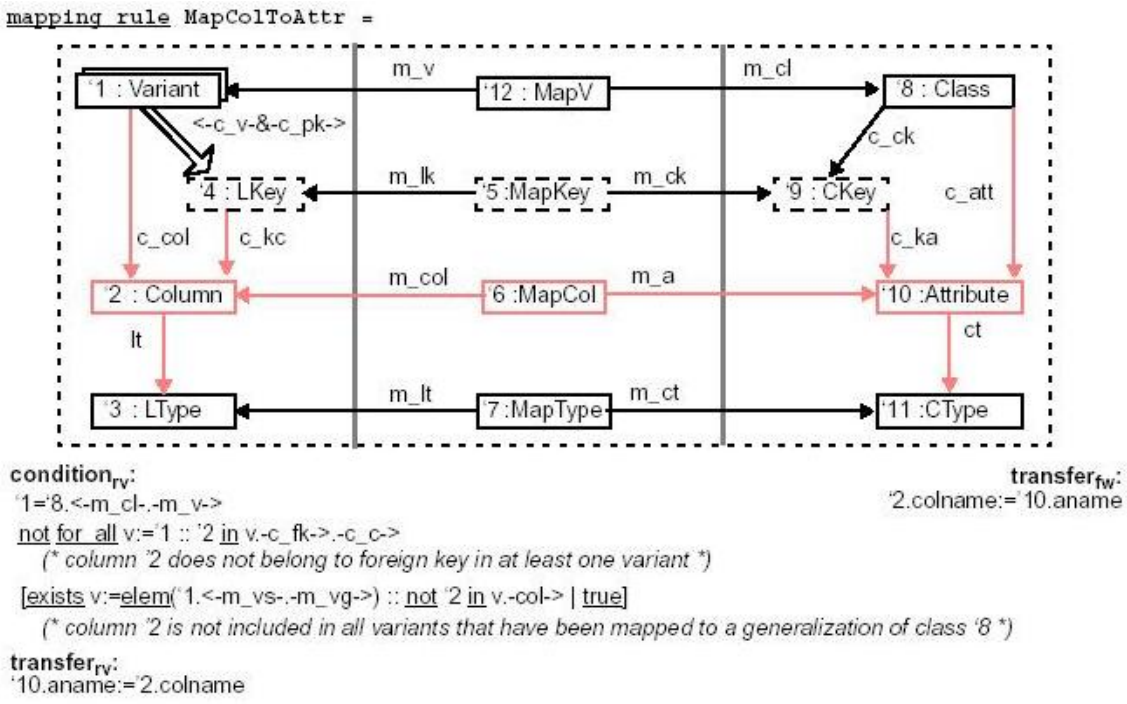

Abbildung 5.4: TGG-Regel MapColToAttr (Quelle: [Jah99, S.133])

