**Universität Paderborn**

Fachbereich 17 - Mathematik/Informatik
Arbeitsgruppe Softwaretechnik
Warburger Straße 100
33098 Paderborn

# Java Codegenerierung für Negative-Knoten und Multilinks einer Graphersetzungsregel

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

von

**Kan-Hung, Wan**
Himbeerweg 4
33689 Bielefeld

**Kan-Sing, Wan**
Himbeerweg 4
33689 Bielefeld

vorgelegt bei
**Prof. Dr. Wilhelm Schäfer**

Paderborn, August 2001

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 worden ist. Alle Ausführungen, die wörtlich oder sinngemä bernommen worden sind, sind als solche gekennzeichnet.

Paderborn,

Datum _____ Kan-Hung, Wan _____

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 worden ist. Alle Ausführungen, die wörtlich oder sinngemä bernommen worden sind, sind als solche gekennzeichnet.

Paderborn,

Datum _____ Kan-Sing, Wan _____

# Inhaltsverzeichnis

# Kapitel 1

# Einleitung

(Kan-Hung Wan, Kan-Sing Wan)

## 1.1 Motivation

Viele Probleme in der Informatik lassen sich durch einen Graphen beschreiben. Komplexe Sachverhalte können durch Objekte und Verbindungen zwischen ihnen einfach und übersichtlich modelliert werden. Graphen beschreiben aber nur eine statische Struktur und sind nicht in der Lage das dynamische Verhalten eines Systems zu beschreiben. In einem dynamischen System würde ein Graph also nur einen momentanen Zustand beschreiben. Für die Ablaufbeschreibung von dynamischen Prozessen wird deshalb eine Folge von Zuständen verwendet, die jeweils durch eine Übergangsregel genau spezifiziert sind. Genau dies tuen Graphersetzungsregeln.

Eine Graphersetzungsregel [1] führt eine Graphtransformation aus, die einen Teilgraph in einen anderen Teilgraph umwandelt. Sie wird durch zwei Graphen definiert, einen linken und rechten Graphen. Der linke Graph beschreibt die Anwendungsstelle in dem Wirtsgraphen, während der rechte Graph die Modifikationen definiert. Bedingungen und Modifikationen können durch entsprechende visuelle Konstrukte der Graphersetzungssprache in den Graphen spezifiziert werden.

Durch die visuellen Sprachkonstrukte der Graphersetzungssprache und die Unabhängigkeit der Graphen vom Code eignen sich Graphersetzungssysteme insbesondere für die Spezifikation von Softwaresystemen bei denen die

Manipulationen von Datenstrukturen im Vordergrund stehen. Algorithmen können dagegen mit einem Graphersetzungssystem nur schlecht spezifiziert werden.

Die zwei Entwicklungsumgebungen FUJABA [2] und PROGRES [3] verwenden für die Spezifikation der dynamischen Anteile eines Softwaresystems eine Graphersetzungssprache. PROGRES ist eine streng typisierte Sprache, die sowohl textuelle als auch visuelle Sprachelemente besitzt. Für die Graphschemata verwendet PROGRES gerichtete, attributierte, knoten- und kantenmarkierte (gakk) Graphen [4] [5]. Attribute sind nur an Knoten und nicht an Kanten erlaubt. PROGRES unterstützt folgende Merkmale: parametrisierte Graphersetzungsregeln, attributierte Knoten, Mengen, negative und optionale Knoten, Pfadausdrücke und semantisches Backtracking bei den Kontrollstrukturen für die Unterstützung von deklarativen Problembeschreibungen [6].

Fujaba verwendet *Story Diagramme* als Graphersetzungssprache für die Spezifikation von dynamischen Abläufen. Die Semantik der Story Diagramme basiert auf ihrem Vorläufer PROGRES. Ein Großteil der Sprachelemente in Fujaba ist deshalb identisch mit PROGRES. Story Diagramme wurden mit dem Ziel entwickelt eine bessere Integration in die Design- und Implementierungssprachen wie zum Beispiel UML [8] und Java [9] zu ermöglichen. Aufgrund der propritären graphischen Notation und der Backtracking Semantik des Kontrollflusses war dies in PROGRES nur schwer möglich. Für die Kontrollstrukturen der Story Diagramme wurde deshalb auf das Backtracking verzichtet, um eine einfache Umsetzung in eine OOP-Sprache zu ermöglichen. In Story Diagrammen werden die Kontrollstrukturen graphisch dargestellt statt textuell wie in PROGRES. Die graphische Notation der Kontrollstrukturen orientiert sich dabei an UML-Aktivitätsdiagramme. In Anlehnung an UML-Klassendiagramme unterstützt das Graphmodell von Story Diagrammen auch sortierte, qualifizierte Assoziationen und Aggregation.

Im Gegensatz zu PROGRES werden in Fujaba die zu-n Assoziationen und Mengenvariablen auf entsprechende Container-Strukturen abgebildet. Weil während der Teilgraphensuche häufig Elemente in einer Mengenvariablen eingefügt oder gelöscht werden, ist es sinnvoll die Zugriffszeiten für beide Operationen klein zu halten. Deshalb werden Mengenvariablen in Fujaba auf Bäume abgebildet. Die benötigte Zeit für das Einfügen oder Löschen von Elementen aus einem Baum ist in beiden Fällen O(log n) [7]. Für die zu-n Assoziation hängt die Wahl der Datenstruktur des Containers von dem Typ der Assoziation ab. Qualifizierte Assoziationen werden auf Hash-Tabellen abgebildet, während geordnete und sortierte Assoziationen Listen verwenden.

Hash-Tabellen erlauben über einen Schlüssel einen effizienten Zugriff auf ein entsprechendes Element. Sie sind deshalb besonders geeignet für die Implementierung von qualifizierten Assoziationen. Listen sind aufgrund ihrer einfachen Datenstruktur leicht zu sortieren, besitzen aber im schlechtesten Fall eine Zugriffszeit von O(n). Für die sequentielle Traversierung der einzelnen Elemente einer zu-n Assoziationen werden in Fujaba Iteratoren benutzt.

Ziel dieser Studienarbeit ist die Erweiterung der Java Codegenerierung der Entwicklungsumgebung Fujaba um Attributbedingungen für negative Knoten und Multilinks. Dies umfaßt ebenso eine Erweiterung der Zugriffsmethoden für Beziehungen zwischen Klassen, sowie eine Änderung an der optimierten Teilgraphensuche für Story Diagramme. Mit negativen Knoten können in einem Graphen Ausschlußbedingungen spezifiziert werden. Wird ein Knoten als negativ markiert, so darf der Knoten nicht existieren. Die spezifizierten Bedingungen dürfen nicht gelten damit die Graphersetzungsregel angewendet werden kann. Neben negativen Knoten unterstützt das Graphmodell von Story Diagrammen auch negative Links. Mit Hilfe von negativen Links können Beziehungen zwischen Objekten ausgeschlossen werden. Für Negative-Knoten und -Links gibt es in Kombination mit optionalen, attributierten und mengenwertigen Knoten, qualifizierten, zu-1 und zu-n Links viele Sonderfälle, die einzeln zu betrachten sind. Für jeden Sonderfall muß entsprechend der Semantik der richtige Code erzeugt werden.

Der statische Anteil eines Softwaresystems wird in der Entwicklungsumgebung Fujaba durch UML-Klassendiagramme modelliert, während für den dynamischen Anteil des Softwaresystems die Story Diagramme verwendet werden. In den Klassendiagrammen können zu-n Assoziationen über Bedingungen (Constraints) als sortiert oder geordnet spezifiziert werden. Diese Assoziationen werden auf spezielle Container und Zugriffsmethoden abgebildet. In den Graphersetzungsregeln ist es nun möglich über ein entsprechendes Sprachelement, ein Multilink, auf die einzelnen Elemente des Containers zuzugreifen. Es soll zum Beispiel möglich sein das erste oder letzte Element eines Containers oder den Nachfolger einer Variablen durch einen Multilink zu spezifizieren.

## 1.2 Struktur der Arbeit

In Kapitel 2 wird die Syntax und Semantik von Negativen-Knoten und Negativen-Links näher erläutert. Die verschiedenen Typen und Ausprägungen werden einzeln aufgeführt und besprochen. Eine Erwähnung finden eben-

so Sprachelemente die semantisch nicht möglich sind. Ebenso werden Begriffe eingeführt, die im Kontext von Negativen-Knoten und -Links in der Studienarbeit durchgehend verwendet werden.

In Kapitel 3 wird die Syntax und Semantik von Multilinks erklärt. In dem ersten Abschnitt wird auf die Semantik der unterschiedlichen Multilink-Typen eingegangen. Anschließend wird erklärt, was ein Multilink-Pfad ist. In dem letzten Abschnitt wird die Syntax der Multilinks vorgestellt.

In Kapitel 4 werden die Grundlagen für die Teilgraphensuche in Fujaba geschaffen. Auf das Kapitel bauen die zwei nachfolgenden Kapitel direkt auf. Es wird erklärt wie Fujaba die Teilgraphensuche intern durch Prioritätsklassen realisiert und nach welchen Strategien es vorgeht, um eine Optimierung der Suche zu erreichen. Durch die Optimierung wird der Suchaufwand verringert und die Suche insgesamt beschleunigt.

In Kapitel 5 wird aufbauend auf die Kapitel 2 und 4 die implementierte Teilgraphensuche speziell für Negative-Knoten und -Links erklärt. Für die in Kapitel 2 semantisch besprochenen Negativen-Typen wird gezeigt wie die entsprechende Codegenerierung zu realisieren ist. Die Codegenerierung für die unterschiedlichen Negativen-Typen bildet auch die Grundlage für ihre Einordnung in die entsprechenden Prioritätsklassen.

In Kapitel 6 wird die implementierte Teilgraphensuche der Multilinks vorgestellt. Zuerst werden nur die einzelnen Multilink-Typen betrachtet. Anschließend werden die komplexeren Multilink-Pfade behandelt, wobei hier für die Teilgraphensuche zum ersten Mal die unteren und oberen Schranken eingeführt werden. Nach der Betrachtung der Teilgraphensuche erfolgt eine Aufwandsabschätzung der Multilinks, was zu einer Einteilung der Multilinks in Prioritätsklassen führt.

## 1.3 Einführung in Story-Diagramme

Mit Hilfe von Story Diagrammen können die Methoden einer Klasse implementiert werden. Story Diagramme sind erweiterte UML-Aktivitätsdiagramme, deren Aktivitäten aus Java-Code oder aus Story-Pattern bestehen. Ein Story-Diagramm hat einen Startzustand und mindestens einen Endzustand. Zwischen dem Startzustand und den Endzuständen befinden sich die Aktivitäten, die über Transitionen miteinander verbunden werden. Die Transitionen realisieren den Kontrollfluß zwischen den Aktivitäten.

Es existieren unterschiedliche Typen von Transitionen, die den Kontrollfluß zwischen den Aktivitäten steuern. Boolesche Transitionen besitzen einen booleschen Ausdruck, der erfüllt sein muß, bevor darüber traversiert werden kann. Der boolesche Ausdruck kann Klassenmethoden und Attribute von gebundenen Objekten verwenden. In die *else*-Transition wird verzweigt, wenn alle bisherigen Transitionen nicht in Frage kommen. Die *success*-Transition wird verfolgt, wenn die Ausführung des Story Pattern erfolgreich war. Es darf nur eine *success*-Transition pro Story-Pattern vorhanden sein. Für den umgekehrten Fall wird die *failure*-Transition benutzt, das heißt bei einer nicht erfolgreichen Anwendung des Story Pattern.

Eine Sonderform der Aktivität Story-Pattern stellt das *For-All-Pattern* dar. In einem For-All-Pattern wird die Transformationsregel auf die Objektstruktur solange ausgeführt, solange ein passender Teilgraph gefunden werden kann. Die *each time*- und *end*-Transitionen sind spezielle Transitionen, die nur in Verbindung mit dem For-All-Pattern verwendet werden können. Die *each time*-Transition wird jedesmal bei einer erfolgreichen Regelanwendung traversiert. Ansonsten wird das For-All-Pattern über die *end*-Transition beendet.

## 1.4 Einführung in Story-Pattern

Story-Patterns [10] sind Graphersetzungsregeln, die das dynamische Verhalten eines Softwaresystems beschreiben. Sie spezifizieren die Modifikationen an Objekten und deren Beziehungen und damit die Objektstruktur des Softwaresystems. Ein Story Pattern besteht aus Variablen und Links, die eine spezielle Objektstruktur beschreiben auf die der Benutzer seine spezifizierten Modifikationen ausführen möchte. Bei der Ausführung des Story-Patterns werden die einzelnen Variablen mit Objekten aus der Laufzeitobjektstruktur gebunden. Die Objekte müssen dabei die an den Variablen spezifizierten Bedingungen erfüllen, damit sie gebunden werden können. Für die Abbildung des Story-Patterns auf die Laufzeitobjektstruktur gelten die Bedingungen des Graphisomorphismus. Soll es den Variablen gestattet werden, daß sie sich auch auf gleiche Objekte aus der Laufzeitobjektstruktur abbilden können (homomorphe Abbildung), so mußt das im Story-Pattern durch eine *maybe*-Klausel spezifiziert werden. In dieser Studienarbeit wird der Einfachheitshalber davon ausgegangen, daß die isomorphe Abbildung die Regel ist und die Anwendung der *maybe*-Klausel in einem Story-Pattern die Ausnahme bleibt. Ist das Story-Pattern nach der Ausführung vollständigt gebunden,

können die Modifikationen an der Laufzeitobjektstruktur ausgeführt werden. Die Laufzeitobjektstruktur kann durch das Löschen oder Erzeugen von Objekten und Links modifiziert werden. Hierzu werden die Variablen oder Links entsprechend mit «*create*» oder «*destroy*» markiert.

Die normale Variable wird durch ein Rechteck dargestellt, wobei der Name und Typ der Variablen darin eingetragen wird. Neben den normalen Variablen gibt es noch die optionalen und mengenwertigen Variablen. Optionale Variablen werden durch gestrichelte Rechtecke gekennzeichnet. Bei der Ausführung von Story-Pattern müssen optionale Variablen nicht unbedingt gebunden werden. Mengenvariablen werden durch zwei versetzt übereinander liegende Rechtecke dargestellt. Eine Mengenvariable speichert dabei alle Objekte, die an sie gebunden werden können. Da eine Menge auch leer sein kann, ist sie zudem optional. Operationen auf die Mengenvariable beziehen sich immer auf alle Objekte in dieser Menge.

Wie oben schon angedeutet werden zwischen gebundenen und ungebundenen Variablen unterschieden. Ungebundene Variablen werden bei der Ausführung des Story-Patterns mit einem passenden Objekt aus der Laufzeitobjektstruktur gebunden. Gebundene Variablen referenzieren also bereits auf ein Objekt aus der Laufzeitobjektstruktur. Um auf den Inhalt bereits gebundener Variablen zu referenzieren, wird eine Variable mit demselben Namen benutzt, wobei die Typangabe weggelassen wird. In Anlehnung an OOP-Sprachen, existiert die gebundene Variable *this*, die eine Referenz auf die aktuelle Klasseninstanz der spezifizierten Methode darstellt.

Ein Link zwischen zwei Variablen stellt eine Bedingung dar, die bei der Teilgraphensuche beachtet werden muß. Zwischen den zwei konkreten Objekten muß also ein entsprechender Verweis existieren. Ein Link wird durch eine einfache Linie abgebildet. Neben dem einfachen Link gibt es noch die optionalen und qualifizierten Links. Optionale Links werden durch eine gestrichelte Linie dargestellt. Der Link muß für eine erfolgreiche Ausführung des Story-Patterns nicht existieren. Beim qualifizierten Link wird der Schlüssel in eckigen Klammern angegeben.

## 1.5 Beispiel

In den nachfolgenden Kapiteln werden neue Sprachkonstrukte für die Story-Pattern eingeführt. Die Sprachelemente werden durch Beispiele veranschaulicht, die auf das folgende Klassendiagramm aufbauen:

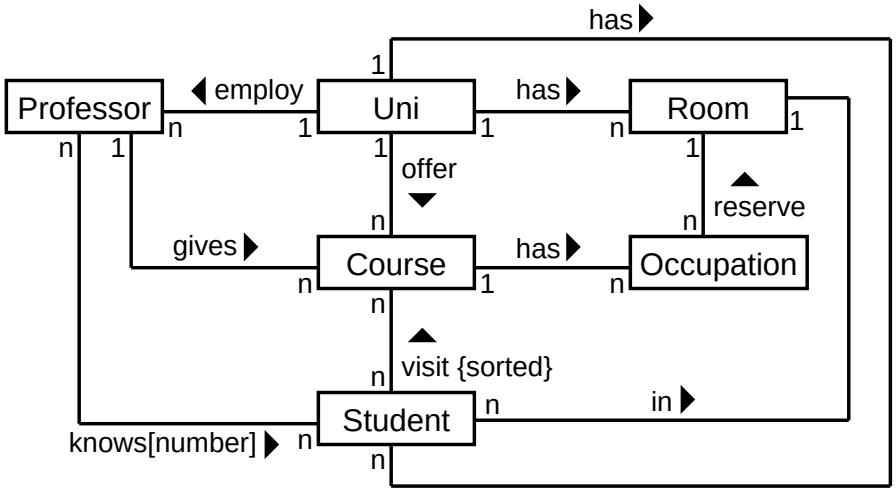



Abbildung 1.1: Uni-Klassendiagramm

Das zentrale Element in diesem Klassendiagramm bildet die Klasse *Uni*. Alle Klassen können über der Klasse *Uni* direkt oder indirekt erreicht werden. Ein Professor hält mehrere Vorlesungen und kennt die Studenten über ihre Matrikelnummer, was durch die Assoziationen *gives* und *knows[number]* modelliert wird. Ein Student kann mehrere Vorlesungen besuchen und sich in einem bestimmten Raum aufhalten. Dies wird im Klassendiagramm durch die Assoziationen *visit* und *in* modelliert. Die Studenten einer Vorlesung werden zudem alphabetisch nach ihrem Namen sortiert. Eine Vorlesung belegt nicht für den ganzen Tag einen Raum, sondern nur für eine bestimmte Zeitspanne. Eine Vorlesung besitzt deshalb mehrere Raumbelegungen, die jeweils einen Start- und Endzeitpunkt für die Reservierung eines Raumes festlegen. In dem Klassendiagramm wird die Raumbelegung durch die Klasse *Occupation* modelliert, das durch die Assoziation *reserved* auf den reservierten Raum verweist. Der Start-und Endzeitpunkt einer Raumbelegung wird in den zwei Attributen *start* und *end* der Klasse *Occupation* gespeichert

# Kapitel 2

# Semantik von Negativen-Knoten und -Links

(Kan-Hung, Wan)

## 2.1 Einführung

In diesem Kapitel werden die Sprachelemente *Negative-Knoten* und *Negative-Links* für Story-Pattern eingeführt. Die Negativen-Knoten und -Links spezifizieren Eigenschaften, die eine Variable nicht besitzen darf. Mit einem Negativen-Knoten ist es zum Beispiel möglich zu spezifizieren, daß eine Variable keine Beziehungen zu bestimmten Nachbarobjekte haben kann. Dazu werden die nicht erwünschten Nachbarobjekte einfach als Negative-Knoten gekennzeichnet. In Fujaba ist es möglich diese „negativen Eigenschaften“ auch ohne die neuen Sprachkonstrukte *Negative-Knoten* und *-Links* semantisch nachzubilden.