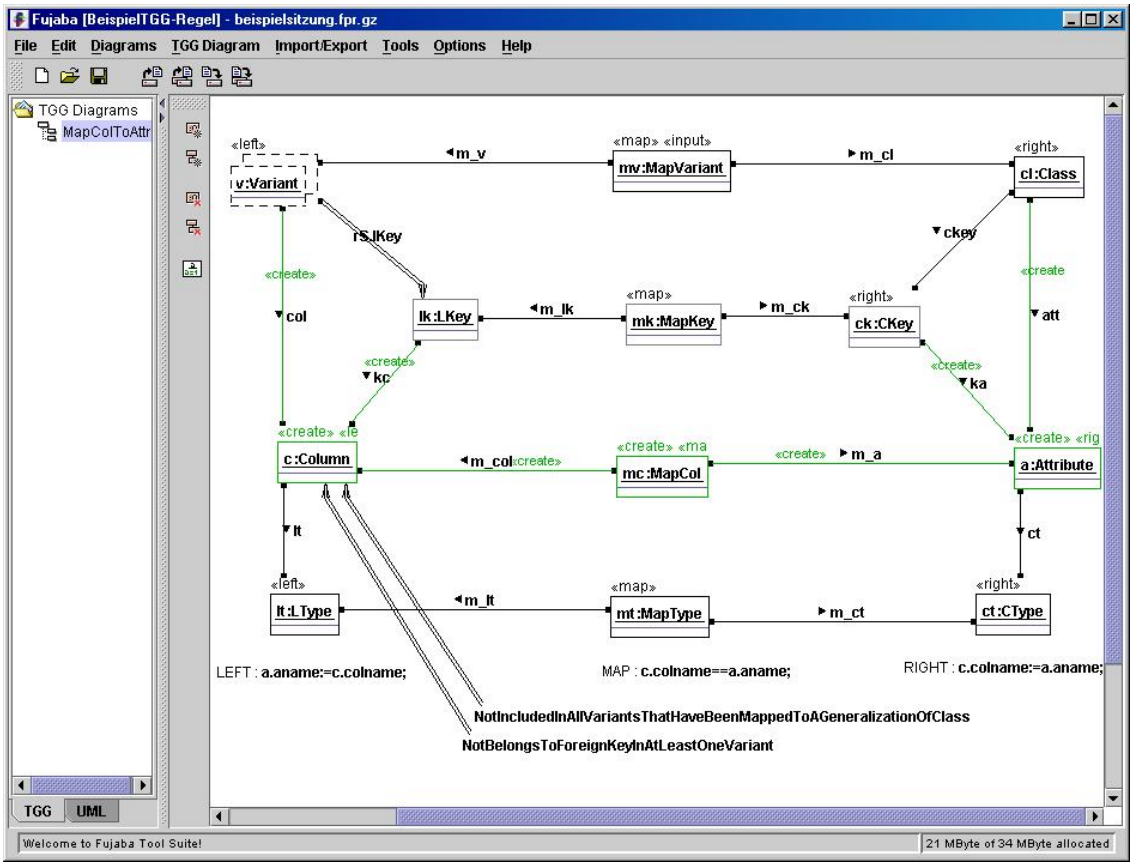

Abbildung 5.5: TGG-Regel MapColToAttr im TGG Editor

Edit Side Constraint auf. Dadurch öffnet sich ein Fenster, das alle bisher definierten Side-Constraints enthält und Editierungsmöglichkeiten bietet. Man kann vorhandene Side-Constraints ändern oder löschen und neue hinzufügen. Für jeden Side-Constraint lässt sich eine der Seiten LEFT, MAP oder RIGHT festlegen. Im TGG-Diagramm werden die so definierten Side-Constraints angezeigt. Im Beispiel werden die Side-Constraints genutzt, um die Namen der Objekte vom Typ *Column* und *Attribute* konsistent zu halten. Die Abbildung 5.6 zeigt das Fenster, in dem die Side-Constraints editiert werden können.

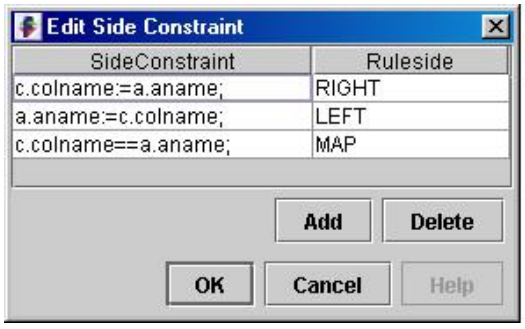

Abbildung 5.6: Editieren der Side-Constraints

Für das Hinzufügen eines graphischen Constraints muss vorher das TGG-Objekt, zu dem dieser Constraint angegeben werden soll, markiert sein. Dann kann entweder über das per Rechtsklick erscheinende Popup-Menü oder über das Menü *TGG Diagram* ein neuer Constraint erstellt werden. Nachdem man einen Namen vergeben hat, erscheint er im Diagramm mit einem Doppelpfeil auf das zugehörige Objekt. Mit einem Rechtsklick auf den Constraint gelangt man über das auftauchende Popup-Menü zum zugehörigen Storydiagramm, das man dann mit Inhalt füllen kann. In unserem Beispiel besitzt das Objekt *c* vom Typ *Column* die folgenden zwei graphischen Constraints:

1. NotIncludedInAllVariantsThatHaveBeenMappedToAGeneralizationOfClass
2. NotBelongsToForeignKeyInAtLeastOneVariant

Da alle Varianten (*Variant*) des relationalen Schemas auf Klassen (*Class*) im konzeptionellen Schema abgebildet werden, die in einer Vererbungshierarchie zueinander in Beziehung stehen, soll jedes Attribut des relationalen Schemas (*Column*) nur einmal auf ein Attribut (*Attribute*) des konzeptionellen Schemas abgebildet werden. Die anderen Klassen erhalten dieses Attribut durch Vererbung. Um diese Restriktion sicherzustellen, wird der erste Constraint benötigt.

Im Gegensatz zum relationalen Schema werden im konzeptionellen Schema keine Fremdschlüsselbeziehungen mehr benötigt, da dort Beziehungen z.B. durch Assoziationen modelliert werden können. Daher soll die Regel nur solche Attribute (Columns) auf die Attribute des konzeptionellen Schemas abbilden, die in mindestens einer der Varianten nicht einem Fremdschlüssel angehören. Der zweite Constraint beinhaltet diese Restriktion.

Durch einen Rechtsklick auf einen Constraint erhält man die Option *Go To Activity Diagram*. So gelangt man zu dem Storydiagramm, das die Semantik des Constraints festlegt. Der zweite Constraint aus unserem Beispiel wird durch ein Storydiagramm beschrieben, das in Abbildung 5.7 zu sehen ist. Das als Parameter übergebene Objekt *c* ist mit einem Objekt *v* über einen Link verbunden. Wird eine Struktur gefunden, in der *c* und *v* verbunden sind, aber ein Objekt vom Typ *ForKey* nicht existiert, so kann das Story Pattern angewendet werden und der Constraint ist erfüllt. Dies ist gleichbedeutend damit, dass *c* für mindestens eine Variante *v* nicht einem Fremdschlüssel *fk* angehören darf.

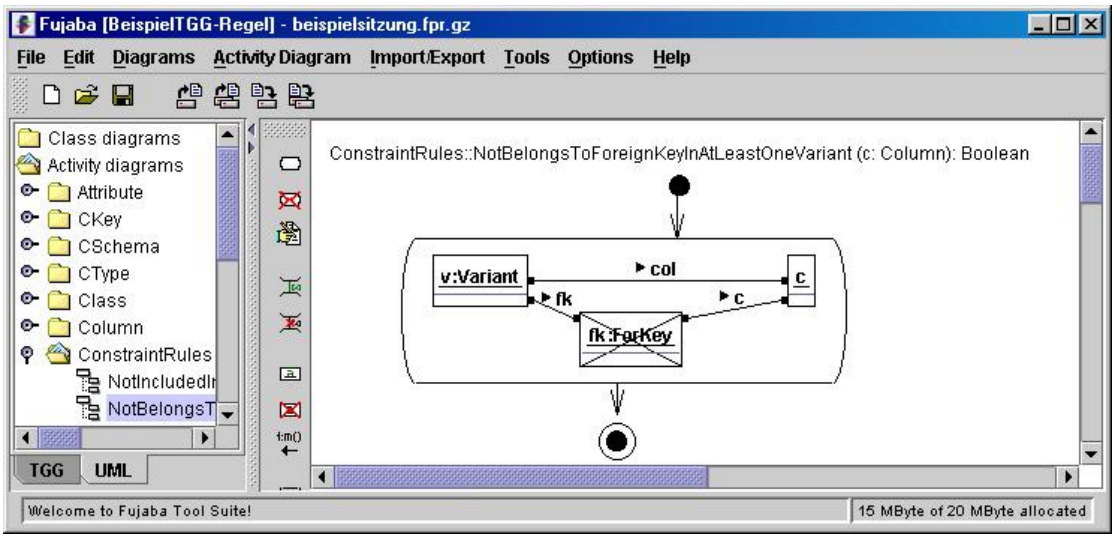

Abbildung 5.7: Storydiagramm, das graphischen Constraint beschreibt

In unserem Beispiel existiert auch ein graphischer Pfadausdruck. Um diesen einzufügen, markiert man zunächst das Objekt, von dem der Pfad ausgehen soll. Anschließend ruft man über das Menü *TGG Diagram* den Punkt *New Path Expression* auf und kann dort den Pfadausdruck sowie den Namen der Zielobjektmenge eingeben.