Aus pragmatischer Sicht ist es natürlich einfacher, wenn Fujaba die neuen „Negativen“-Sprachelemente schon unterstützt. Der Benutzer muß die Sprachelemente dann nicht durch zusätzliche Story-Diagramme nachbilden. Eine Unterstützung der Sprachelemente durch Fujaba verringert die Fehleranfälligkeit bei der Nachbildung der Negativen-Knoten und -Links durch den Benutzer. Verschiedene Benutzer haben eine unterschiedliche Vorstellung wie eine Negation in Story Pattern umzusetzen beziehungsweise zu modellieren ist was dazu führt, daß die Semantik der Negation auch entsprechend unterschiedlich ausfällt. Durch die Integration der Negativen-Sprachkonstrukte

in Fujaba kann eine einheitliche und konsistente Semantik unabhängig vom Benutzer garantiert werden. Sie führt auch zu einer besseren Übersicht, weil dadurch weniger Story-Patterns für die Nachbildung der Negation erstellt werden müssen. Als ein Sprachelement in Fujaba kann eine Syntaxüberprüfung der negativen Sprachelemente erfolgen und bei Fehler ein Hinweis an den Anwender ausgegeben werden. Weiterhin ist eine bessere Integration in die bestehenden Sprachelemente möglich, was eine Optimierung der negativen Sprachelemente für die Teilgraphensuche ermöglicht.

## 2.2 Negative-Knoten

Besitzt eine Variable in einen Story-Pattern einen Link, der auf einen negativen Knoten zeigt, so wird die Variable dadurch zusätzliche Bedingungen (Constraints) auferlegt, die sie zu erfüllen hat. Die zusätzlichen Bedingungen führen zu einer genaueren Spezifikation der Variable und schränken damit die Menge der möglichen Objekte in der Laufzeitobjektstruktur ein, die für die Bindung an der Variable in Frage kommen können. Die durch einen negativen Knoten definierten Constraints beschreiben Eigenschaften, die eine Variable nicht besitzen darf und werden für den weiteren Verlauf dieses Kapitels kurz als *negative Constraints* bezeichnet. Hängen mehrere Negative-Knoten an einer Variable, so werden sie einzeln unabhängig voneinander überprüft. Jeder Negativen-Knoten spezifiziert seine eigenen negativen Constraints für die Variable.

### Einfache Negative-Knoten

Negative-Knoten haben die Semantik, daß sie bei der Ausführung des Story-Pattern nicht existieren dürfen. Sie werden in der Notation durch ein Kreuz über den Knoten spezifiziert. Diese Semantik gilt aber nur für ungebundene Variablen. Besitzt eine Variable in einem Story Pattern einen Link zu einem negativen Knoten, der ungebunden ist, so darf die Variable keine Links zu Objekten haben, deren Typ durch den negativen Knoten definiert wurde. Mit anderen Worten, die Variable darf keine Links zu Instanzen vom Typ des negativen Knoten besitzen. Weil eine negative ungebundene Variable semantisch durch die Negation nicht existieren darf, kann die Variable als Konsequenz auch nie gebunden werden. In der Abbildung 2.1 wird die Semantik eines negativen Knoten anhand eines einfachen Beispiels erklärt. Die Semantik des negativen Knoten wird dabei mit Hilfe der schon bestehen-

den Sprachelemente des Story Pattern beschrieben. Der negative Knoten *s1* wird durch den booleschen Constraint „`{c1.noStuds()}`“ ersetzt. Die Variable *c1* kann nur erfolgreich gebunden werden, wenn der boolesche Constraint zu `true` ausgewertet wird. Im Constraint wird die Methode `c1.noStuds()` aufgerufen, die `true` zurück gibt, wenn die Vorlesung keine Links zu einem Studenten-Objekt besitzt. Ansonsten gibt die Methode `false` als Rückgabewert zurück.

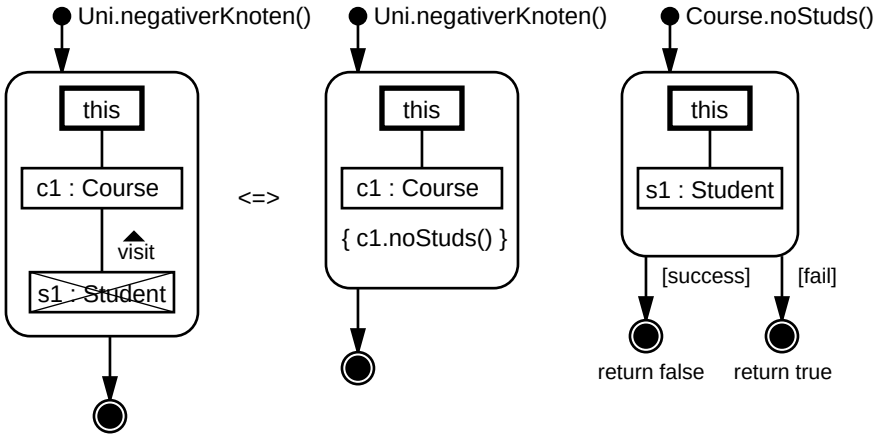



Abbildung 2.1: Semantik vom negativen Knoten

Folgende Begriffe werden in den weiteren Abschnitten im Kontext für Negative-Knoten verwendet und sollen hier näher erläutert werden: *Source-* und *Target-Variablen*. Als *Source-Variablen* werden Variablen bezeichnet, denen negative Constraints auferlegt werden. Das heißt eine Variable, die einen Link zu einem negativen Knoten besitzt wird als Source-Variable bezeichnet. *Target-Variablen* hingegen definieren die negativen Constraints für die Source-Variable. Ein negativer Knoten ist also eine Target-Variable. Die Target-Variable kann im Kontext von Negativen Knoten niemals gebunden werden. Die Erklärungen der Source- und Target-Variablen lassen sich ähnlich auf *Source-* und *Target-Objekte* übertragen. Mit einem Source-Objekt ist das Objekt gemeint, daß an der Source-Variable gebunden werden kann, wenn es dazu alle negativen Constraints erfüllt. Mit Target-Objekte sind alle Objekte gemeint, die einen entsprechenden Link zum Source-Objekt besitzen. Das Target-Objekt ist vom Typ des negativen Knoten beziehungsweise der Target-Variablen. Das Source-Objekt darf keine Target-Objekte besitzen, die die negativen Constraints verletzen, sonst darf es nicht an der Source-Variable gebunden werden.

Die Kennzeichnung eines Knoten beziehungsweise einer Variablen als negativ wird in der Regel nur auf ungebundene Variablen angewendet. Negative *gebundene* Variablen besitzen eine andere Semantik, als *ungebundene* negative Variablen. Weil die negativen gebundenen Variablen nur in Verbindung mit den *Attribut-Constraints* semantisch einen Sinn ergeben, werden sie erst weiter unten nach dem Abschnitt »Attribut-Constraints« behandelt.

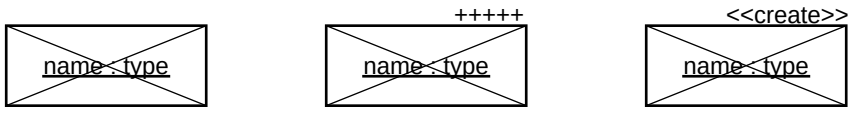



Abbildung 2.2: Die graphische Notation vom negativen Knoten

Es ist zulässig eine negative ungebundene Variable mit dem Modifizierer «create» zu markieren. Eine Markierung der Variable mit dem Modifizierer «create» führt zu einer Veränderung der Semantik des negativen Knoten. Wird bei der Ausführung eines Story-Pattern festgestellt, daß die mit «create» markierte negative Variable nicht existent ist, so wird ein neues Objekt vom Typ des negativen Knoten erzeugt. Existiert ein solches Objekt bereits bei der Ausführung des Story-Pattern, so wird kein neues Objekt erzeugt und die Teilgraphensuche wird ohne Erfolg beendet. In Abbildung 2.2 ist die graphische Notation der negativen Knoten abgebildet.

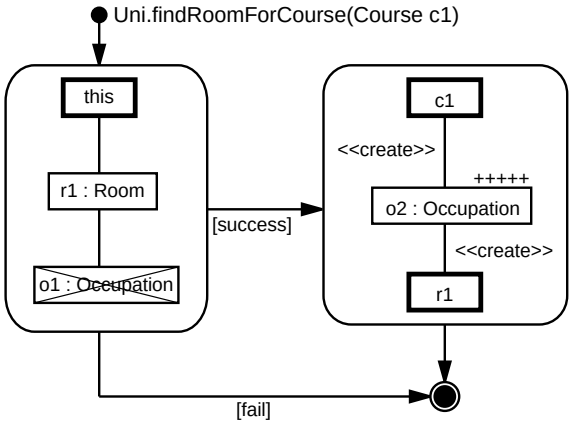



Abbildung 2.3: Beispiel mit negativem Knoten

Das Beispiel Story-Pattern in der Abbildung 2.3 soll die Anwendung der negativen Knoten etwas näher veranschaulichen. Das Story-Pattern spezifiziert die Methode `findRoomForCourse(..)` der Klasse *Uni*. Die Methode

sucht für eine Vorlesung, die als Parameter übergeben wird, einen Raum, der nicht belegt ist.

Damit bei der Ausführung des Story-Pattern nicht eine Instanz der Klasse *Room* an der Variablen *r1* gebunden wird, die schon durch einen anderen Kurs belegt wurde, muß die Source-Variable *r1* zusätzlich durch einen Constraint beschränkt werden. An der Variablen *r1* wird die negative Variable *o1* als Constraint „angehängt“. Als Konsequenz dürfen an der Source-Variablen *r1* jetzt nur Objekte aus der Laufzeitobjektstruktur gebunden werden, die keine Raumbelegungen haben.

### Attribut-Constraints

In Story-Pattern ist es möglich eine Variable um *Attributbedingungen* (Attribut-Constraints) zu erweitern. Mit der Hilfe von Attribut-Constraints können die Attribute der Variablen geändert oder durch zusätzliche Bedingungen beschränkt werden. Eine Attributbedingung setzt sich aus dem Klassenattribut, eine Vergleichsoperation und einem Ausdruck zusammen. Folgende Vergleichsoperationen können dabei verwendet werden: `=` (gleich), `!=` (ungleich), `<` (kleiner), `>` (größer), `<=` (kleiner gleich) und `>=` (größer gleich). Der Ausdruck ist entweder eine Konstante oder ein Methodenaufruf. Wichtig ist, daß am Ende ein boolescher Ausdruck entsteht.

Die Attributbedingungen lassen sich in zwei Klassen aufteilen, die *Pre*- und *Post*-Bedingungen. Attributbedingungen, die vor der Ausführung eines Story-Pattern gelten werden als *Pre*-Bedingungen bezeichnet. Analog werden Attributbedingungen, die nach der Ausführung gelten als *Post*-Bedingungen bezeichnet. Unter *Pre*-Bedingungen fallen alle Attribut-Constraints, die sich aus einer Vergleichsoperation zusammensetzen. Verändert ein Attribut-Constraint durch eine Zuweisung einen Attributwert, so fällt er unter den *Post*-Bedingungen. Vom Interesse für die Negative-Knoten sind nur die *Pre*-Bedingungen, weil sie genauso wie die Negativen-Knoten die Variablen mit zusätzlichen Bedingungen versehen und damit die Menge der möglichen Kandidatenobjekte, die an die Variable gebunden werden können, einschränken. Wenn in den folgenden Textabschnitten von Attributbedingungen die Rede ist, so sind damit immer die *Pre*-Bedingungen gemeint.

An eine Variable können beliebig viele Attributbedingungen angehängt werden. Wird eine normale Variable (nicht negative Variable) mit einer Attributbedingung verknüpft, so werden bei der Ausführung des Story-Pattern nur die Objekte aus der Laufzeitobjektstruktur betrachtet, die die Attri-

butbedingung erfüllen. Hängen an einer normalen Variable mehrere Attributbedingungen, so muß das zu bindende Objekt alle Attributbedingungen erfüllen, das heißt die Attribut-Constraints werden alle miteinander logisch `UND`-verknüpft.

Nachdem die Attribut-Constraints erläutert wurden, kann jetzt auf die Semantik von negativen gebundenen und ungebundenen Variablen mit Attributbedingungen näher eingegangen werden.

### Negativer ungebundener Knoten mit Attribut-Constraints

Ist eine negative Variable mit einer Attributbedingung verknüpft, so bedeutet dies semantisch, daß es *keine* Variable geben darf, die diese Attributbedingung erfüllt. Hängt an eine (Source-)Variable ein negativer Knoten mit einem Attribut-Constraint, so darf die Source-Variable durchaus Links zu Objekten vom Typ des negativen Knoten besitzen. Die Objekte dürfen aber die mit dem negativen Knoten verknüpfte Attributbedingung nicht erfüllen, das heißt sie müssen ungleich der Bedingung sein. Dieses „ungleich“ entspricht einer logischen Negation der Attributbedingung. Setzt sich die Attributbedingung aus einer „<“-Operation zusammen, so sind also alle Werte des Attributs erlaubt, die größer gleich dem rechten Ausdruck der Vergleichsoperation sind.

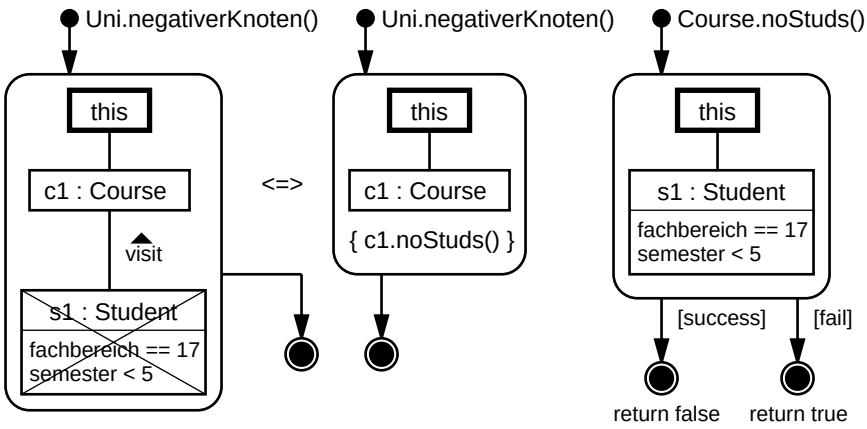



Abbildung 2.4: Semantik vom ungebundenen negativen Knoten mit Attribut-Constraints

Hängen an einem negativen Knoten mehrere Attributbedingungen, so darf die entsprechende Source-Variable semantisch keine Links zu Objekten vom Typ des negativen Knoten besitzen, die alle Attributbedingungen

erfüllen. Die mit dem negativen Knoten verknüpften Attributbedingungen werden also alle miteinander logisch `UND`-verknüpft. Analog den negativen Knoten mit nur einem Attribut-Constraint, sind von der Source-Variable ausgehend, alle Links zu Objekten vom Typ des negativen Knoten erlaubt, die dem negierten Ausdruck, aus miteinander logisch `UND`-verknüpften Attributbedingungen, entsprechen.

Besteht zum Beispiel ein negativer Knoten aus den drei Attributbedingungen $A_1, A_2$ und $A_3$, so sind alle Nachbar-Objekte vom Typ des negativen Knoten an der Source-Variable erlaubt, die folgende logische Bedingung erfüllen: $(\overline{A_1 \wedge A_2 \wedge A_3})$. Laut *De Morgan* sind die beiden logischen Formel $(\overline{A_1 \wedge A_2 \wedge A_3})$ und $(\bar{A}_1 \vee \bar{A}_2 \vee \bar{A}_3)$ äquivalent. Das heißt, daß an der Source-Variable alle Nachbar-Objekte vom Typ des negativen Knoten erlaubt sind, die schon *mindesten* eines der Attributbedingungen *nicht* erfüllen. In der Tabelle 2.4 ist die Semantik anhand eines Beispiels mit Hilfe der Sprachkonstrukte von Story Pattern beschrieben.

### Negativer gebundener Knoten mit Attribut-Constraints

Besitzt eine negative gebundene Variable eine oder mehrere Attributbedingungen (siehe Abbildung 2.5), so bezieht sich die Negation der Variable *nur* auf die Attributbedingungen. Für diese Attributbedingungen gilt der selbe Sachverhalt, wie für die Attributbedingungen an einer negativen ungebundenen Variable oben. Also mehrere Bedingungen werden logisch `UND`-verknüpft und es sind nur Objekte erlaubt, die mindesten eines der Attribut-Constraints nicht erfüllen. Weil in diesem Fall die Variable schon gebunden ist muß nur überprüft werden, ob eines der Attributbedingungen der Variable nicht erfüllt ist. Wird bei der Ausführung des Story-Pattern festgestellt, daß die negative gebundene Variable alle mit ihr verknüpften Attributbedingungen erfüllt, so schlägt die Ausführung des Story-Pattern fehl. Die Semantik wird durch ein einfaches Beispiel in der Abbildung 2.5 erläutert.

Eine negative gebundene Variable kann es nur mit Attributbedingungen geben. Denn eine negative gebundene Variable *ohne* Attributbedingungen führt zu einer widersprüchlichen Semantik und ist in einem Story-Pattern deshalb nicht erlaubt. Der Widerspruch ergibt sich dadurch, daß eine gebundene Variable aufgrund der Semantik als existent gilt, die Negation der gebundenen Variable fordert aber, daß sie nicht existieren darf. Eine Ausnahme bilden negative gebundene Variablen mit Attribut-Constraints, weil sich die Negation nicht auf die Existenz beziehungsweise „nicht Existenz“ der Variable bezieht, sondern auf die zu ihr gehörenden Attribut-Constraints.

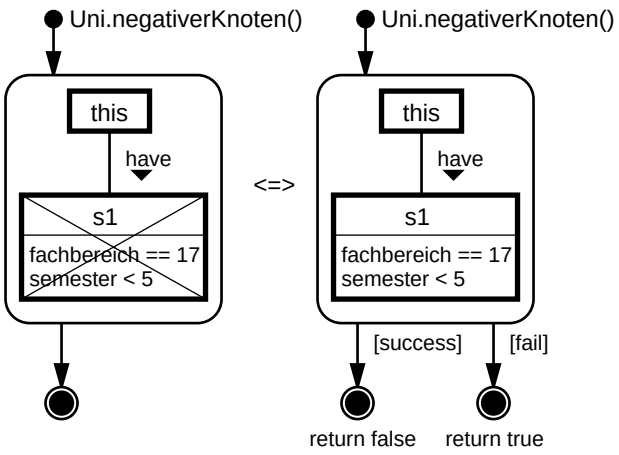



Abbildung 2.5: Semantik vom gebundenen negativen Knoten mit Attribut-Constraints

Durch die Negation wird also nur die Attributbedingung oder bei mehreren Attributbedingungen der Ausdruck aus `UND`-verknüpften Attributbedingungen negiert.

Das sich die Negation auf die Attributbedingungen und nicht auf die „nicht Existenz“ der Variable bezieht ist der wichtigste Unterschied im Vergleich zu einer negativen ungebundenen Variable. Diese Tatsache bringt folgende Konsequenzen mit sich:

- Eine negative gebundene Variable mit Attribut-Constraints ist von der Negation abgesehen semantisch äquivalent zu einer normalen gebundenen Variable.
- Eine negative gebundene Variable kann niemals eine Target-Variable werden. Das heißt sie kann keine negative Constraints für eine andere Variable definieren. Weil sich die Negation auf die Attributbedingungen bezieht und nicht auf die Variable kann sich die Negation auch nicht auf die an der Variable hängenden Links auswirken. Die Negation bleibt also „innerhalb“ des Knoten auf die Attribut-Constraints beschränkt.
- Es ist möglich eine negative gebundene Variable mit Attribut-Constraints als Source-Variable mit zusätzlichen negativen Constraints zu versehen. Die Variable darf also Links zu Negative-Knoten besitzen.