Abbildung 5.8: Erstellung eines graphischen Pfadausdrucks

Nach der Erstellung kann man das Zielobjekt ganz normal editieren. In unserem Beispiel wird über einen Pfad ein optionales Objekt gesucht. Da als Standardwert jedoch immer eine Menge von Objekten gefunden werden soll, muss man hier das Zielobjekt in ein optionales Objekt umändern.

Die Modellierung der TGG-Regel ist damit abgeschlossen. Um nun die Storydiagramme zu generieren, ruft man *Generate Code* im Menü *TGG Diagram* auf und wählt eine Strategie (in unserem Fall die Strategie Traditional Generation). Die erzeugten Storydiagramme finden sich dann in FUJABA im Ordner *Activity diagrams* unter *MappingRules*. Abbildung 5.9 zeigt einen Ausschnitt der Vorwärtsregel, die aus der im TGG Editor spezifizierten TGG-Regel generiert worden ist. Das *Story Pattern Set* zur Überprüfung der Menge *v* und der Constraints und das *Story Pattern Normal*, das alle Objekte bindet und neue Objekte und Links erstellt, sind dort zu sehen.

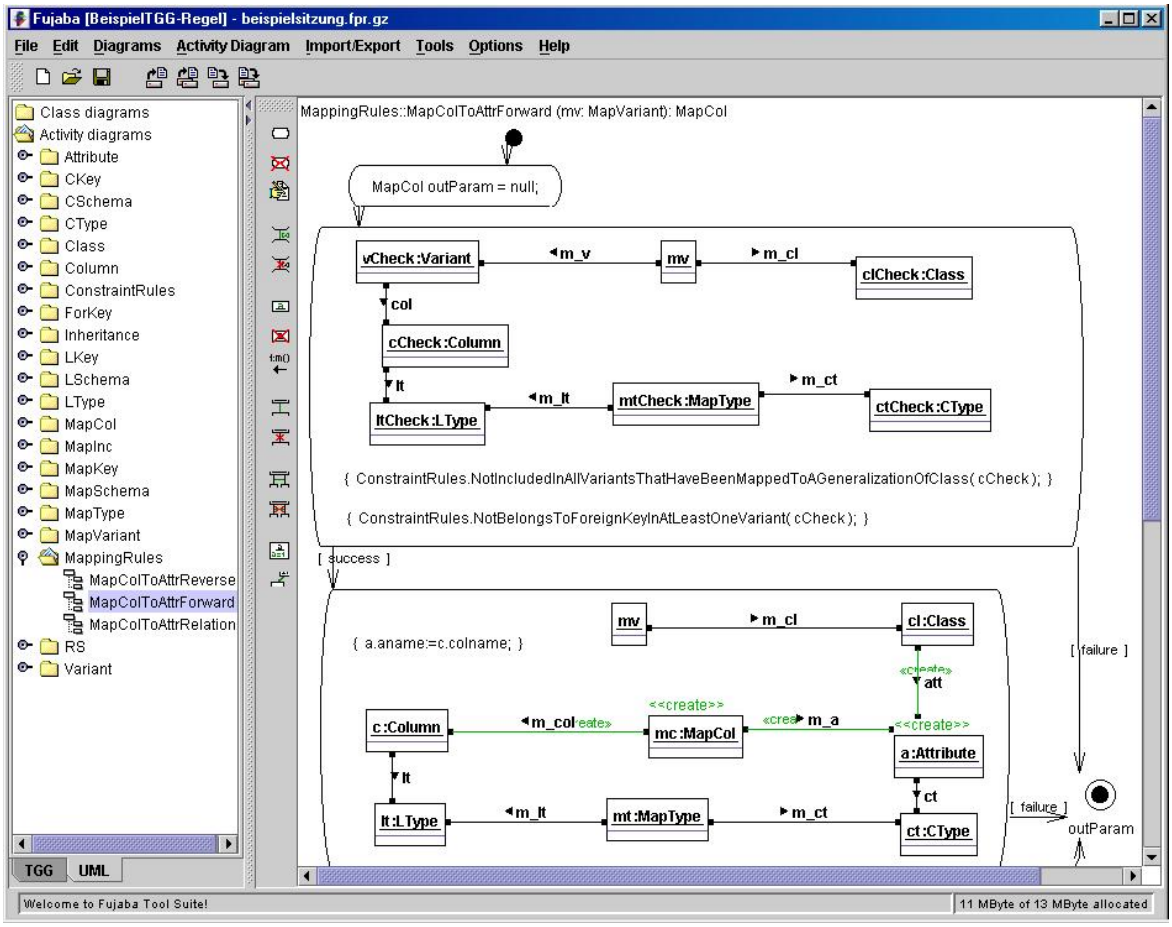

Abbildung 5.9: Storydiagramm der Vorwärtsregel (Ausschnitt)

Jetzt kann aus den erzeugten Diagrammen Java Code generiert werden. Dazu verwendet man die Option *Export Class To Java* oder *Export All Classes To Java* aus dem Menü *Import/Export*. Man kann sich auch mit Hilfe der Option *Edit Source Code* eine Vorschau anzeigen lassen und sich den Code mit dem Editor *mpEDIT* betrachten. In

Abbildung 5.10 kann man den Anfang der Klasse *MappingRules* sehen. Der Quellcode wird mit Hilfe von *mpEDIT* angezeigt und kann dort auch editiert werden.

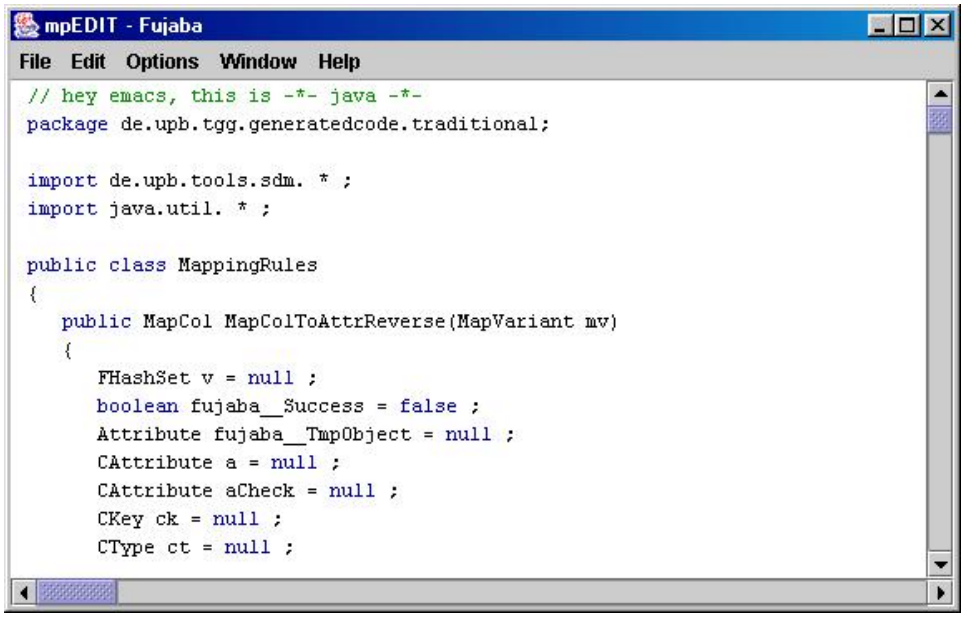

Abbildung 5.10: Vorschau auf Java Code mit mpEDIT

Kapitel 6: Zusammenfassung und Ausblick

In dieser Studienarbeit wurde ein Editor zur Erstellung von Konsistenzerhaltungsregeln um graphische Constraints und Pfadausdrücke erweitert und seine Codegenerierung an diese neuen Konstrukte angepasst. Die Constraints bieten die Möglichkeit, boolsche Bedingungen zu Elementen einer TGG-Regel in Form eines Storydiagramms zu definieren. Die Pfadausdrücke dienen einer verkürzten und damit übersichtlicheren Darstellung der im Editor definierten TGG-Regeln. Zur Transformation der Diagramme in Quellcode ohne zusätzlichen manuellen Aufwand dient die in dieser Arbeit erweiterte Codegenerierung.