Das Beispiel in der Abbildung 2.6 erweitert die schon bekannte Metho-

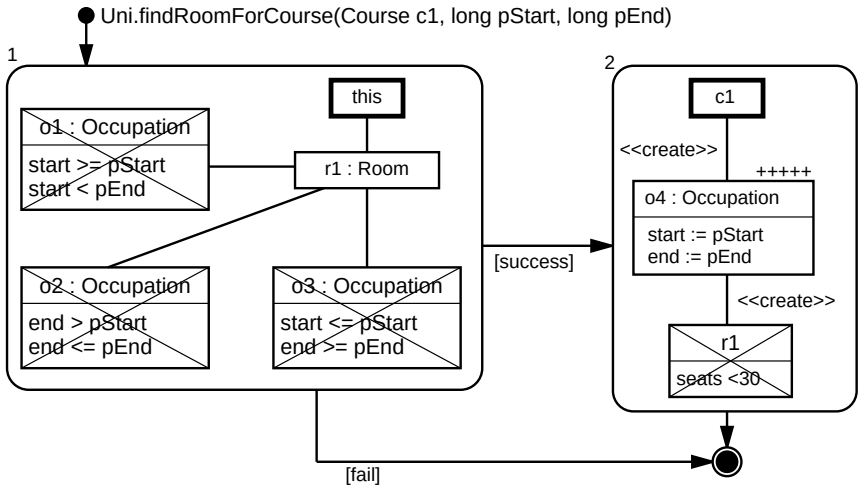



Abbildung 2.6: Beispiel mit negativem Knoten und Attribut-Constraints

de `findRoomforCourse(..)` der Klasse *Uni*. Eine Vorlesung belegt nicht für den ganzen Tag einen Raum, sondern nur für eine bestimmte Zeitspanne. Deshalb wird in der Klasse *Occupation* zwei neue Attribute *start* und *end* eingeführt, die den Anfangs- und Endzeitpunkt der Raumbelegung für eine Vorlesung modellieren. Die Methode wird um zwei Parameter *pStart* und *pEnd* erweitert, die den Anfangs- und Endzeitpunkt der Vorlesung festlegen. Die Argumente werden als Attribut-Constraints für die negativen Knoten *o1*, *o2* und *o3* verwendet. Wird bei der Ausführung des Story-Pattern nach einem passenden Raum für den Kurs *c1* in der Laufzeitobjektstruktur gesucht, so kommen nur Räume in Betracht, die keine Raumbelegungen haben, die sich zeitlich mit der Vorlesungszeit des Kurses *c1* überschneiden. Wurde ein passender Raum gefunden, so wird im zweiten Story-Pattern durch die negative gebundene Variable *r1* überprüft, ob der Raum mindesten 30 Sitzplätze besitzt. Ist die Anzahl der Sitze des Raumes kleiner als 30, so schlägt die Ausführung des Story-Pattern fehl.

### Negative-Knoten mit Vererbung

Eine Assoziation spezifiziert eine Beziehung zwischen zwei Klassen. Wegen der „is-a“ Beziehung der Vererbung, gilt diese Assoziation auch für alle Klassen, die von eines der beiden assoziierten Klassen direkt oder indirekt ableiten. Die Assoziation definiert also auch eine Beziehung zwischen zwei Vererbungshierachien, deren oberste Klasse in der Hierarchie einer der zwei as-

sozierten Klassen ist. Eine negative Variable kann als Typ eine beliebige Klasse in dieser Vererbungshierarchie haben. Weil eine abgeleitete Klasse eine Spezialisierung ihrer Oberklasse ist, kann die Vererbung als ein zusätzlicher Constraint angesehen werden, der auch bei einer negativen Variable zu beachten ist.

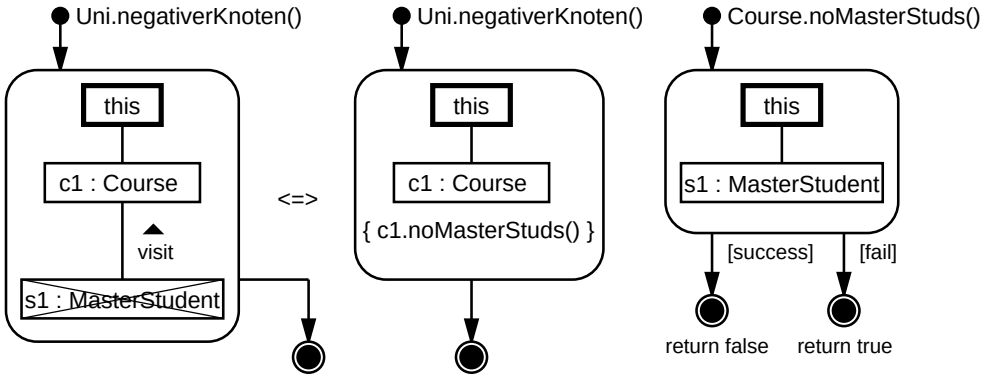



Abbildung 2.7: Semantik vom negativen Knoten mit Vererbung

Eine Source-Variable mit einem negativen Knoten darf nur Links zu Objekten besitzen, deren Klasse in der Vererbungshierarchie oberhalb der Klasse des negativen Knoten ist. Sie darf keine Links zu Objekten besitzen, die vom Typ des negativen Knoten sind oder direkt oder indirekt vom Typ des negativen Knoten abgeleitet haben. Besitzt die negative Variable Attributbedingungen, so beziehen sich die Attributbedingungen analog nur auf alle Objekte, die vom Typ des negativen Knoten oder in der Vererbungshierarchie unterhalb der Klasse des negativen Knoten sind. Die Semantik ist in der Abbildung 2.7 mit Hilfe von Story Pattern noch einmal beschrieben worden. Die Klasse *MasterStudent* im Beispiel ist eine Ableitung der Klasse *Student*.

### Negative-Knoten an einer optionalen Variable

Hängt an eine optionale (Source-)Variable ein negativer Knoten (siehe Abbildung 2.8), so dürfen die negativen Constraints für die Source-Variable nicht sofort überprüft werden. Eine optionale Variable muß laut Definition bei der Anwendung des Story-Pattern nicht gebunden werden beziehungsweise sein. Deshalb muß zuerst überprüft werden, ob die optionale Variable gebunden ist. Ist an der optionalen Variable ein Objekt aus der Laufzeitobjektstruktur gebunden, so können die negativen Constraints überprüft werden. Umgekehrt

ist die optionale Variable ungebunden, so ist eine Überprüfung der negativen Constraints nicht erforderlich.

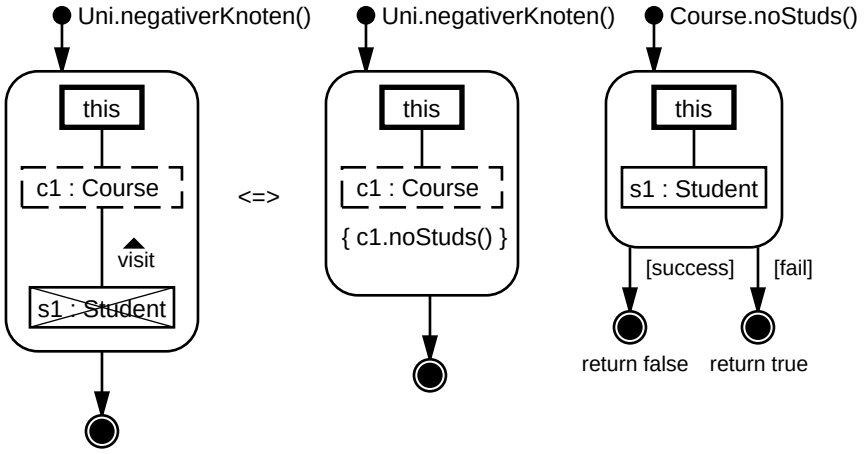



Abbildung 2.8: Semantik vom negativen Knoten an einer optionalen Variable

### Negative-Knoten an einer Mengenvariable

Eine Mengenvariable speichert die Menge aller Objekte, die an ihr gebunden werden können. Durch die Verknüpfung einer ungebundenen Mengenvariablen mit einem negativen Knoten können die möglichen Kandidaten, die in die Mengenvariable aufgenommen werden, durch die negativen Constraints zusätzlich eingeschränkt werden. Erfüllt ein möglicher Kandidtat nicht die negativen Bedingungen, so wird das Objekt nicht in die Mengenvariable aufgenommen. Nur Objekte, die alle negativen Constraints erfüllen kommen in die Menge rein. Das erste Beispiel in der Abbildung 2.9 beschreibt diese Semantik mit Hilfe von Story Pattern anhand eines einfachen Beispiels.

Ist die Mengenvariable gebunden, so ändert sich die Semantik für den negativen Knoten. Der Unterschied zwischen einer gebundenen und ungebundenen Mengenvariable ist, daß eine ungebundene Mengenvariable noch mit Objekten „gefüllt“ werden muß, während eine gebundene Mengenvariable bereits Objekte beinhaltet. Hängt an einer gebundenen Mengenvariable nun ein negativer Knoten so bedeutet dies semantisch, daß die Mengenvariable keine Objekte in seiner Menge besitzen darf, die die spezifizierten Bedingungen des negativen Knoten verletzen. Die Semantik wird im zweiten Beispiel in der Abbildung 2.9 nochmals durch Story Pattern verdeutlicht.

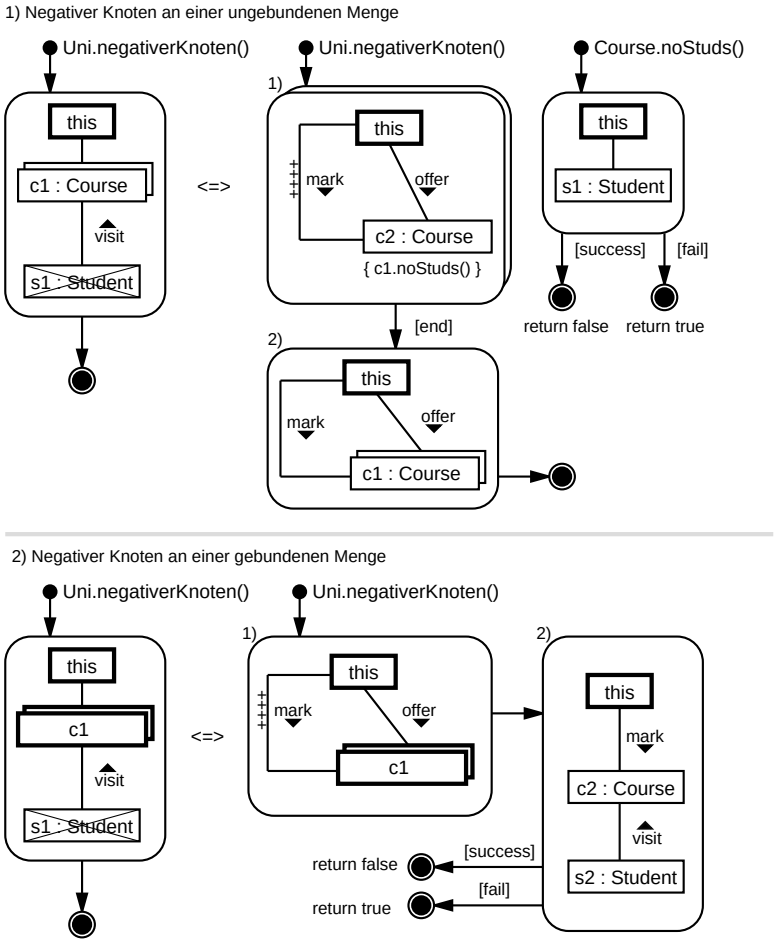



Abbildung 2.9: Semantik vom negativen Knoten an einer Mengenvariable

### Negative-Menge

Eine Mengenvariable die negativ ist hat die Semantik, daß die Menge leer sein soll. Das Source-Objekt darf also keine Links zu Objekte besitzen, die vom Typ der negativen Menge sind. Diese Semantik entspricht aber der Semantik des ungebundenen negativen Knoten. Eine negative Menge ist also semantisch nicht erforderlich, weil sie durch einen negativen Knoten abgebildet werden kann. Eine Menge kann aber auch leer sein und eine leere Menge existiert immer. Würde man die leere Menge als negativ spezifizieren so bedeutet das, daß es die leere Menge nicht geben darf, was aber ein Widerspruch ist. Eine negative Menge kann es also nicht geben.

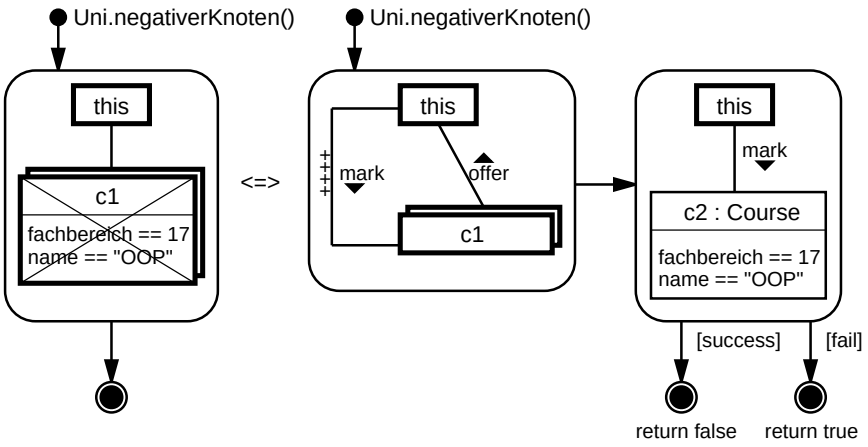



Abbildung 2.10: Semantik von einer negativen Menge

Eine negative Menge kann es nicht geben, wenn sich die Negation auf die Variable bezieht und die Mengenvariable keine Attributbedingungen besitzt. Ist eine Mengenvariable aber mit Attribut-Constraints verknüpft (siehe Abbildung 2.10), so kann sich die Negation auf die Attributbedingungen beziehen. Dieser Sachverhalt entspricht genau dem Fall wie bei den negativen gebundenen Variablen. In der ungebundenen Mengenvariable werden also nur Objekte aufgenommen, die die Attributbedingung nicht erfüllen oder wenn die Mengenvariable mit mehr als eine Attributbedingung verknüpft ist, mindesten eines der Attributbedingungen nicht erfüllen. Ist die Mengenvariable gebunden so darf die Menge keine Objekte besitzen, die alle Attributbedingungen erfüllen.

### Negative optionale Variable

Eine optionale Variable kann bei der Anwendung des Story-Pattern gebunden werden, muß es aber nicht sein. Sie kann also keinen Bindungsfehler verursachen und damit zum Abbruch der Teilgraphensuche beitragen. Ist eine optionale Variable negativ markiert (siehe Abbildung 2.14), so kann das aber zu einem Bindungsfehler und zum Abbruch des Story-Pattern führen. Wird bei der Regelanwendung die optionale Variable nicht gebunden, so ist die Sache noch in Ordnung und es kann zu keinen Fehler durch die Negation kommen. Ein Fehler tritt aber auf, wenn die optionale Variable gebunden wird. Die Negation sagt aus, daß es kein Objekt geben darf, es wurde aber gerade eines gebunden, was dann zu einem Bindungsfehler und zum Abbruch

des Story-Pattern führt. Eine negative optionale Variable widerspricht also der Semantik einer optionalen Variable und kann es deshalb nicht geben.

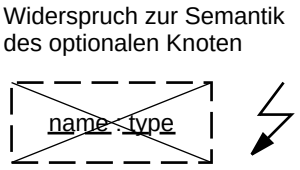



Abbildung 2.11: Negative optionalen Variable

Wie bei der negativen Menge kann es die negative optionale Variable aber mit Attribut-Constraints geben. Die Negation bezieht sich dann nur auf die Attributbedingungen und nicht auf die „nicht Existenz“ der optionalen Variable. Es wird dann nur ein Objekt an der optionalen Variable gebunden, das die Attributbedingung nicht erfüllt oder wenn die optionale Variable mit mehr als eine Attributbedingung verknüpft ist, mindesten eines der Attributbedingungen nicht erfüllt werden kann. Erfüllen alle möglichen Objekte aus der Laufzeitobjektstruktur die Attributbedingungen, so kann kein Objekt an der optionalen Variable gebunden werden. Es wird dabei kein Bindungsfehler erzeugt, was der Semantik einer optionalen Variable entspricht.

## 2.3 Negative-Links

### Einfache Negative-Links

Um zu modellieren, daß ein Link bei der Ausführung eines Story-Pattern nicht existieren darf, wird ein negativer Link verwendet. Negative-Links werden in der Notation durch einen Kreuz über den Link dargestellt. Sind also zwei Variablen durch einen negativen Link miteinander verbunden, so darf zwischen den beiden Variablen kein Link existieren.


Abbildung 2.12: Die graphische Notation von negativen Links

Es ist zulässig einen negativen Link mit «create» zu kennzeichnen. Wird bei der Anwendung eines Story-Pattern festgestellt, daß der Link nicht exi-

stiert, so wird ein neuer Link zwischen den Objekten erzeugt. Existiert der Link bereits, so wird kein neuer Link erzeugt und das Story-Pattern kann nicht erfolgreich ausgeführt werden. In der Abbildung 2.12 ist die graphische Notation der negativen Links abgebildet.

### Gleiche Semantik von Negativen-Knoten und -Links bei unterschiedlicher Syntax

Negative-Knoten und Negative-Links haben zwar eine unterschiedliche Syntax und Semantik, doch in einigen Bereichen sind sie semantisch äquivalent. Bevor erklärt wird unter welchen Bedingungen sie trotzt unterschiedlicher Syntax die selbe Semantik erzeugen, soll der Begriff *Target-Objekt* nochmals aufgegriffen werden. Aufgrund einer anderen Semantik von negativen Links gibt es einige Unterschiede für das Target-Objekt im Vergleich zu Negative-Knoten.

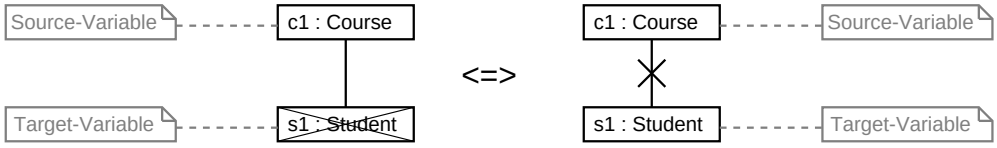



Abbildung 2.13: Beispiel semantischer Äquivalenz vom negativen Knoten und negativem Link bei unterschiedlicher Syntax

Werden die negativen Constraints durch einen negativen Knoten spezifiziert, so ist das Target-Objekt immer der negativen Knoten. Das Target-Objekt ist im Kontext für Negative-Knoten immer ungebunden. Werden die negativen Constraints durch einen negativen Link definiert, so kann das Target-Objekt im Kontext von negativen Links sowohl gebunden als auch ungebunden vorkommen. Ein negativer Link kann zwischen gebundenen, ungebundenen oder beiden Variablen modelliert werden.

Im folgenden werden die Bedingungen für die negativen Links aufgezählt, unter denen sie die selbe Semantik erzeugen wie die negativen Knoten:

- Die Target-Variable des negativen Link muß ungebunden sein. Sie darf nicht gebunden werden.

- Die Target-Variable darf also keine Links zu anderen Variablen haben außer dem negativen Link zu der Source-Variable.

- Der negative Link der Target-Variable darf nicht qualifiziert sein.

Durch die ersten beiden Bedingungen kann die Target-Variable des negativen Link niemals gebunden werden, weil sie nur einen Link zu einer Variable besitzt und dieser ist negativ. Damit entspricht sie genau der Target-Variable für Negative-Knoten, die auch nie gebunden werden kann. Unter diesen Bedingungen sind die negativen Links und negativen Knoten semantisch äquivalent. Im Kontext des negativen Knoten ist die Target-Variable negativ und die Source-Variable darf keinen Link zu der Target-Variable besitzen. Im Kontext des negativen Link ist die Target-Variable zwar nicht negativ, aber weil der Link zu der Target-Variablen negativ ist, darf die Source-Variable keinen Link zu der Target-Variable besitzen. Beide Fälle, negative Knoten und Negative-Links, drücken also unter diesen Bedingungen dasselbe aus. Ein Beispiel für die semantische Äquivalenz von negativen Knoten und negativen Links ist in der Abbildung 2.13 dargestellt.

Wird die erste oder letzte Bedingung verletzt, das heißt die Target-Variable ist gebunden oder der negative Link ist qualifiziert, so wird eine Semantik erzeugt, die sich von der Semantik der negativen Knoten unterscheidet. Die Semantik für einen negativen Link an einer gebundenen Target-Variable und für einen qualifiziertem negativen Link wird in den folgenden Abschnitten erklärt. Für den Fall das die zweite Bedingung nicht gilt, daß also die Target-Variable mehr als einen Link besitzt, sei auf das Kapitel 5.4 verwiesen. Bis zum Ende dieses Kapitels wird angenommen, daß die Target-Variable nur einen Link besitzt, der mit dem Source-Objekt verbunden ist.

### Negative-Links mit gebundenen Variablen

Ist bei einem negativen Link die Target-Variable gebunden, so ergibt sich daraus im Vergleich zu einer ungebundenen Target-Variable, eine andere Semantik. Semantisch heißt das, daß es von der Source-Variable aus keinen Link zu dem Objekt geben darf, das an der Target-Variablen gebunden ist. Das Source-Objekt darf durchaus Links zu Objekten vom Typ der Target-Variablen haben, nur eben keinen Link zum gebunden Target-Objekt. Die Abbildung 2.14 verdeutlicht diese Semantik anhand eines Beispiels mit den schon bestehenden Sprachelementen des Story Pattern.

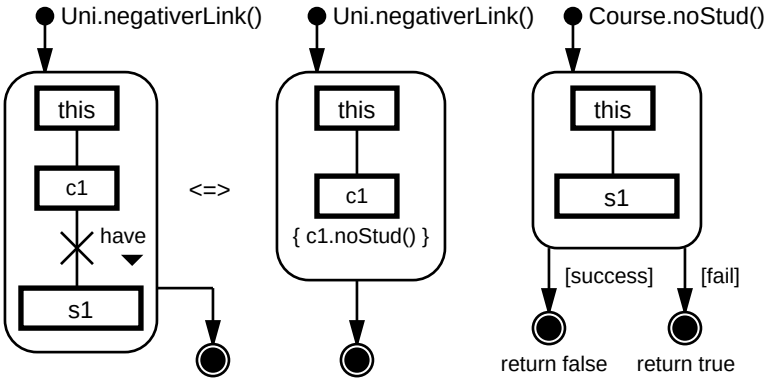



Abbildung 2.14: Semantik vom negativen Link mit einer gebundenen Variablen

### Qualifizierte Negative-Links

Es ist möglich einen negativen Link zwischen zwei Variablen zu qualifizieren. Ein qualifizierter Link assoziiert einen Schlüssel mit einem Objekt oder eine Menge von Objekten. Mit Hilfe des Schlüssels kann auf die assoziierten Objekte bezug genommen werden. Durch die Negation des qualifizierten Links werden also nur auf die Objekte bezug genommen, die mit dem Schlüssel in Verbindung stehen. Was ein qualifizierter negativer Link semantisch zu bedeuten hat, hängt davon ab, ob die Target-Variable gebunden ist oder nicht.

Ist die Target-Variable ungebunden, so haben qualifizierte Negative-Links die Semantik, daß die Source-Variable unter dem angegebenen Schlüssel keine Objekte besitzen darf, die durch einen Link mit ihr verbunden sind. Bei einer gebundenen Target-Variable darf die Source-Variable keinen Link zu dem gebundenen Objekt besitzen, das mit dem angegebenen Schlüssel assoziiert wurde. Das heißt die Source-Variable darf unter dem Schlüssel keine Objekte besitzen oder nur Objekte die vom gebundenen Objekt verschieden sind. Die Source-Variable kann das gebundene Objekt aber unter einem anderen Schlüssel, der vom gegebenen Schlüssel verschieden ist, besitzen. In der Abbildung 2.15 ist die Semantik für eine gebundene und ungebundene Target-Variable beschrieben.

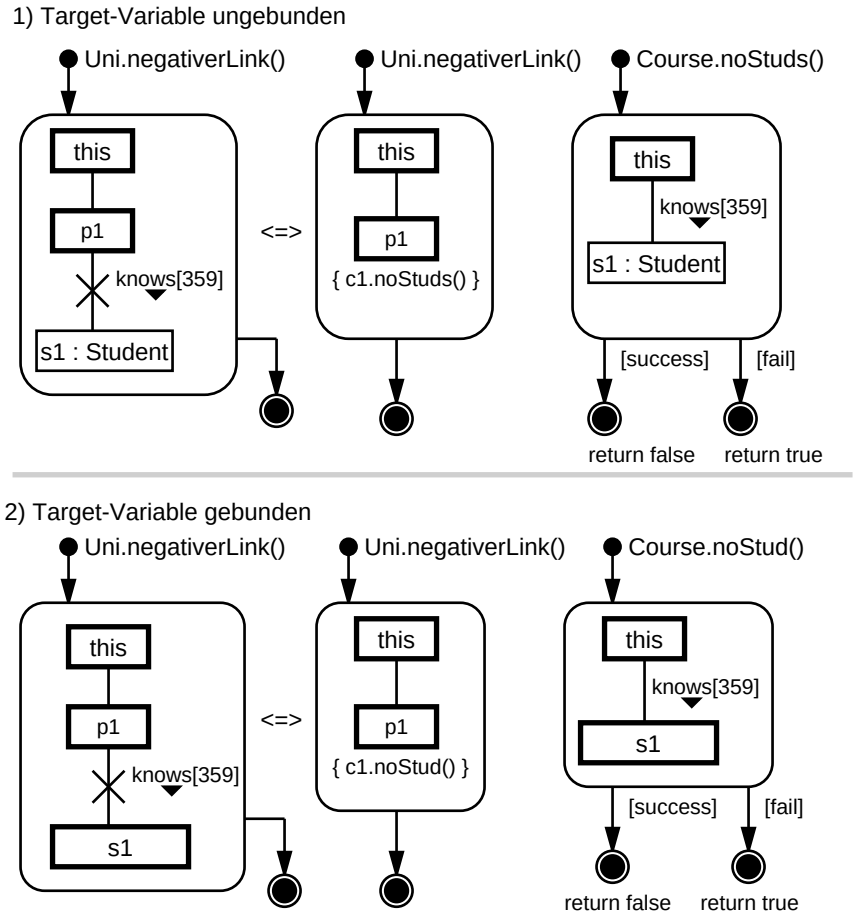



Abbildung 2.15: Semantik vom qualifizierten negativen Link

### Optionale Negative-Links

Ein optionaler Link muß bei der Anwendung des Story-Pattern nicht existieren. Er kann also auch keinen Fehler verursachen und damit zum Abbruch des Story-Pattern beitragen. Ist ein optionaler Link negativ markiert (siehe Abbildung 2.7), so kann das aber zu einem Fehler und zum Abbruch des Story-Pattern führen. Existiert kein optionaler Link bei der Regelanwendung zwischen zwei Variablen, so ist die Sache in Ordnung und es kann zu keinen Fehler durch den negativen Link kommen. Ein Fehler tritt aber auf, wenn der optional Link existiert. Die Negation sagt aus, daß es keinen Link geben darf, es wurde aber ein Link zwischen den betreffenden Variablen festgestellt, was dann zu einem Fehler und zum Abbruch des Story-Pattern führt. Ein

negativer optionaler Link widerspricht also der Semantik eines optionalen Link und kann es deshalb nicht geben.

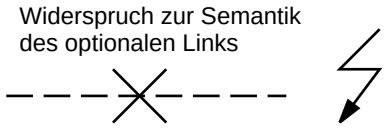



Abbildung 2.16: Negativer optionaler Link

### Negative-Links an einer gebundenen Menge

Eine Source-Variable die einen negativen Link zu einer gebundenen Mengenvariablen enthält (siehe Abbildung 2.17) bedeutet semantisch, daß die Source-Variable keine Links zu den Objekten besitzen darf, die in der Mengenvariablen enthalten sind. Ein negativer Link zu einer ungebundenen Menge ist semantisch äquivalent zu einer negativen Menge. Eine negative Menge kann es aber wie oben erläutert nicht geben, es sei denn die Mengenvariable ist mit Attributbedingungen verknüpft. In diesem Fall ist die Semantik für den negativen Link äquivalent und muß nicht noch einmal erwähnt werden.

Ist der negative Link zur gebundenen Menge qualifiziert, so darf keines der mit dem gegebenen Schlüssel assoziierten Objekte in der gebundenen Mengenvariable enthalten sein.

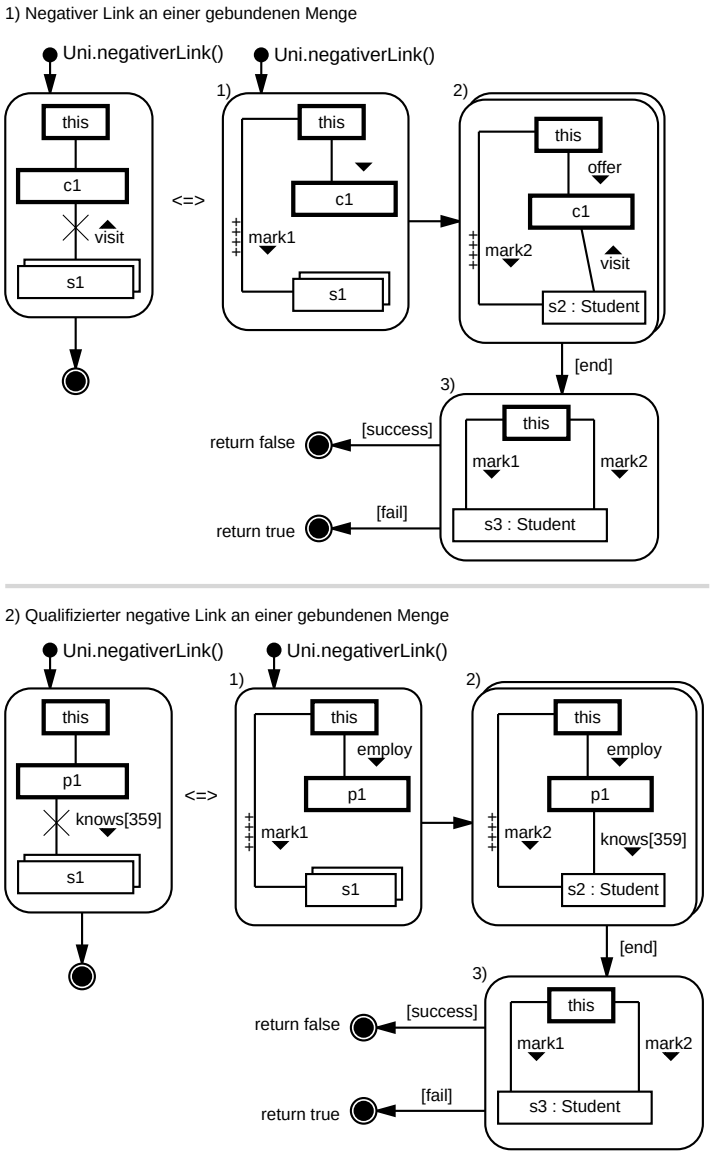



Abbildung 2.17: Semantik vom negativen Link an einer gebundenen Menge

# Kapitel 3

# Syntax und Semantik von Multilinks

(Kan-Sing, Wan)

## 3.1 Einleitung

Mit den bisherigen Sprachelementen des Story-Patterns war es bislang nicht möglich Ordnungsbeziehungen zwischen Elementen einer geordneten oder sortierten zu-n Assoziation[1] direkt zu spezifizieren. Es ist zum Beispiel nicht möglich direkt auf das erste oder letzte Element oder das n-te Element einer sortierten zu-n Assoziation zuzugreifen. Auch die Modellierung einer Vorgänger- oder Nachfolgerbeziehung zwischen zwei Variablen war nicht ohne weiteres möglich. Um in einem Story-Pattern eine Ordnungsbeziehung zu modellieren konnte der Benutzer entweder das Klassendiagramm entsprechend erweitern oder die Ordnungsbeziehung mußte mit den bisherigen Sprachelementen des Story-Patterns nachgebildet werden.

In dem Klassendiagramm muß die entsprechende Klasse der sortierten zu-n Assoziation, um eine Selbstassoziation oder Klasse erweitert werden, die die Ordnungsbeziehung zwischen den Instanzen dieser Klasse modelliert. Das Story-Pattern kann nun durch den Zugriff auf die Links dieser Asso-

[1] In geordneten zu-n Assoziationen werden die Elemente entsprechend der Reihenfolge in der sie eingefügt geordnet. Die Elemente einer sortierten zu-n Assoziation besitzen dagegen eine logische Ordnung. Die Elemente können zum Beispiel alphabetisch sortiert sein.

ziation oder Variablen dieser Klasse zum Beispiel auf den Vorgänger oder Nachfolger einer Variablen zu greifen. Eine andere Möglichkeit ist, die Klasse um einen Attribut zu erweitern, das die Position des Objektes innerhalb der Sortierung wiederspiegelt. Über Attribut-Constraints ist es dem Story-Pattern nun möglich, gezielt auf die einzelnen Elemente der zu-n Assoziation zuzugreifen. Diese Vorgehensweise setzt aber voraus, daß eine Sortierfunktion existiert, das die Elemente der zu-n Assoziation voher sortiert und das Attribut entsprechend aktualisiert.

Beide Vorgehensweisen sind mit einem gewissen Aufwand verbunden und erfordern entsprechende Änderungen an der Klassenstruktur oder den betroffenen Klassen. Dies kann zu einer erhöhten Komplexität der Klassendiagramme führen, was deren Lesbarkeit beeinträchtigt. Auch die nachträglichen Implementierungen können zu Fehlern führen. Mit den Multilinks ist es nun möglich, Ordnungsbeziehungen zwischen den Objekten direkt in einem Story-Pattern zu spezifizieren ohne Erweiterungen in den Klassen oder der Klassenstruktur vornehmen zu müssen. In einem UML-Klassendiagramm muß die entsprechende zu-n Assoziation nur als sortiert oder geordnet spezifiziert werden.

Durch die Einführung des Multilinks als ein weiteres Sprachelement des Story-Patterns besitzt sie eine feste Syntax und Semantik. Die Syntax garantiert die Konsistenz des Story-Patterns und verhindert eine falsche Anwendung seitens des Benutzers. Die Semantik des Multilinks spezifiziert eindeutig das Verhalten des Multilinks. Dadurch kann Anwender bei der Benutzung der Multilinks ein bestimmtes Verhalten erwarten. Ohne die Multilinks müßte man ihr Verhalten durch entsprechende Story-Diagramme nachbilden. Die Story-Patterns würden dadurch an Umfang und Komplexität zunehmen, was die Lesbarkeit und Wartung erschweren würde. Besonders, wenn viele oder komplexe Ordnungsbeziehungen modelliert werden müssen. Ein weiteres Problem entsteht, wenn sich mehrere Benutzer dazu entschließen, die Multilinks durch Story-Diagramme nachzubauen. Die unterschiedlichen Interpretationen der Benutzer können zu einer uneinheitlichen Semantik führen.

## 3.2 Semantik der Multilinks

In UML-Klassendiagrammen [8] können zu-n Assoziationen über Constraints als sortiert oder geordnet spezifiziert werden. Die Objekte einer sortierten zu-n Assoziation können zum Beispiel nach ihren Namen, ihrer ID-Nummer oder anhand einer beliebigen Sortierfunktion sortiert werden. Durch die Sor-

tierung wird eine Ordnungsrelation über die Objekte definiert. Diese Ordnungsrelation kann in einem Story-Pattern durch einen Multilink verdeutlicht werden. Im folgenden werden die unterschiedlichen Multilink-Typen vorgestellt.

Zu den einfachsten Multilink-Typen gehören der *first*- und *last-Multilink*. Der *first*-Multilink legt fest, daß das erste Element einer zu-n Assoziation einer Variablen zugeordnet wird. In einem Story-Pattern wird der *first*-Multilink durch einen eingehenden Pfeil an einem Link repräsentiert. Dieser Pfeil kann optional auch mit {first} beschriftet werden (siehe Abbildung 3.1). Der erwähnte Link verweist auf die entsprechende zu-n Assoziation, dessen erstes Element an der Variablen gebunden werden soll. Analog bindet der *last*-Multilink das letzte Element einer zu-n Assoziation an einer Variablen. In einem Story-Pattern wird es durch einen ausgehenden Pfeil an einem Link dargestellt. Optional kann der Pfeil mit {last} beschriftet werden (siehe Abbildung 3.1). Beide Multilink-Typen sind eindeutig in Bezug auf das zu bindende Objekt und erlauben den direkten Zugriff auf das erste oder letzte Element einer zu-n Assoziation. Die Semantik der *first*- und *last*-Multilinks werden in der folgenden Abbildung 3.1 durch einen Story Diagramm nachgebildet. Jedes Objekt der zu-n Assoziation besitzt eine Instanzvariable *position*, das die Position des Objektes innerhalb der zu-n Assoziation wiederspiegelt. Der Wert der Instanzvariable wird zuvor durch den Aufruf der Methode *sortStudents()* der Klasse *Uni* gesetzt.

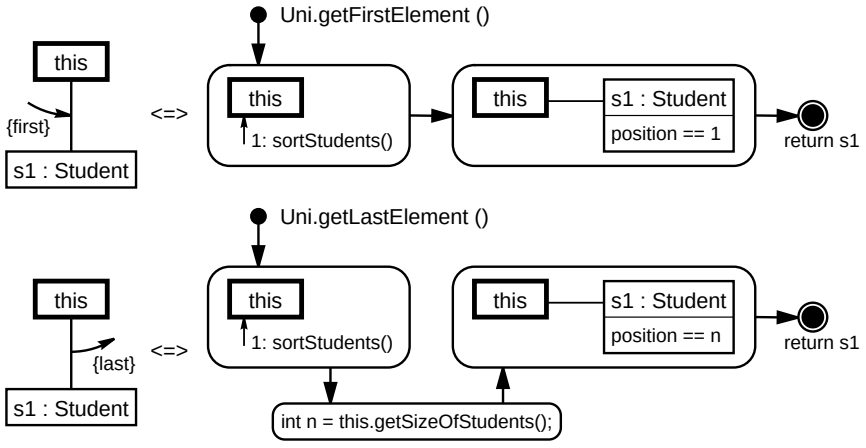



Abbildung 3.1: Die graphische Darstellung eines *first*- und *last*-Multilinks

Will man auf dem Nachfolger des ersten Elements zugreifen, reichen die bisherigen Sprachelemente des Story-Patterns nicht aus, um dies zu spezi-

fizieren, außer über dem Umweg der Story-Diagramme. Deshalb wird der *direct-Multilink* als ein weiteres Sprachelement eingeführt. Durch einen *direct*-Multilink ist es möglich, den direkten Nachfolger einer Variablen zu spezifizieren. Mit Hilfe des *first*-Multilinks ist es somit möglich, das zweite Element einer zu-n Assoziation einer Variablen zuzuordnen. In einem Story-Pattern wird ein *direct*-Multilink durch einen Pfeil zwischen zwei Links repräsentiert. Der Pfeil wird mit {next} beschriftet (siehe Abbildung 3.2). Für die Links gilt, daß sie von derselben Variable ausgehen und auf zu-n Assoziationen desselben Typs verweisen. Der Nachfolger einer Variable ist durch den *direct*-Multilink eindeutig bestimmt. Ein direkter Zugriff auf den Nachfolger ist dadurch möglich.

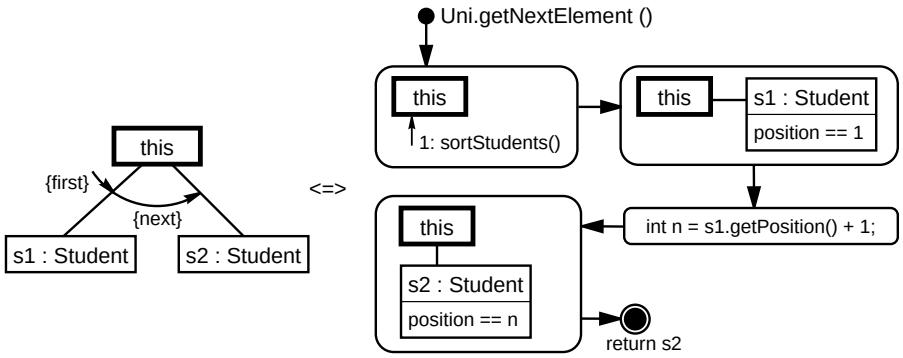



Abbildung 3.2: Die graphische Darstellung eines *next*-Multilinks

Der *index-Multilink* ist eine Verallgemeinerung des *direct*-Multilinks. Mit ihm ist es möglich, den n-ten Nachfolger einer Variable zu spezifizieren. Ein *index*-Multilink mit dem Index 1 besitzt dieselbe Semantik wie der *direct*-Multilink. Der Index ist eine positive natürliche Zahl echt größer Null. Benutzt man ein *first*-Multilink in Kombination mit einem *index*-Multilink, ist es möglich, über den Index des *index*-Multilinks jedes Element einer zu-n Assoziation exakt zu spezifizieren. In einem Story Pattern unterscheidet sich die graphische Darstellung des *index*-Multilinks vom *direct*-Multilink nur durch seine Beschriftung. Ein *index*-Multilink wird durch die Beschriftung {next[n]} gekennzeichnet, wobei das n den Index repräsentiert und durch eine geeignete Zahl zu ersetzen ist (siehe Abbildung 3.3).