Als Einführung in die Thematik wurde in Kapitel 2 zunächst die Entwicklungsumgebung FUJABA vorgestellt, deren Plugin-Konzept vom Editor genutzt wird. In diesem Zusammenhang wurde auch speziell auf Story Pattern und Storydiagramme eingegangen, die für die Codegenerierung des Editors und für die graphischen Constraints eine zentrale Rolle spielen. Story Pattern basieren auf Graph Grammatiken und sind eine Form von Regeln zur Veränderung von Graphstrukturen. Storydiagramme sind eine Erweiterung der UML-Aktivitätendiagramme um Story Pattern. Dabei werden die einzelnen Aktivitäten durch Story Pattern beschrieben.

Ebenfalls in dieser Arbeit wurde das Konzept der Tripel Graph Grammatiken erläutert. Sie stellen ein Instrument zur Konsistenzerhaltung zwischen zwei verschiedenen Dokumenten dar und nutzen dazu ein sogenanntes Integrationsdokument, das die zwei unterschiedlichen Arten von Dokumenten miteinander verknüpft und auf diese Weise eine Transformation von einem in das andere Dokument ermöglicht. In dem in dieser Arbeit erweiterten Editor können TGG-Regeln erstellt werden. Aus diesen Regeln können die drei Graphersetzungsregeln Vorwärtsregel, Rückwärtsregel und Konsistenzregel generiert werden.

In Kapitel 3 wurde der Aufbau des TGG Editors näher vorgestellt. Zunächst wurde die Einbindung des TGG Editors als Plugin in FUJABA beschrieben. Anschließend wurde der bisherige Funktionsumfang des Editors erläutert, der im wesentlichen aus der Erstellung von Objekten und Links bestand. Danach wurden die Konstrukte vorgestellt, um die der Editor im Rahmen dieser Studienarbeit erweitert wurde. Dazu gehören die graphischen Constraints, die graphischen Pfadausdrücke, die Side-Constraints und die austauschbare Generierung von Storydiagrammen aus TGG-Regeln. Dabei handelt es sich bei den Side-Constraints um eine Möglichkeit, die Attribute von den Objekten der verschiedenen Dokumente zueinander konsistent zu halten. Die austauschbare Generierung hingegen bietet die Möglichkeit, aus einer TGG-Regel nach unterschiedlichen Strategien entsprechende Graphersetzungsregeln in Form von Storydiagrammen zu erzeugen.

Der Weg zur Erstellung von Java Code aus TGG-Regeln wurde in Kapitel 4 erläutert. Dazu wurde zunächst am Beispiel der *Traditional Generation* gezeigt, wie die TGG-

Regeln in Storydiagramme umgesetzt werden. Anschließend wurde die Einbindung der vorgenommenen Erweiterungen in die Storydiagramme erläutert. Zum Schluss wurde erklärt, wie die Java Codegenerierung von FUJABA für die graphischen Pfadausdrücke erweitert wurde und dabei die Klasse *Path* zum Einsatz kommt.

Abschließend wurde in Kapitel 5 der weiterentwickelte Prototyp des TGG Editors vorgestellt und sein Funktionsumfang im Rahmen einer Beispielsitzung demonstriert. Dabei wurde auch auf die Bedienung des Editors näher eingegangen.

Die Erweiterung des TGG Editors um graphische Constraints und Pfade wurde innerhalb dieser Studienarbeit abgeschlossen. Allerdings sind durchaus noch einige Verbesserungen und Ergänzungen des TGG Editors vorstellbar. Zum Beispiel könnten bei den Pfadausdrücken anstatt der Rollennamen der Objekte die Namen der Links, die zwischen den Objekten liegen, den Pfadausdruck beschreiben. Diese Art der Definition wurde zum Beispiel bei [Jah99] für Pfadausdrücke verwendet. Wichtig wäre dann aber, dass auch eine Richtung angegeben wird, in der ein Link durchlaufen werden soll.