Alle bisher vorgestellten Multilink-Typen spezifizieren eindeutig, welches Element einer zu-n Assoziation an einer Variable zu binden ist. In manchen Situationen ist eine weniger restriktive Spezifikation erwünscht oder eine genauere nicht möglich, weshalb als ein weiteres Sprachelement des Story-

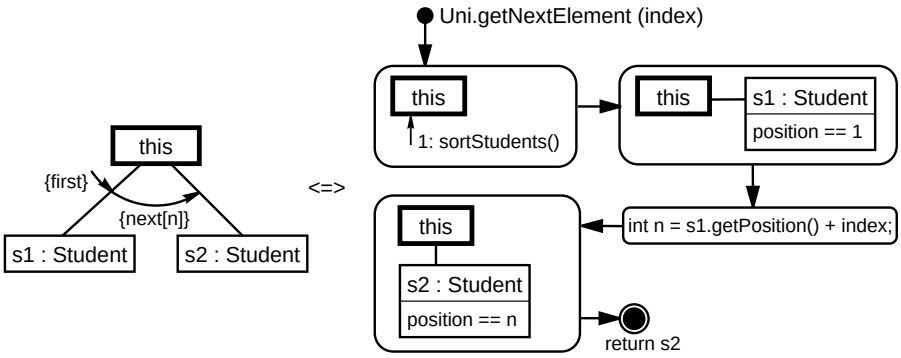



Abbildung 3.3: Die graphische Darstellung eines *index*-Multilinks

Patterns der *indirect-Multilink* eingeführt wird. Der *indirect*-Multilink spezifiziert den indirekten Nachfolger einer Variable. Der Nachfolger ist nicht eindeutig bestimmt, sondern kann ein beliebiges Element einer zu-n Assoziation sein, solange es die durch den Multilink spezifizierte Ordnungsbeziehung nicht verletzt. In einem Story-Pattern wird der *indirect*-Multilink durch die Beschriftung {...} unterhalb des Pfeils gekennzeichnet (siehe Abbildung 3.4).

Damit wurden alle Sprachelemente der Multilinks vorgestellt. Sollen komplexere Ordnungsbeziehungen zwischen mehreren Variablen modelliert werden, ist es möglich diese Sprachelemente miteinander zu kombinieren. Hierzu werden die einzelnen Multilinks miteinander verkettet zu einem *Multilink-Pfad* (siehe Abbildung 3.5). Die Multilinks sowie die Variablen können dabei einen beliebigen Typ haben. Ein einfacher Multilink kann als den einfachsten Fall eines Multilink-Pfads betrachtet werden. Der erste Multilink eines

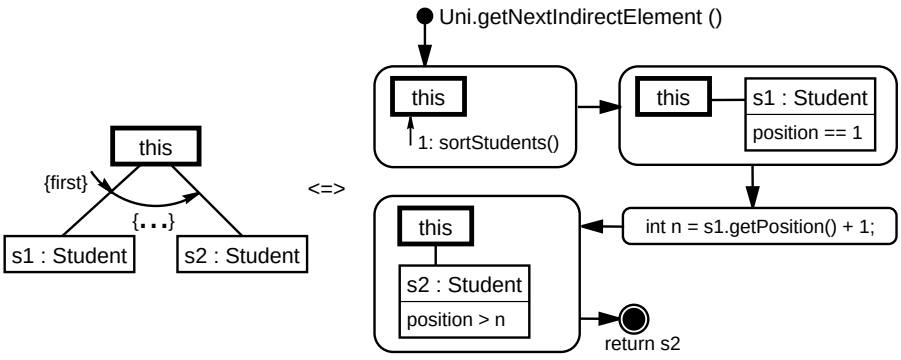



Abbildung 3.4: Die graphische Darstellung eines *indirect*-Multilinks

Pfades wird als *entry*-Link bezeichnet, wobei ein *first*-Multilink, falls vorhanden, nicht berücksichtigt wird. Die Semantik eines Multilinks bleibt in einem Multilink-Pfad unverändert. Die Besonderheit des Multilink-Pfades ist seine transitive Eigenschaft, die von allen beteiligten Multilinks des Multilink-Pfades eingehalten werden mußsen. Alle Elemente einer zu-n Assoziation, die an einem Multilink-Pfad gebunden werden, müssen diese Eigenschaft erfüllen. Man ersetze die Multilinks durch das „$\leq$“ Symbol und die gebundenen Elemente der zu-n Assoziation durch eine Zahl, die ihre entsprechende Position in der Assoziation wiederspiegeln, wenn man die einzelnen Elemente der Assoziation vom Anfang bis Ende entsprechend ihrer Sortierung durchnummerieren würde. Die Ungleichung, die sich daraus ergibt, muß erfüllt sein. Eine Variable kann durchaus mehrere Multilink-Pfade besitzen, die aber jeweils voneinander unabhängig sind. Die transitive Eigenschaft gilt nur für die Multilinks desselben Multilink-Pfades und erstreckt sich nicht über Multilinks anderer Multilink-Pfade.

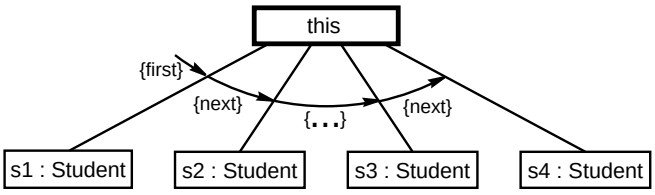



Abbildung 3.5: Beispiel eines Multilinks-Pfades

Die Multilinks wurden bisher nur in Verbindung mit normalen Variablen benutzt. In einem Story-Pattern existieren aber auch optionale und mengenwertige Variablen. Die Semantik einer optionalen Variable in Verbindung mit einem Multilink ist unverändert geblieben. Sie stellt für die optionale Variable lediglich eine zusätzliche Bedingung dar. Kann kein geeignetes Element gefunden werden, bleibt die Variable ungebunden und es wird kein Bindungsfehler erzeugt. Dagegen besitzt die mengenwertige Variable in Zusammenhang mit den Multilinks eine erweiterte Semantik, besonders wenn die Mengenvariable innerhalb eine Multilink-Pfades verwendet wird. Durch die Multilinks wird der Bindungsbereich einer Mengenvariablen eingegrenzt. Anhand des folgenden Beispiels wird auf die geänderte Semantik der Mengenvariablen eingegangen.

Das Beispiel in der Abbildung 3.6 zeigt eine Mengenvariable *m1*, die links und rechts von zwei gebundenen Variablen eingegrenzt wird. Die Variablen *s1* und *s2* bilden in diesem Fall eine untere und obere Schranke für die Mengenvariable *m1*. Aufgrund der transitiven Eigenschaft des Multilink-Pfades,

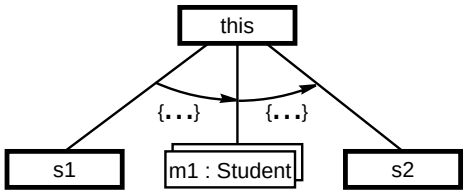



Abbildung 3.6: Beispiel eines Multilinks-Pfades mit einer Mengenvariablen

kann die Mengenvariable *m1* nun nur die Objekte der zu-n Assoziation aufnehmen, die zwischen den beiden gebundenen Variablen *s1* und *s2* liegen. Alle Objekte außerhalb des von *s1* und *s2* gebildeten Intervalls würde die Transitivität des Multilink-Pfades verletzen.

Die Anwendung der Multilinks mit diesen Variablen-Typen ist nicht immer sinnvoll, deshalb werden bestimmte Anwendungsfälle ausgeschlossen. Es ist zum Beispiel nicht sinnvoll ein *first*- oder *last*-Multilink zusammen mit einer Mengenvariablen zu benutzen. Eine Variable, die über einem *first*-Multilink gebunden wurde, dient meistens als Bezugspunkt für andere Multilinks, um weitere Variablen zu binden. Verwendet man statt einer normalen Variable eine Mengenvariable, geht der Bezugspunkt verloren, da eine Menge noch weitere Elemente einer zu-n Assoziation besitzen kann. Jedes Element einer sortierten zu-n Assoziation besitzt einen eindeutigen direkten Nachfolger oder Vorgänger. Betrachtet man nun eine Menge von Elementen als Ganzes, kann man dieser Menge keinen eindeutigen direkten Nachfolger oder Vorgänger zuordnen. Es besitzt aber sehrwohl einen indirekten Nachfolger oder Vorgänger. Deshalb können Mengenvariablen nur mit *indirect*-Multilinks verwendet werden. Diese Einschränkungen spiegeln sich auch in der Syntax der Multilinks wieder.

## 3.3 Syntax der Multilinks

Eine Syntax beschreibt den Aufbau einer Sprache und ihre zulässigen Elemente. Das gleiche gilt für die Syntax von Story-Pattern. Sie beschreibt die Menge der gültigen Graphen, die sich durch die Anwendung der einzelnen Sprachelemente unter Einhaltung bestimmter Regeln ergeben. Die Syntax von Story-Pattern wird im Rahmen der Sprachspezifikation »Story Driven Modelling« in [10] genauer beschrieben. Durch die Erweiterung der Sprache um die Multilinks, muß die bisherige Syntax erweitert werden, um weiterhin

einen konsistenten und gültigen Graphen garantieren zu können.

Die Erweiterung der Syntax beschreibt nur den zulässigen Aufbau der Multilinks. Geht man davon aus, daß das Story-Pattern in einem konsistenten Zustand ist und man nur Operationen zuläßt, die einen zulässigen Multilink generieren, bleibt der Graph konsistent. Deshalb reicht eine Beschreibung der gültigen Multilinks hier aus. Bevor aber darauf eingegangen wird, erfolgt erst einmal eine Begriffsdefinition, die für die Erklärung der Syntax und in den nachfolgenden Kapiteln verwendet wird.

In dem vorangegangenen Abschnitt wurde die graphische Darstellung der einzelnen Multilink-Typen vorgestellt. Mit Ausnahme der *first-* und *last-*Multilinks, wurde ein Multilink immer zwischen Links gezeichnet, die von derselben Variable ausgingen. Diese Variable wird als *Container-Variable* bezeichnet. Die Variable wird so genannt, weil das an ihr gebundene Objekt die zu-n Assoziation durch einen Container [11] realisiert, um die Elemente dieser zu-n Assoziation zu verwalten. Container dienen dazu, eine Menge von Objekten zu verwalten und können zum Beispiel durch eine Liste, eine Map oder einen Baum realisiert werden. Das entsprechende Objekt wird als *Container-Objekt* bezeichnet. Multilinks beziehen sich auf Elemente einer zu-n Assoziation und gehören deshalb dem Container-Objekt an, das die zu-n Assoziation implementiert. Die Links werden entsprechend ihrer Lage zum Multilink bezeichnet. Der Link mit dem ausgehenden Multilink wird als *Source-Link* bezeichnet und der Link mit dem eingehenden Multilink als *Target-Link*. Die am Source-Link hängende Variable, nicht die Container-Variable, wird dementsprechend als *Source-Variable* bezeichnet. Analog dazu wird die am Target-Link hängende Variable als *Target-Variable* bezeichnet. Die an den Variablen gebunden Objekte werden ähnlich benannt. Die Abbildung 3.7 zeigt eine Übersicht der eingeführten Begriffe.

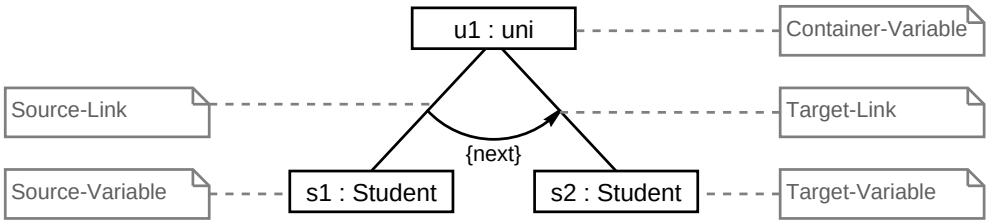



Abbildung 3.7: Übersicht der von den Multilinks verwendeten Begriffe

Die Anwendbarkeit eines Multilinks wird durch die Links einer Variablen bestimmt. Multilinks können nur zusammen mit Links benutzt werden, die

eine Instanz auf eine geordnete oder sortierte zu-n Assoziation besitzen. Für die *direct*-, *index* und *indirect*-Multilinks gilt zusätzlich, daß deren Source- und Target-Links auf dieselbe Instanz einer Assoziation verweisen. Diese Regelung gilt nicht für die *first*- und *last*-Multilinks, da sie nur auf einem Link angewendet werden. Eine weiter Regel schränkt die Anzahl der ein- und ausgehenden Multilinks von einem Link ein. Zu jedem Link darf es maximal nur einen ein- und ausgehenden Multilink geben. Es wäre durchaus möglich gewesen mehr als einen ein- und ausgehenden Multilink zu erlauben, dies hätte aber die Lesbarkeit, Syntaxüberprüfung, Implementierung und Teilgraphensuche erschwert ohne eine größere Ausdrucksmächtigkeit zu erreichen. Es ist nicht schwierig dafür ein semantisch äquivalentes Story-Pattern beziehungsweise Story Diagramm zu konstruieren, bei dem alle Links maximal nur einen ein- und ausgehenden Multilink besitzen.

Die zulässigen Typen, die ein Multilink annehmen kann, werden durch dessen Source- und Target-Variable bestimmt. Die Links spielen für die Entscheidung, welcher Multilink-Typ verwendet werden kann, keine Rolle. Die *direct*- und *index*-Multilinks können nur zwischen normalen und optionalen Variablen verwendet werden. Sie können nicht verwendet werden, wenn einer der beteiligten Source- oder Target-Variablen mengenwertig ist. Ist ein Source- oder Target-Link mit einer Mengenvariable verbunden, so sind die von diesem Link ein- und ausgehenden Multilinks grundsätzlich vom Typ *indirect*. Der *indirect*-Multilink kann für jede beliebige Typ-Kombination der Source- und Target-Variablen benutzt werden. Die *first*- und *last*-Multilinks sind nur anwendbar mit den normalen oder optionalen Variablen.

In einem Story-Pattern sollte man darauf achten die *first*- und *last*-Multilinks für dieselbe zu-n Assoziation nicht mehrmals zu verwenden. In einem konkreten Story-Pattern bedeutet es, daß alle Variablen desselben Typs, die über Multilinks auf Elemente derselben zu-n Assoziation zugreifen, insgesamt nur einen *first*- und *last*-Multilink besitzen sollten. Andernfalls wird die isomorphe Grapheigenschaft des Story-Patterns verletzt, sofern keine *maybe*-Klausel verwendet wurde.

## 3.4 Erzeugen und löschen mit Multilinks

In diesem Abschnitt wird die Semantik der Multilinks in Zusammenhang mit den «*create*»- und «*destroy*»-Modifikatoren erklärt. Entsprechend der Semantik von Story-Pattern, werden die Modifikationen erst ausgeführt, wenn die Teilgraphensuche für das gesamte Story-Pattern erfolgreich beendet wur-

de. Die Semantik des ≪*destroy*≫-Modifikators hat sich nicht geändert und wird hier nur der Vollständigkeitshalber erwähnt. Hat die Variable einen ≪*destroy*≫-Modifikator, wird das an ihr gebundene Objekt gelöscht und damit automatisch aus dem Container entfernt. Besitzt ein Link den ≪*destroy*≫-Modifikator, wird das an ihr hängende Objekt nicht gelöscht, sondern nur aus dem Container entfernt.

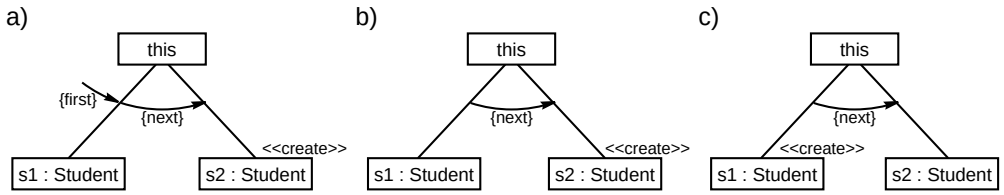



Abbildung 3.8: Einfügen von Objekten mit Multilinks

Der ≪*create*≫-Modifikator erzeugt ein neues Objekt und fügt es dem Container hinzu, wenn er zusammen mit einer Variable benutzt wird. Wird ein Link mit dem ≪*create*≫-Modifikator ausgezeichnet, wird das entsprechende Objekt, das an der Variable gebunden wurde, nur dem Container hinzugefügt ohne es zuvor neu zu erzeugen. In der Regel wird das Objekt dann am Anfang des Containers eingefügt. Über die Multilinks ist nun möglich zu spezifizieren, an welcher relativen bzw. absoluten Position das Objekt in dem Container eingefügt werden soll. Das Beispiel a) in der Abbilding 3.8 modelliert eine Einfügeoperation mit absoluten Positionsangaben. Durch die Kombination des *first*- und *index*-Multilinks wird eindeutig festgelegt, wo das neue Objekt einzufügen ist. Das Beispiel b) spezifiziert ein relative Einfügeoperation. Da nur festgelegt wird, daß das neue Objekt nach dem gebundenen Objekt der Variable *s1* eingefügt werden soll. Im Beispiel c) wird das neue Objekt in dem Container vor dem gebundenen Objekt an der Variablen *s2* eingefügt.

# Kapitel 4

# Einführung in die Teilgraphensuche

(Kan-Hung Wan, Kan-Sing Wan)

## 4.1 Einleitung

Ein Story Pattern ist eine Graphersetzungsregel für dessen erfolgreiche Anwendung ein isomorpher Teilgraph in einem Wirtsgraph gefunden werden muß. Der Wirtsgraph ist in diesem Fall die Laufzeitobjektstruktur. Das Problem ist also die Objekte der Laufzeitobjektstruktur auf die Variablen des Story-Patterns abzubilden. Da das Problem der Teilgraphensuche NP-vollständig [13] ist und ein deterministischer Algorithmus eine exponentielle Laufzeit hat, muß die Teilgraphensuche für den praktischen Einsatz optimiert werden.

## 4.2 Kostenbewertung von Links

Für die Optimierung der Laufzeit der Teilgraphensuche werden in Fujaba [10] zwei Ansätze verfolgt, das *Force-Failure*-Prinzip und die Einschränkung des Suchraums. Beim Force-Failure Ansatz wird versucht möglichst früh Fehler beim Binden von Objekten an Variablen zu erkennen. Tritt ein Fehler auf, kann die Suche abgebrochen und das Story Pattern verlassen werden, weil das Story Pattern nicht mehr erfolgreich ausgeführt werden kann. Ein Fehler

tritt auf, wenn kein passendes Objekt in der Objektstruktur gefunden werden kann.

Eine weitere Verbesserung der Laufzeit kann durch die Eingrenzung des Suchraums erreicht werden. Ein naiver Algorithmus würde z.B. für das Matchen einer Variable alle Instanzen einer Klasse entsprechend des Variablentyps durchlaufen. Jedes Objekt muß dann noch getestet werden, ob es die im Story-Pattern spezifizierten Bedingungen erfüllt. Um den Suchraum einzugrenzen, werden in Fujaba deshalb nur die Nachbarobjekte eines Objektes betrachtet. Die Nachbarobjekte eines Objektes sind über die Links erreichbar. Statt alle Instanzen einer Klasse zu durchlaufen, werden nun nur die über einem Link erreichbaren Instanzen einer Klasse betrachtet.

Ein Teilgraph wird in Fujaba also sukzessiv über bereits gebundene Variablen aufgebaut. Dies setzt voraus, daß in einem Story Pattern mindestens eine gebundene Variable für jede Zusammenhangskomponente existiert von der aus dann die Suche begonnen werden kann. In den meisten Fällen ist es die *this* Variable.

Um eine weitere Optimierung der Laufzeit zu erreichen werden die beiden besprochenen Ansätze kombiniert. Die Begrenzung des Suchraums wird dabei dem Force-Failure-Prinzip untergeordnet. Hierzu werden die von einem Objekt ausgehenden Assoziationen hinsichtlich ihrer Kosten bewertet. Die Höhe der Kosten entsprechen dabei dem geschätzten Suchaufwand. Eine Einschätzung des Suchaufwands kann über die Kardinalität der Assoziation erreicht werden. So ist der Suchaufwand für eine zu-n Assoziation gewöhnlich höher als bei einer zu-1 Assoziation. Nach dem Force-Failure-Prinzip wird deshalb versucht zuerst ein Objekt über die zu-1 Assoziation zu binden.

Das Force-Failure-Prinzip wird in Fujaba über Prioritäten realisiert. Jeder Link in Fujaba erhält eine Priorität. Die Priorität bestimmt die Abarbeitungsreihenfolge des Links bezogen auf seine Kosten. Ein Link mit einer hohen Priorität ist demzufolge günstiger als ein Link mit einer niedrigen Priorität und wird deshalb zuerst abgearbeitet.

Der kostengünstigste Link in einem Story Pattern, ist ein *Check-Link*. Ein Check-Link ist ein Link zwischen zwei gebundenen Variablen. Dieser Fall kann eintreten, wenn die beiden Variablen in dem Story-Pattern als gebunden spezifiziert worden sind oder über günstigere Wege gebunden wurden. Es wird also nur überprüft, ob zwischen den an den Variablen gebunden Objekten ein Link existiert. Ein Check-Link ist insofern günstig, da die Suche sofort abgebrochen werden kann, wenn zwischen den beiden gebundenen Objekten kein Link gefunden werden kann. Das Story Pattern kann als Folge nicht

mehr erfolgreich ausgeführt werden.

Die nächste Prioritätsklasse umfaßt alle nicht optionalen Links zwischen zwei normalen Variablen, von denen eine Variable gebunden ist. Für alle Links dieser Klasse muß also ein passendes Objekt in der Objektstruktur gefunden werden. Die Links dieser Klasse lassen sich in zwei Kategorien einteilen, Links die eine zu-1 oder zu-n Assoziation repräsentieren. Da die zu-1 Assoziation einen kleineren Suchraum hat, bekommt sie eine höhere Priorität gegenüber der zu-n Assoziation. Diese Bewertung entspricht dem Force-Failure-Prinzip. Ein kleiner Suchraum ist schneller abgesucht als ein großer, weshalb Fehler auch schneller erkannt werden können. Enthält der gesamte Suchraum kein passendes Objekt, ist das Story Pattern nicht mehr zu erfüllen und die Teilgraphensuche kann vorzeitig abgebrochen werden.

Die dritte Prioritätsklasse enhält alle Links, die optional sind oder zu einer optionalen Variable führen. Diese Klasse läßt sich in drei Kategorien einteilen, die optionalen Check-Links, die optionalen zu-1 Links und die optionalen zu-n Links. In dieser Klasse hat ein optionaler Check-Link die höchste Priorität, die optionalen zu-1 Link und zu-n Link erhalten jeweils eine niedrigere Priorität. Ein optionaler Check-Link ist ein Link, bei dem mindestens einer der beiden gebundenen Variablen, eine optionale Variable ist. Die zu normalen Check-Links und normalen Links gemachten Aussagen, treffen auch auf die optionalen zu-1 und zu-n Links zu. Der Unterschied ist, daß optionale Variablen keine Bindungsfehler erzeugen, wenn sie nicht gebunden werden konnten. Das ist der Grund, warum optionale Links entsprechend des Force-Failure-Prinzips allgemein niedriger bewertet werden.

| Klasse | Name der Prioritäten | Beschreibung |
|---|---|---|
| 1 | P-Check | Check-Links |
| 2 | P-To-One | normale zu-1 Links |
| | P-To-Many | normale zu-n Links |
| 3 | P-Optional-Check | optionale Check-Links |
| | P-Optional-To-One | optionale zu-1 Links |
| | P-Optional-To-Many | optionale zu-n Links |
| 4 | P-Set | mengenwertige zu-n Links |

Abbildung 4.1: Auflistung der Prioritäten in absteigender Reihenfolge

Die vierte Prioritätsklasse bezieht sich auf alle Links über die mengenwertige Variablen erreicht werden können. Diese Klasse enthält nur zu-n

Links, da zu-1 Links die Semantik der mengenwertigen Variablen widersprechen würden. Die mengenwertigen Variablen werden zuletzt behandelt, weil mengenwertige Variablen optional sind und im Durchschnitt einen höheren Suchaufwand verursachen. Die Suche nach einem passenden Objekt in der Laufzeitobjektstruktur für eine optionale Variable kann abgebrochen werden, sobald eine passende Belegung für die Variable gefunden wurde. Bei einer mengenwertigen Variable muß dagegen der gesamte Suchraum überprüft werden, um alle möglichen Kandidaten zu binden. Die Abbildung 4.1 zeigt alle Prioritäten zusammengefaßt in absteigender Reihenfolge.

Die Bewertung aller erreichbaren Links bestimmt also die Abarbeitungsreihenfolge eines Story Patterns. Jedesmal nach dem Binden einer Variable müssen die Links neu bewertet werden. Da sich die Priorität eines Links nur ändert, wenn sich einer der am Link hängenden Variablen ändert, müssen nur die Links an einer neu gebundenen Variablen neu bewertet werden. Danach wird der nächste Link mit der höchsten Priorität ausgewählt, über dem die Teilgraphensuche fortgesetzt wird. Nachdem alle Variablen in dem Story-Pattern gebunden wurden sind, können die spezifizierten Modifikationen an der Laufzeitobjektstruktur ausgeführt werden.

# Kapitel 5

# Codegenerierung für Negative-Knoten und -Links

(Kan-Hung, Wan)

## 5.1 Einführung

In diesem Kapitel wird auf die Codegenerierung speziell für *Negative-Knoten* und *Negative-Links* eingegangen, nachdem im vorhergehenden Kapitel die notwendigen Grundlagen für die Teilgraphensuche in Fujaba geschaffen wurden.

Die implementierte Teilgraphensuche hat die Aufgabe sicherzustellen, daß die negativen Constraints auch erfüllt werden. Sie muß zum Beispiel gewährleisten, daß an einer Source-Variable mit einem negativen Knoten nur Objekte aus der Laufzeitobjektstruktur gebunden werden können, die auch die negativen Constraints erfüllen. Das heißt für die implementierte Teilgraphensuche, sie muß alle in Frage kommende Source-Objekte überprüfen, ob sie Links zu Objekten vom Typ des negativen Knoten besitzen. Hat das Source-Objekt Links zu Objekten, die durch den negativen Knoten verboten wurden, so kommt das Source-Objekt für die Bindung nicht in Frage.

Die Überprüfung der Links, die durch den negativen Knoten verursacht wurden, stellt ein zusätzlicher Suchaufwand dar. Hätte der negative Knoten noch Attributbedingungen, so würde sich der Suchaufwand noch weiter erhöhen. Eine Optimierung der Teilgraphensuche, wie im Kapitel 3 geschil-

dert, ist also auch für die Negativen-Knoten und Negativen-Links erforderlich. Für eine Optimierung der Teilgraphensuche ist die Einschätzung des Suchaufwandes für die Negative-Knoten und -Links wichtig. Deshalb wird die implementierte Teilgraphensuche für die einzelnen Typen von Negativen-Knoten und -Links vorgestellt. Anschließend wird die Kostenbewertung und die darauf aufbauende Optimierung der Negativen-Knoten und Negativen-Links behandelt.

## 5.2 Negative-Knoten

### Links mit Negativen-Knoten

Semantisch spielt es keine Rolle, ob eine Source-Variable durch einen zu-1 oder zu-n Link mit einem negativen Knoten verbunden ist. Die Semantik des negativen Knoten verändert sich nicht durch die Kardinalität des Links. Unabhängig von der Kardinalität gilt, daß die Source-Variable keine Links zu Objekten besitzen darf, die vom negativen Knoten ausgeschlossen wurden. Für die Codegenerierung spielt die Kardinalität des am negativen Knoten hängenden Links aber eine entscheidene Rolle.

Im folgenden wird der Begriff *Assoziationsattribut* erklärt, der bis zum Ende dieses Kapitels verwendet wird. Das Assoziationsattribut implementiert die Assoziation zwischen zwei Klassen. Beide Klassen besitzen jeweils ein Assoziationsattribut, das seine Assoziation auf die andere Klasse realisiert. Bei einer zu-1 Assoziation ist das Assoziationsattribut eine einfache Objektreferenz auf die andere assoziierte Klasse. Besitzt die Assoziation die Kardinalität `n`, so wird das Assoziationsattribut durch einen *Container-Objekt* realisiert. Das Container-Objekt speichert die Links beziehungsweise Objektreferenzen, die er zur assoziierten Klasse besitzen darf.

Ist die Source-Variable durch einen zu-1 Link mit einem negativen Knoten verbunden, so muß bei der Überprüfung der negativen Constraints nur sichergestellt werden, daß das entsprechende Assoziationsattribut des zu-1 Link keine Referenz auf ein anderes Objekt enthält. Handelt es sich um einen zu-n Link, so wird die Assoziation durch einen Container-Objekt realisiert und es muß überprüft werden, ob das Container-Objekt leer ist. Ist das Container-Objekt nicht leer, so werden die negativen Constraints verletzt und das Source-Objekt darf nicht gebunden werden.

Die Tabelle 5.1 zeigt in der linken Spalte zwei einfache Beispiele mit

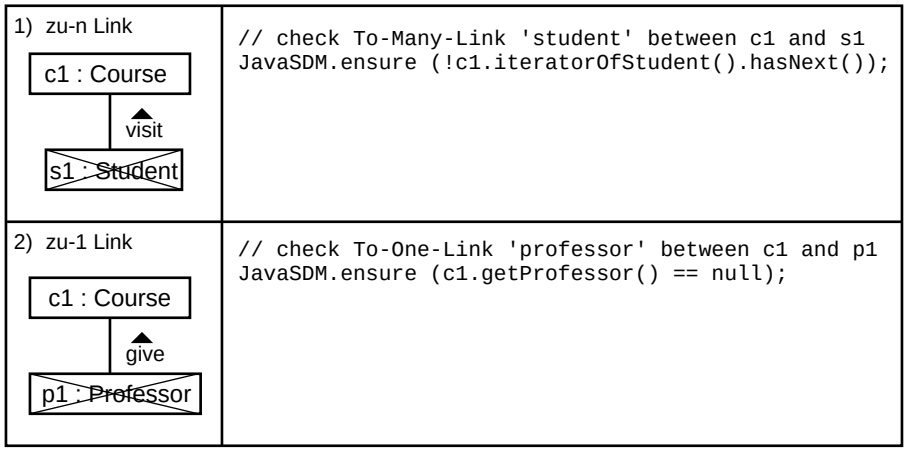



Tabelle 5.1: Codegenerierung für einfache Negative-Knoten

negativem Knoten und in der rechten Spalte der entsprechende generierte Quellcode für das jeweilige Beispiel. Für diese und alle weiteren folgenden Tabellen wird angenommen, daß die Teilgraphensuche immer von der oberen zur unteren Variable traversiert. Das heißt die obere Variable ist die Source-Variable und die untere Variable die Target-Variable.

Im ersten Beispiel ist die Source-Variable *c1* durch einen zu-n Link mit dem Negativen-Knoten *s1* verbunden. Die Vorlesung *c1* darf semantisch keine Links zu Objekten besitzen, die vom Typ *Student* sind. Die Überprüfung der negativen Constraints wird mit Hilfe der Methode `JavaSDM.ensure()` durchgeführt. Die Anweisung erwartet als Parameter einen booleschen Ausdruck. Ergibt der Ausdruck den Wert `false`, so wird eine Exception ausgelöst. Passiert das bei der Überprüfung der negativen Constraints so bedeutet dies, daß das Source-Objekt die negativen Constraints verletzt hat. In der `JavaSDM.ensure()` Methode wird nun überpüft, ob das Container-Objekt des Assoziationsattributs von *c1*, das die Assoziation zwischen den Klassen *Course* und *Student* realisiert, leer ist. Analog wird für das zweite Beispiel überprüft, ob die Objekt-Referenz des Assoziationsattribut von *c1* zu einem *Professor*-Objekt gleich `null` ist. Die negativen Constraints werden nur erfüllt, wenn der boolesche Ausdruck der `JavaSDM.ensure()` Methode den Wert `true` ergibt.

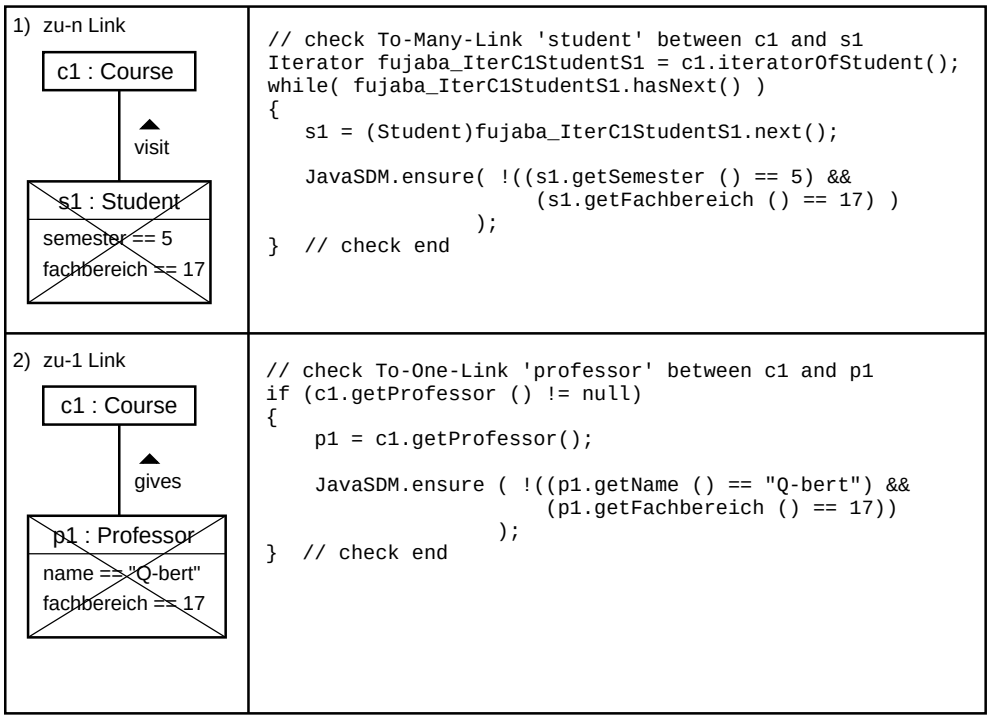



Tabelle 5.2: Codegenerierung für Negative-Knoten mit Attribut-Constraints

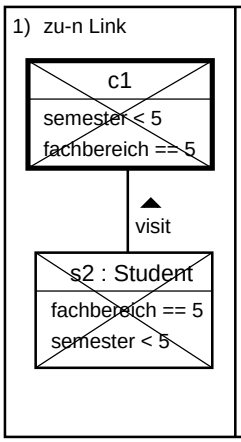
### Negativer Knoten mit Attribut-Constraints

Bei einem zu-1 Link der Source-Variable an einem Negativen-Knoten mit Attributbedingungen ist es nicht mehr möglich die Objektreferenz des entsprechenden Assoziationsattributs, wie im obigen Fall, einfach auf `null` zu überprüfen. Schließlich darf die Source-Variable semantisch durchaus Links zu Objekten besitzen, die vom Typ des Negativen-Knoten sind. Die möglichen Objekte dürfen nur nicht alle Attributbedingungen erfüllen. Bei der Codegenerierung wird nun zuerst überprüft, ob das Source-Objekt eine Referenz auf eine Instanz vom Typ des negativen Knoten enthält. Ist dem nicht so, erübrigt sich auch eine Überprüfung der Attributbedingungen. In diesem Fall erfüllt das Source-Objekt die negativen Constraints und kann an der Source-Variable gebunden werden. Besitzt das Source-Objekt aber eine Referenz auf ein Objekt vom Typ des negativen Knoten, so müssen die Attributbedingungen kontrolliert werden. Treffen alle Attributbedingungen zu, so schlägt die Teilgraphensuche für das Source-Objekt fehl und sie kann nicht an die Source-Variable gebunden werden.

Ist die Source-Variable durch einen zu-n Link mit einem Negativen-Knoten mit Attribut-Constraints verknüpft, so kann wie beim zu-1 Link das Container-Objekt des Assoziationsattributs nicht mehr einfach auf die leere Menge überprüft werden. Stattdessen müssen alle Links des Source-Objektes zu Objekten vom Typ des negativen Knoten einzeln kontrolliert werden. Das heißt die implementierte Teilgraphensuche iteriert durch alle Links beziehungsweise Objektreferenzen des entsprechenden Container-Objekts, das die Assoziation zum negativen Knoten implementiert. Ist das Container-Objekt leer, so ist keine Überprüfung der Attributbedingungen notwendig. Das Source-Objekt verletzt also keine negativen Constraints und kann deshalb an der Source-Variable gebunden werden. Enthält das Container-Objekt Elemente, so muß jedes Element im Container einzeln auf die Attribut-Constraints getestet werden. Dabei darf es entsprechend der Semantik kein Element im Container geben, das alle Attributbedingungen erfüllt. Erfüllt nur ein Element alle Attributbedingungen, so schlägt die Teilgraphensuche für das Source-Objekt fehl. Umgekehrt, kann das Source-Objekt erfolgreich an der Source-Variable gebunden werden, wenn keines der Elemente im Container alle Attributbedingungen erfüllt.

In der Tabelle 5.2 sind zwei Beispiele mit Code dargestellt. In beiden Beispielen enthält die Source-Variable *c1* einen Negativen-Knoten, der mit Attributbedingungen verknüpft ist. Das erste Beispiel enthält einen zu-n Link von der Source-Variable zum Negativen-Knoten *s1* vom Typ *Student*. Durch den Negativen-Knoten *s1* darf die Vorlesung *c1* semantisch keine Studenten besitzen, die im Fachbereich 17 und im fünften Semester sind. Der abgedruckte Code rechts spiegelt die Teilgraphensuche, wie sie oben für einen zu-n Link beschrieben wurde, wieder. In der `while`-Schleife wird das Container-Objekt von *c1* mit Hilfe eines Iterators durchlaufen. Das Container-Objekt enthält alle Links des aktuellen Kurs-Objekts zu seine Studenten-Objekte. Die `while`-Schleife wird solange durchlaufen, bis alle Studenten-Objekte überprüft wurden. In der Schleife werden die Attribut-Constraints für das aktuelle Studenten-Objekt kontrolliert. Erfüllt das aktuelle in Bearbeitung befindliche Studenten-Objekt alle Attributbedingungen, so wird durch die Methode `JavaSDM.ensure()` eine Exception ausgelöst und die Teilgraphensuche bricht erfolglos für das Source-Objekt *s1* ab.