Außerdem könnte die Möglichkeit geschaffen werden, mehrere Pfade durch einen UND oder ODER-Operator zu verknüpfen. Es wäre auch ein transitiver Abschluss für die Pfadausdrücke vorstellbar, durch den alle auf dem Pfad liegenden Objekte gebunden würden. Dies würde die Mächtigkeit der Pfadausdrücke erhöhen.

Eine denkbare Erweiterung für die Definition der Struktur der Dokumente wäre, jedes Dokument in einem eigenen Klassendiagramm zu erstellen. Bisher muss in den TGG-Diagrammen für jedes Objekt angegeben werden, ob es zum linken, zum rechten oder zum Integrationsdokument gehört. Diese Angabe könnte dann wegfallen, da jedes Objekt eine Instanz einer Klasse ist, die eindeutig zu einem bestimmten Dokument gehören würde.

Weiterhin wäre es sicher noch wünschenswert, den TGG Editor um weitere Strategien zur Generierung von Graphersetzungsregeln aus TGG-Regeln zu ergänzen, um ihn noch mehr an die verschiedenen Anforderungen der Benutzer anzupassen. Ebenfalls sinnvoll könnte ein Einsatz des Editors in einem realen Projekt sein, um seine Alltagstauglichkeit zu überprüfen.

Abbildungsverzeichnis

Literaturverzeichnis

[BGN03] S. Burmester, H. Giese, J. Niere, M. Tichy, J.P. Wadsack, R. Wagner, L. Wendehals, A. Zündorf: *Tool Integration at the Meta-Model Level within the FUJABA Tool Suite*, Proc. of the Workshop on Tool-Integration in System Development (TIS), Helsinki, Finland, (ESEC / FSE 2003 Workshop 3), 2003.

[FNT98] T. Fischer, J. Niere, L. Torunski: *Konzeption und Realisierung einer integrierten Entwicklungsumgebung für UML, Java und Story-Driven-Modeling*, Diplomarbeit, Universität Paderborn, 1998.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*, Reading, MA : Addison-Wesley, 1995.

[Jah99] J. Jahnke: *Management of Uncertainty and Inconsistency in Database Reengineering Processes*, Dissertation, University of Paderborn, 1999.

[Lef95] M. Lefering: *Integrationswerkzeuge in einer Softwareentwicklungsumgebung*, Verlag Shaker, Aachen, 1995.

[LS96] M. Lefering, A. Schürr: *Specification of Integration Tools* , aus: M. Nagl (ed.): Building Thightly-Integrated (Software) Development Environments: The IPSEN Approach, LNCS 1170, Berlin: Springer Verlag, 1996; 324-334.

[Moa02] S. Moat: *Fujaba Code Generation for static UML Models*, work undertaken in connection with Projektgruppe SHUTTLE, University of Paderborn, Software Engineering Group, Department of Mathematics and Computer Science, 2002.

[Müh00] J. H. Mühlenhoff: *Integration und Konsistenzprüfung von XML Datenbeständen*, Diplomarbeit, Universität Paderborn, 2000.

[Pra71] T.W. Pratt: *Pair Grammars, Graph Languages and String-to-Graph Translations*, Journal of Computer and System Sciences, Volume 5, San Diego: Academic Press, 1971; 560-595.

[Sch94] A. Schürr: *Specification of Graph Translators with Tripel Graph Grammars*, In G. Tinhofer (ed.): Proc. WG 94 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science, Herrsching, Germany, Juni 1994. LNCS 903, Berlin, Springer Verlag, 1994; 151-163.

[UML03] Rational Software Corporation: *UML documentation version 1.5*, 2003, Online at http://www.rational.com, 2003.

[Wad98] J. P. Wadsack: *Inkrementelle Konsistenzerhaltung in der transformationsbasierten Datenbankmigration*, Diplomarbeit, Universität Paderborn, 1998.

[Wag01] R. Wagner: *Realisierung eines diagrammübergreifenden Konsistenzmanagement-Systems für UML-Spezifikationen*, Diplomarbeit, Universität Paderborn, 2001.