Die Source-Variable im zweiten Beispiel enthält einen zu-1 Link zum Negativen-Knoten *p1*. Semantisch darf die Vorlesung *c1* von keinem Professor gehalten, der im Fachbereich 17 ist und „Q-bert“ heißt. Der Code implementiert die Teilgraphensuche für einen zu-1 Link, wie er oben allgemein beschrieben wurde. In der `if`-Anweisung wird überpüft, ob das Source-




```
// check Negative Attribute-Constraints for c1
JavaSDM.ensure (    (c1 != null) &&
                   !((c1.getSem () < 5) &&
                     (c1.getFachbereich () == 5))
               );

// check To-Many-Link 'student' between c1 and s2
Iterator fujaba__IterC1StudentS2 = c1.iteratorOfStudent ();
while (fujaba__IterC1StudentS2.hasNext ())
{
   s2 = (Student) fujaba__IterC1StudentS2.next ();

   JavaSDM.ensure ( !((s2.getFachbereich () == 5) &&
                      (s2.getSemester () < 5))
                  );
}  // while end
```

Tabelle 5.3: Codegenerierung für gebundener Negative-Knoten mit Attribut-Constraints

Objekt eine Referenz zum Professor-Objekt enthält. Falls nicht, so sind die negativen Constraints für das Source-Objekt erfüllt und er kann an erfolgreich an die Source-Variable *c1* gebunden werden. Enthält das Source-Objekt eine Referenz zum Professor-Objekt, so werden die Attribut-Constraints überprüft. Schlägt die Überprüfung fehl wird eine Exception ausgelöst und das Source-Objekt kann nicht gebunden werden.

### Negativ gebundene Variable mit Attribut-Constraints

Im Vergleich zum ungebundenen Negativen-Knoten muß für die gebundene negative Variable kein Teilgraphensuche gestartet werden, um ein Source-Objekt zu binden, da die Variable schon gebunden ist. Daher bezieht sich die Negation der Variable semantisch auch nicht auf die Variable selbst, sondern auf die mit ihr verknüpften Attribut-Constraints. Die gebundene Variable darf nicht alle Attributbedingungen gleichzeitig erfüllen. Werden alle Attributbedingungen erfüllt, so führt das zum Abbruch der Teilgraphensuche. Die Teilgraphensuche hat also nur die Aufgabe sicherzustellen, daß sie mindesten eines der Attributbedingungen nicht erfüllt.

Die Tabelle 5.3 enthält ein Beispiel mit einer negativen gebundenen Variable *c1*. Die Variable *c1* ist mit zwei Attributbedingungen verknüpft. Die Vorlesung *c1* darf semantisch keine Vorlesung für den Fachbereich 5 und für das Grundstudium sein. Erfüllt die Variable *c1* alle Attribut-Constraints so wird durch die Methode `JavaSDM.ensure()` eine Exception ausgelöst und die

Teilgraphensuche abgebrochen. Weil eine negative gebundene Variable abgesehen von der Negation einer gebundenen Variable semantisch äquivalent ist, darf sie auch Links zu negativen Knoten enthalten. Zur Veranschaulichung hat die Variable einen zu-n Link zum negativen Knoten *s1*. Durch den negativen Knoten darf die Vorlesung semantisch von keinen Studenten im Grundstudium und im Fachbereich 5 gehört werden. Der Code für den negativen Knoten verändert sich nicht durch die negative gebundene Source-Variable.

### Negativer-Knoten mit Vererbung

Besteht eine negative Variable aus einem abgeleiteten Typ, so muß die Vererbung auch in der Teilgraphensuche beachtet werden. Denn die Source-Variable darf semantisch keine Links zu Objekten vom Typ des negativen Knoten oder zu Objekten, deren Typ direkt oder indirekt vom Typ des negativen Knoten abgeleitet wurde besitzen. Bei einem zu-1 Link der Source-Variable zum negativen Knoten wird zuerst überprüft, ob die Objektreferenz des Assoziationsattributs ungleich `null` ist. Ist die Objektreferenz gleich `null`, so ist eine Überprüfung der negativen Constraints nicht erforderlich und das Source-Objekt kann erfolgreich an der Source-Variable gebunden werden. Eine Überprüfung des Typs des Target-Objektes muß vorgenommen werden, wenn die Objektreferenz des Assoziationsattributs ungleich `null` ist. Aufgrund der „is-a“ Beziehung der Vererbung genügt es nur zu überprüfen, ob das Target-Objekt eine Instanz vom Typ des negativen Knoten ist. Ist das Objekt eine Instanz vom Typ des Negativen-Knoten, so wird eine Exception ausgelöst mit der Konsequenz, daß die Teilgraphensuche für das aktuelle Source-Objekt erfolglos abbricht. Ist das Target-Objekt eine direkte oder indirekte Oberklasse vom Typ des Negativen-Knoten, so kann das Source-Objekt an die Source-Variable gebunden werden.

Handelt es sich um einen zu-n Link der Source-Variable zum negativen Knoten so wird zuerst kontrolliert, ob das Container-Objekt des Assoziationsattributs, das die Assoziation zum negativen Knoten implementiert, leer ist. Bei einem leeren Container-Objekt kann die Teilgraphensuche für das Source-Objekt beendet werden und das Source-Objekt ohne Probleme an die Source-Variable gebunden werden. Enthält das Container-Objekt Elemente, so muß jedes Element im Container kontrolliert werden, ob es eine Instanz vom Typ des negativen Knoten ist. Damit die Teilgraphensuche erfolgreich für das Source-Objekt beendet werden kann, müssen alle Elemente im Container-Objekt ein Vorfahre vom Typ des negativen Knoten in der Vererbungshierarchie sein. Ist nur ein Element vom Typ des negativen Kno-

ten, so kann das Source-Objekt nicht mehr an die Source-Variable gebunden werden.

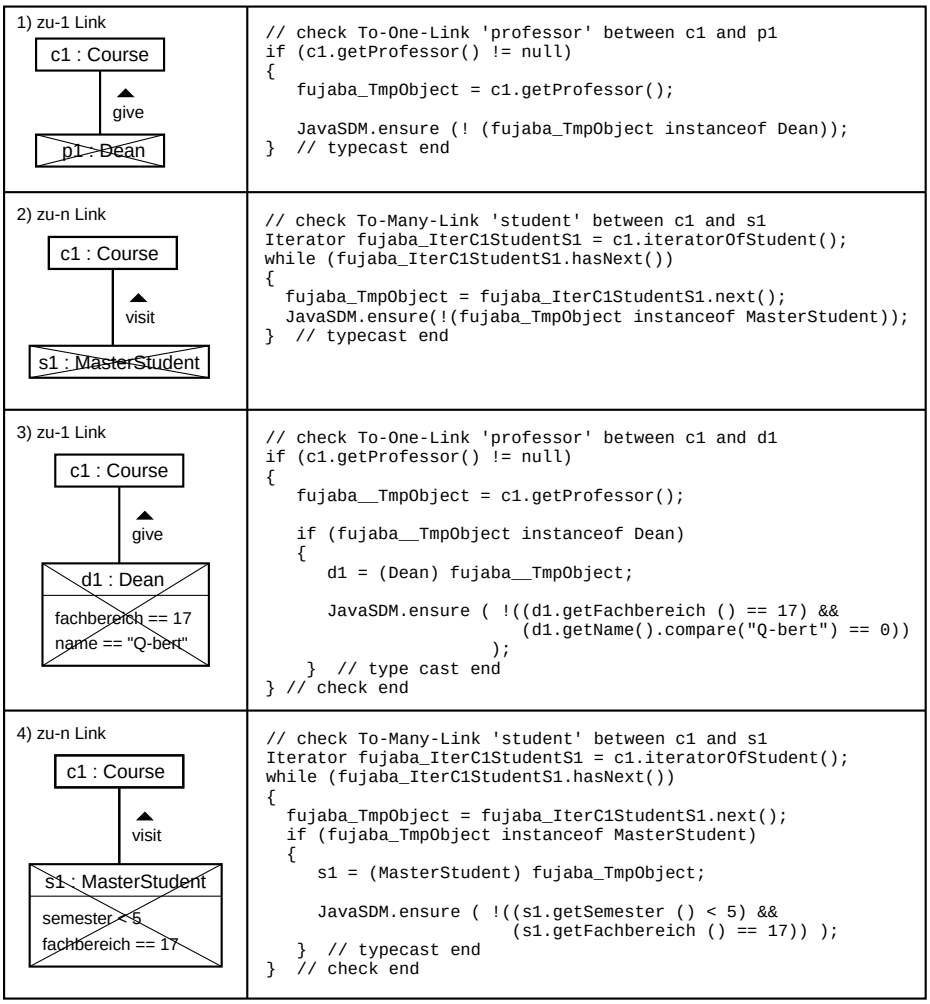



Tabelle 5.4: Codegenerierung für Negative-Knoten mit Vererbung

Besitzt der negative Knoten auch noch Attribut-Constraints, so beziehen sich die Attributbedingungen nur auf Instanzen vom Typ des negativen Knoten. Die Attributbedingungen müssen also für Objekte, deren Typ eine direkte oder indirekte Oberklasse vom Typ des negativen Knoten ist nicht beachtet werden. Bei der Teilgraphensuche wird wie im obigen Fall für den zu-1 und zu-n Link geschildert vorgegangen. Der Unterschied ist, daß jetzt die Teilgraphensuche nicht sofort für das Source-Objekt erfolglos beendet wird, wenn das Target-Objekt eine Instanz vom Typ des Negativen-Knoten ist. Stattdes-

sen wird anschließend die Attributbedingungen für das Target-Objekt überprüft. Erst wenn das Target-Objekt alle Attribut-Constraints erfüllt scheitert die Teilgraphensuche für das Source-Objekt. Erfüllt das Target-Objekt mindesten eines der Attributbedingungen nicht, so kann das Source-Objekt erfolgreich gebunden werden.

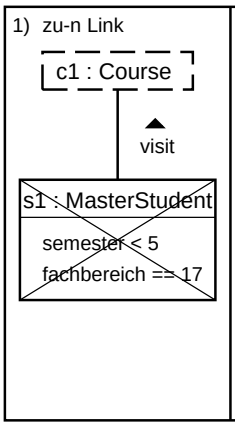
In der Tabelle 5.4 sind zur Veranschaulichung vier Beispiele mit Code abgedruckt. Die ersten beiden Beispiele zeigen den Code für einen negativen Knoten, dessen Typ von einer Klasse abgeleitet wurde. Die Klasse *MasterStudent* ist eine Ableitung der Klasse *Student*, während die Klasse *Dean* (Dekan) eine Ableitung der Klasse *Professor* ist. Der generierte Code entspricht der Teilgraphensuche wie sie oben beschrieben wurde. Die letzten beiden Beispiele haben am negativen Knoten zusätzliche Attributbedingungen. Im dritten Beispiel wird in der ersten `if`-Anweisung zuerst überprüft, ob die Source-Variable *c1* eine Referenz zu einem Objekt besitzt. Ist die Referenz ungleich `null` wird anschließend kontrolliert, ob das referenzierte Objekt eine Instanz der Klasse *Dean* ist. Die Attributbedingungen an der Target-Variablen *p1* werden nur überprüft, wenn die Objektreferenz auf einen Dekan-Objekt verweist. Hat das Dekan-Objekt den Namen „Q-bert“ und ist im Fachbereich 17, so wird eine Exception durch die Methode `JavaSDM.ensure()` ausgelöst.

Der implementierte Code für die Teilgraphensuche im vierten Beispiel durchläuft mit Hilfe eines Iterators in einer `while`-Schleife alle Elemente im Container-Objekt des Assoziationsattributs, welches die Assoziation von der Klasse *Course* zu der Klasse *Student* implementiert. In der `while`-Schleife werden alle Objekte im Container, die eine Instanz der Klasse *MasterStudent* sind, auf ihre Attributbedingungen überprüft. Erfüllt nur ein Objekt im Container-Objekt alle Bedingungen, so wird die Schleife durch eine Exception verlassen und die Teilgraphensuche für das aktuelle Vorlesungs-Objekt abgebrochen.

### Negative-Knoten an einer optionalen Variable

Besitzt die optionale Source-Variable einen Link zu einem negativen Knoten so muß die Teilgraphensuche zuerst überprüfen, ob die optionale Variable ungleich `null` ist bevor sie mit der Überprüfung der negativen Constraints beginnen kann. Ist die optionale Variable gleich `null` ist eine Kontrolle der negativen Constraints auch nicht erforderlich. Der generierte Code für die negativen Constraints verändert sich nicht durch die optionale Source-Variable außer, daß die negativen Constraints nur überprüft werden, wenn die optionale Variable durch ein Objekt gebunden wurde. In der Tabelle 5.5 ist ein

Beispiel abgebildet.




```
// check To-Many-Link 'student' between c1 and s1
if (c1 != null)
{
  Iterator fujaba_IterC1StudentS1 = c1.iteratorOfStudent();
  while (fujaba_IterC1StudentS1.hasNext())
  {
      fujaba_TmpObject = fujaba__IterC1StudentS1.next();
      if (fujaba_TmpObject instanceof MasterStudent)
      {
         s1 = (MasterStudent) fujaba_TmpObject;

         JavaSDM.ensure ( !((s1.getSemester () < 5) &&
                            (s1.getFachbereich () == 17))
                        );
      }
  }
}  // optional end
```

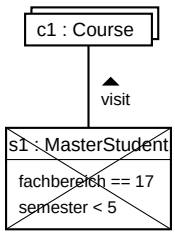
Tabelle 5.5: Negative-Knoten an einer optionalen Variable

### Negative-Knoten an einer Mengenvariablen

Eine ungebundene Mengenvariable, die mit einem negativen Knoten durch einen Link verbunden ist, darf semantisch in seiner Menge keine Objekte besitzen, die durch den negativen Knoten verboten wurden sind. In der implementierten Teilgraphensuche wird zuerst die Mengenvariable erzeugt. Das heißt die Menge, implementiert durch einen Container, wird zuerst mit allen Objekten aus der Laufzeitobjektstruktur gefüllt, die für die Bindung an der Mengenvariablen in Frage kommen können. Erst nachdem die Menge „gefüllt“ wurde, werden die negativen Constraints für die Menge in der implementierten Teilgraphensuche betrachtet. Alle Elemente in der Mengenvariablen werden jetzt durchlaufen und auf die negativen Constraints überprüft. Erfüllt ein Objekt die negativen Constraints nicht, so wird es aus der Menge entfernt.

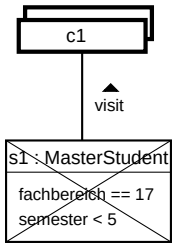
In der Tabelle 5.6 ist im ersten Beispiel der entsprechende Code für eine negative ungebundene Mengenvariable abgedruckt. Der Codeausschnitt bezieht sich nur auf den generierten Code für den negativen Knoten *s1*. Die äußere `while`-Schleife iteriert durch alle Kurs-Objekte in der Menge *c1*. Für jedes Kurs-Objekt in der Mengen wird einzeln die durch den negativen Knoten *s1* definierten Constraints überprüft. Die Überprüfung der negativen Constraints findet im `try`-Block statt. Verletzt ein Kurs-Objekt die negativen Constraints, so wird eine Exception ausgelöst. Im `catch`-Block wird die entsprechende Exception abgefangen und anschließend das aktuelle Kurs-

1) zu-n Link




```
fujaba__EnumC1 = c1.iterator();
while  (fujaba_EnumC1.hasNext() )
{
  Course fujaba_Course = (Course) fujaba_EnumC1.next();
  try
  {
     // check To-Many-Link 'student' between c1 and s1
     Iterator fujaba_Iter_Student = fujaba_Course.iteratorOfStudent ();
     while (fujaba_Iter_Student.hasNext ())
     {
        fujaba_TmpObject = fujaba_Iter_Student.next ();
        if (fujaba_TmpObject instanceof MasterStudent)
        {
            s1 = (MasterStudent) fujaba_TmpObject;

            JavaSDM.ensure ( !((s1.getFachbereich () == 17) &&
                               (s1.getSemester () < 5)) );
        }  // typecast end

     }  // check end

  }
  catch (JavaSDMException fujaba__InternalException)
  {
     fujaba__EnumC1.remove();
  }  // try catch

} // while end
```

2) zu-n Link




```
fujaba__EnumC1 = c1.iterator();
while  (fujaba_EnumC1.hasNext() )
{
  Course fujaba_Course = (Course) fujaba_EnumC1.next();

  // check To-Many-Link 'student' between c1 and s1
  Iterator fujaba_Iter_Student = fujaba_Course.iteratorOfStudent ();
  while (fujaba_Iter_Student.hasNext ())
  {
     fujaba_TmpObject = fujaba_Iter_Student.next ();
     if (fujaba_TmpObject instanceof MasterStudent)
     {
        s1 = (MasterStudent) fujaba_TmpObject;

        JavaSDM.ensure ( !((s1.getFachbereich () == 17) &&
                           (s1.getSemester () < 5)) );
     }  // typecast end

  }  // check end

} // while end
```

Tabelle 5.6: Negative-Knoten an einer ungebundenen Mengenvariablen

Objekt, das die negativen Constraints nicht erfüllt hat, aus der Menge *s1* entfernt.

Was man im Beispiel sieht ist, daß der Code im `try`-Block identisch ist mit dem generierten Code für das Beispiel 4 in der Tabelle 5.4. Der Code zur Überprüfung der negativen Constraints bleibt unverändert auch wenn die Source-Variable eine Menge ist. Ist die Source-Variable eine Menge, dann wird der Code zur Überprüfung der negativen Constraints lediglich durch einen Code-Rahmen eingekapselt und dadurch angepaßt. Der Code-Rahmen besteht aus der `while`-Schleife, die durch alle Elemente in der Menge iteriert und dem `try-catch`-Block innerhalb der Schleife. Wie im Beispiel zu sehen

ist, kommt der unveränderte Code für den negativen Knoten einfach in den `try`-Block rein. Wird eine Exception im `try`-Block ausgelöst, so wird sie im `catch`-Block abgefangen und das Element, das die Exception ausgelöst hat, aus der Menge entfernt.

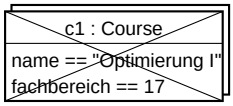
Bei einer *gebundenen* Menge ändert sich die Teilgraphensuche für die Source-Variable ein wenig. Verletzt ein Objekt in der gebundenen Menge die durch den negativen Knoten spezifizierten Bedingungen, so wird entsprechend der Semantik jetzt die Teilgraphensuche ohne Erfolg abgebrochen. Das Objekt, das die negativen Bedingungen nicht erfüllt hat, wird also nicht einfach aus der Menge entfernt, wie im obigen Fall. Würde man das Objekt entfernen, so würde man die Semantik einer gebundenen Variable verletzen.

Das zweite Beispiel in der Tabelle 5.6 zeigt den Code für eine gebundene Mengenvariable. Der Code entspricht größtenteils dem Code aus dem ersten Beispiel. Der einzige Unterschied ist, daß der `try-catch`-Block entfernt wurde. Wird eine Exception ausgelöst, so führt das jetzt zum Abbruch der Teilgraphensuche.

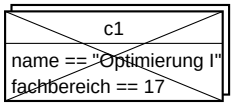
### Negative-Menge

Eine negative markierte ungebundene Menge kann es, wie im Kapitel 2.2 Abschnitt »Negative-Menge« beschrieben, nur mit Attributbedingungen geben. Die Negation bezieht sich dabei auf die mit der Menge verknüpften Attributbedingungen. In der Mengenvariable dürfen also nur Elemente rein, die mindesten eines der Attributbedingungen nicht erfüllen. Die Teilgraphensuche iteriert also durch alle Elemente in der Menge und überprüft dabei die Attribut-Constraints für jedes einzelne Element. Erfüllt ein Element alle Attributbedingungen, so wird das Element aus der Mengenvariablen entfernt. Besitzt die Negative-Menge noch Links zu negativen Knoten, so wird zusätzlicher Code erzeugt wie er im obigen Abschnitt beschrieben wurde.

Ist die Menge gebunden so wird die Teilgraphensuche abgebrochen, wenn eines der Objekte in der Menge alle Attributbedingungen erfüllt. Die Tabelle 5.7 zeigt zwei Beispiele, dessen generierter Code genau so wie oben beschrieben vorgeht.

<table>
<tr>
<td>1) Negative-Menge<br>ungebunden<br>  <br></td>
<td><pre>
// precondition check
fujaba__IterC1 = c1.iterator ();
while  (fujaba_IterC1.hasNext () )
{
  Course fujaba_Course = (Course) fujaba__IterC1.next();

  if( (fujaba_Course.getName () != null) &&
      (fujaba_Course.getName ().compareTo("Optimierung I") == 0)
       && (fujaba_Course.getFachbereich () == 17)
    )
  {
     fujaba_IterC1.remove ();
  }  // if

}  // while
</pre></td>
</tr>
<tr>
<td>1) Negative-Menge<br>gebunden<br>  <br></td>
<td><pre>
// precondition check
fujaba__IterC1 = c1.iterator ();
while  (fujaba_IterC1.hasNext () )
{
  Course fujaba_Course = (Course) fujaba__IterC1.next();

  JavaSDM.ensure(! (fujaba_Course.getName () != null) &&
      (fujaba_Course.getName ().compareTo("Optimierung") == 0) &&
      (fujaba_Course.getFachbereich () == 17)
    )

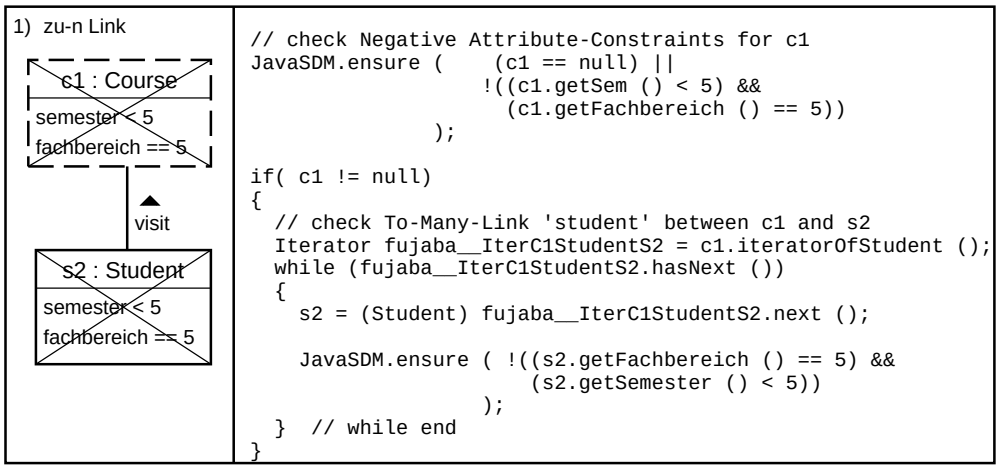
}  // while
</pre></td>
</tr>
</table>

Tabelle 5.7: Negative Menge

### Negative optionale Variable

Für eine negative optionale Variable gilt der selbe Sachverhalt wie bei der Negativen-Menge. Sie kann es nur mit Attributbedingungen geben und die Negation bezieht sich nur auf die Attributbedingungen und nicht auf die „nicht Existenz“ der Variable. Bevor die Attribut-Constraints überprüft werden können, wird die Teilgraphensuche erst kontrollieren, ob die optionale Variable auch mit einem Objekt gebunden ist. Stellt die Teilgraphensuche fest, daß die optionale Variable gebunden ist, so beginnt sie mit der Überprüfung der Attributbedingungen.

In der Tabelle 5.8 ist ein Beispiel mit Code abgedruckt. Der Code für die negative optionale Variable *c1* wird durch die erste `JavaSDM.ensure()` Methode erzeugt. Ist die optionale Variable gebunden und erfüllt sie alle Attributbedingungen, so wird eine Exception ausgelöst. Die optionale Variable ist durch einen Link noch mit einem negativen Knoten verbunden. Die durch den negativen Knoten spezifizierten Constraints werden aber nur überprüft, wenn die optionale Variable ungleich `null` ist. Der negative Knoten soll nur veranschaulichen, daß eine negative optionale Variable mit Attributbedingungen auch Links zu Negativen-Knoten besitzen kann.




```
// check Negative Attribute-Constraints for c1
JavaSDM.ensure (    (c1 == null) ||
                   !((c1.getSem () < 5) &&
                     (c1.getFachbereich () == 5))
               );

if( c1 != null)
{
  // check To-Many-Link 'student' between c1 and s2
  Iterator fujaba__IterC1StudentS2 = c1.iteratorOfStudent ();
  while (fujaba__IterC1StudentS2.hasNext ())
  {
    s2 = (Student) fujaba__IterC1StudentS2.next ();

    JavaSDM.ensure ( !((s2.getFachbereich () == 5) &&
                       (s2.getSemester () < 5))
                   );
  }  // while end
}
```

Tabelle 5.8: Negative optionale Variable

## 5.3 Negative-Links

### Negative-Links mit gebundenen Variablen

Hat eine Source-Variable einen negativen Link zu einer gebundenen Variable, so muß die Teilgraphensuche überprüfen, ob die Source-Variable einen Link zum Target-Objekt besitzt. Besitzt das Source-Objekt einen Link zum Target-Objekt, so schlägt die Teilgraphensuche für das Source-Objekt fehl. Die Kardinalität des negativen Link zur Target-Variablen muß bei der Überprüfung der negativen Constraints in der implementierten Teilgraphensuche berücksichtigt werden. Bei einem zu-1 Link muß nur überprüft werden, ob der Link mit dem Target-Objekt verbunden ist. Bei einem zu-n Link ist der Suchaufwand schon größer. Denn die Teilgraphensuche muß durch alle Links traversieren und überprüfen, ob ein Link mit dem Target-Objekt verbunden ist. Ist nur ein Link mit dem Target-Objekt verbunden, so wird die Teilgraphensuche für das Source-Objekt beendet. Das Source-Objekt kann dann nicht mehr an die Source-Variable gebunden werden.

Die Methode `c1.hasInStudent()` im ersten Beispiel der Tabelle 5.9 überprüft, ob die Source-Variable *c1* einen Link zum Target-Object *s1* besitzt. Besitzt die Source-Variable *c1* einen Link zum Target-Objekt *s1* so wird eine Exception ausgelöst und die Teilgraphensuche für das Source-Objekt *c1* ohne Erfolg beendet. Im zweiten Beispiel wird zuerst kontrolliert, ob die Variable *c1* überhaupt auf ein Professor-Objekt referenziert. Falls ja, so wird

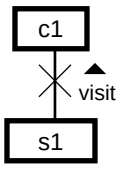
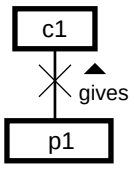
| | |
|---|---|
| 1) zu-n Link   | `// check To-Many-Link 'student' between c1 and s1`<br>`JavaSDM.ensure (!c1.hasInStudent (s1));` |
| 2) zu-1 Link   | `// check To-One-Link 'professor' between c1 and p1`<br>`JavaSDM.ensure ( c1.getProfessor() == null ||`<br>`!c1.getProfessor().equals (p1)`<br>`);` |

Tabelle 5.9: Negative-Links mit gebundenen Variablen

überprüft, ob das Professor-Objekt mit dem Target-Objekt *p1* identisch ist. Sind beide Objekte identisch wird die Teilgraphensuche abgebrochen.

### Qualifizierte Negative-Links

Bei einem qualifizierten Link wird ein Schlüssel mit einem oder mehreren Objekten assoziiert. Ist der qualifizierte Link negativ, so darf die Source-Variable entsprechend keine Links zu den assozierten Objekten besitzen. Die Semantik ist bei einer gebundenen und ungebundenen Target-Variablen unterschiedlich und entsprechend unterscheidet sich auch die Vorgehensweise in der implementierten Teilgraphensuche. Bei einer ungebundenen Target-Variable muß die implementierte Teilgraphensuche überprüfen, ob die Source-Variable unter dem angegebenen Schlüssel Links zu Objekten besitzt. Sind unter dem Schlüssel Objekte zu finden, so schlägt die Suche für das Source-Objekt fehl. Ob unter einem Schlüssel nur ein oder mehrere Objekte assoziiert werden können, hängt von der Kardinalität des negativen Links zur Target-Variablen ab. Bei einem zu-1 Link kann mit einem Schlüssel nur ein Objekt assoziiert werden, während es bei einem zu-n Link mehrere Objekte sein können. Ist die Target-Variable gebunden, so überprüft die implementierte Teilgraphensuche, ob es unter den angegebenem Schlüssel das gebundene Target-Objekt findet. Bei einem zu-1 Link muß sie nur das eine assoziierte Objekt mit der Target-Variable vergleichen. Weil bei einem zu-n Link unter dem angegebenen Schlüssel eine Menge von Objekten zurückgegeben werden kann, unter denen sie es mit dem Target-Objekt vergleicht, kann die Suche natürlich auch

entsprechend länger dauern.

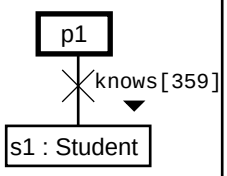
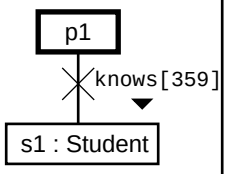
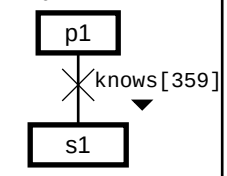
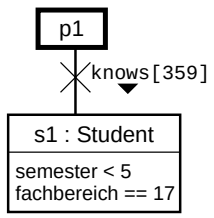
| | |
|---|---|
| 1) zu-n Link   | `// check To-Many-Link 'student' between p1 and s1`<br>`JavaSDM.ensure (!p1.hasKeyInStudent (359));` |
| 2) zu-1 Link   | `// check To-One-Link 'student' between p1 and s1`<br>`JavaSDM.ensure (p1.getFromStudent(359) == null);` |
| 3) zu-1 oder zu-n Link   | `// check To-One-Link 'student' between p1 and s1`<br>`JavaSDM.ensure (!p1.hasInStudent(359, s1));` |

Tabelle 5.10: Qualifizierte Negative-Links

In der Tabelle 5.10 sind die eben besprochenen Fälle als Beispiel abgedruckt. Im ersten Beispiel wird durch die Methode `p1.hasKeyInStudent()` überprüft, ob das Source-Objekt *p1* über den Schlüssel „359“ Objekte assoziiert hat. Falls ja wird `true` ansonsten `false` zurückgegeben. Hat das Source-Objekt *p1* unter dem gegebenen Schlüssel Objekte assoziiert, so wird eine Exception ausgelöst und die Teilgraphensuche für Source-Objekt *p1* beendet. Die Methode `p1.getFromStudent()` im zweiten Beispiel liefert das zu einem Schlüssel assoziierte Objekt zurück. Weil der negative Link nur die Kardinalität `1` besitzt, kann mit dem Schlüssel auch nur ein Objekt assoziiert werden. Es genügt deshalb mit der Methode `p1.getFromStudent()` den Rückgabewert der Methode nur auf `null` zu überprüfen. Ist der Rückgabewert gleich `null`, so kann die Suche ohne Probleme fortgesetzt werden. Die Target-Variable *s1* im dritten Beispiel ist diesmal gebunden. Durch die Methode `p1.hasInStudent()` wird kontrolliert, ob das Source-Objekt *p1* das gebundene Target-Objekt *s1* unter dem Schlüssel „359“ kennt. Gibt es das Target-Objekt *s1* unter dem Schlüssel „359“, so wird eine Exception ausgelöst und die implementierte Teilgraphensuche abgebrochen.

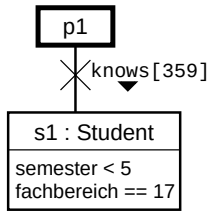
Besitzt die Target-Variable Attribut-Constraints, so muß die implementierte Teilgraphensuche die mit dem Schlüssel assoziierten Objekten darauf kontrollieren. Bei einem zu-n Link zur Target-Variablen muß von der Source-Variable aus alle Objekte, die mit dem Schlüssel des negativen Link assoziiert wurden, durchlaufen und kontrolliert werden. Bei einem zu-1 Link wird zuerst überprüft, ob die Source-Variable unter dem angegebenen Schlüssel ein Objekt assoziiert hat. Falls ja, dann wird das Objekt auf die Attributbedingungen überprüft. In der Tabelle 5.11 sind die entsprechenden Beispiele abgedruckt.

1) zu-n Link




```
// check To-Many-Link 'student' between p1 and s1
Iterator fujaba__IterP1StudentS1 = p1.iteratorOfStudent (359);
while (fujaba__IterP1StudentS1.hasNext ())
{
  s1 = (Student) fujaba__IterP1StudentS1.next ();

  JavaSDM.ensure ( !((s1.getSemester () < 5) &&
                     (s1.getFachbereich () == 17))
                      );
}  // while end
```

2) zu-1 Link




```
// check To-One-Link 'student' between p1 and s1
if (p1.getFromStudent(359) != null)
{
  s1 = p1.getFromStudent(359);

  JavaSDM.ensure ( !((s1.getSemester () < 5) &&
                     (s1.getFachbereich () == 17))
                  );
}  // if end
```

Tabelle 5.11: Qualifizierte Negative-Links mit Attribut-Constraints

Im ersten Beispiel wird der Methode `p1.iteratorOfStudent()` als Parameter der aktuelle Schlüssel übergeben. Als Rückgabewert wird ein Iterator übergeben mit dem über die assoziierten Objekte des Schlüssels iteriert werden kann. In der `while`-Schleife werden alle Studenten-Objekte durchlaufen, die der Professor *p1* unter dem Schlüssel „359“ kennt. Verletzt ein Studenten-Objekt die negativen Constraints, so wird für das Source-Objekt die Suche abgebrochen. Im zweiten Beispiel wird zuerst überprüft, ob das Source-Objekt *p1* einen Studenten unter dem Schlüssel „359“ referenziert. Falls ja, so wird anschließend für das Objekt die Attribut-Constraints kontrolliert. Werden alle Attributbedingungen erfüllt, so wird die Teilgraphensuche ohne Erfolg beendet.

### Rückwärts-Qualifizierte Negative-Links

In der Teilgraphensuche ist es möglich einen negativen qualifizierten Link rückwärts zu durchlaufen. Das durch den Schlüssel qualifizierte Objekt ist also das Source-Objekt, während der Qualifizierer das Target-Objekt ist. Die implementierte Teilgraphensuche muß aber weiterhin garantieren, daß der negative qualifizierte Link seine Gültigkeit hat. Bei einem zu-n Link zur Target-Variablen muß sie also durch alle möglichen Target-Objekte in der Laufzeitobjektstruktur traversieren und prüfen, ob das Target-Objekt unter dem gegebenen Schlüssel das Source-Objekt assoziiert hat. Besitzt das Target-Objekt unter dem gegebenen Schlüssel einen Link zum Source-Objekt, so wird für das Source-Objekt die Suche ohne Erfolg abgebrochen. Hat der qualifizierte Link eine Kardinalität von 1 zur Target-Variablen, dann muß bei der implementierte Teilgraphensuche auch nur ein Objekt kontrolliert werden. Ist die Target-Variable mit Attributbedingungen verknüpft, so werden sie erst überprüft, wenn das Target-Objekt unter dem Schlüssel einen Link zum Source-Objekt besitzt. Wird mindesten eines der Attributbedingungen nicht erfüllt, so kann das Source-Objekt erfolgreich an der Source-Variable gebunden werden. In der Tabelle 5.12 sind die entsprechenden Beispiele für die oben genannten Fälle zu finden.

Im ersten Beispiel wird in einer `while`-Schleife durch alle Professoren-Objekte iteriert, auf die die Source-Variable *s1* einen Link besitzt. Für jeden Professor-Objekt wird einzeln überprüft, ob er den Studenten *s1* unter dem Schlüssel „359“ kennt. Im zweiten Beispiel erfolgt die selbe Prüfung, aber nur mit maximal einem Professor-Objekt aufgrund des zu-1 Links zur Target-Variablen. Schlägt die Überprüfung fehl, so wird eine Exception ausgelöst und die Teilgraphensuche beendet. Bei den letzten beiden Beispielen müssen die Attributbedingungen an der Target-Variablen noch berücksichtigt werden. Die Attribut-Constraints werden nur überprüft, wenn das aktuelle Professor-Objekt unter dem Schlüssel „359“ das Source-Objekt *s1* assoziiert hat. Schlägt die Kontrolle für mindesten eine Attributbedingung fehl, so kann das Source-Objekt mit Erfolg gebunden werden.

### Negativer-Link an einer gebundenen Menge

Bei einer Source-Variable die einen negativen Link zu einer gebundenen Menge besitzt muß die implementierte Teilgraphensuche sich vergewissern, daß die Source-Variable keinen Link zu eines der Objekte in der Menge besitzt. Dazu durchläuft sie alle entsprechenden Target-Objekte zu der die

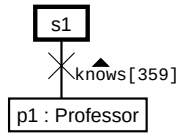
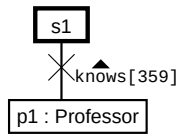
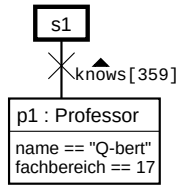
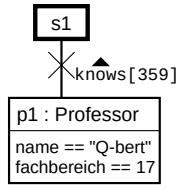
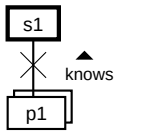
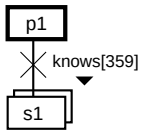
| | |
|---|---|
| 1) zu-n Link<br>  | `// check To-Many-Link 'professor' between s1 and p1`<br>`Iterator fujaba__IterS1ProfP1 = s1.iteratorOfProfessor ();`<br>`while (fujaba__IterS1ProfP1.hasNext ())`<br>`{`<br>`  p1 = (Professor) fujaba__IterS1ProfP1.next ();`<br><br>`  JavaSDM.ensure(!p1.hasInStudent(359, s1))`<br>`}  // while end` |
| 2) zu-1 Link<br>  | `// check To-One-Link 'professor' between s1 and p1`<br>`if (s1.getProfessor() != null)`<br>`{`<br>`  p1 = s1.getProfessor();`<br><br>`  JavaSDM.ensure (!p1.hasInStudent(359, s1));`<br>`}  // if end` |
| 3) zu-n Link<br>  | `// check To-Many-Link 'professor' between s1 and p1`<br>`Ierator fujaba__IterS1ProfessorP1 = s1.iteratorOfProfessor ();`<br>`while (fujaba__IterS1ProfessorP1.hasNext ())`<br>`{`<br>`  p1 = (Professor) fujaba__IterS1ProfessorP1.next ();`<br><br>`  if(p1.hasInStudent(359, s1))`<br>`  {`<br>`  JavaSDM.ensure ( !((p1.getName () != null) &&`<br>`  (p1.getName ().compareTo ("Q-bert") == 0) &&`<br>`  (p1.getFachbereich () == 17))`<br>`  );`<br>`  }  // reversed qualified Link end`<br>`}  // while end` |
| 4) zu-1 Link<br>  | `// check To-One-Link 'professor' between s1 and p1`<br>`if (s1.getProfessor() != null)`<br>`{`<br>`  p1 = s1.getProfessor();`<br><br>`  if(p1.hasInStudent(359, s1))`<br>`  {`<br>`  JavaSDM.ensure ( !((p1.getName () != null) &&`<br>`  (p1.getName ().compareTo ("Q-Bert") == 0) &&`<br>`  (p1.getFachbereich () == 17)) );`<br><br>`  }  // reversed qualified Link end`<br>`}  // if end` |

Tabelle 5.12: Rückwärts-Qualifizierte Negative-Links

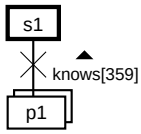
Source-Variable einen entsprechenden Link besitzt und überprüft, ob das Target-Objekt sich in der gebundenen Menge befindet. Befindet sich eines der Target-Objekte in der Menge, so wird die Suche abgebrochen. In der Tabelle 5.13 ist ein Beispiel abgebildet, daß Vorgang besser veranschaulicht (erstes Beispiel).

Im Beispiel wird in einer `while`-Schleife durch alle Professor-Objekte iteriert, zu denen die Source-Variable *s1* einen Link besitzt. Bei jedem Schleifendurchlauf wird kontrolliert, ob sich ein Professor-Objekt in der Menge *p1* befindet. Die Überprüfung geschieht durch die Methode `p1.contains()`. Befindet sich ein Professor-Objekt in der Menge, so wird eine Exception ausgelöst mit der Konsequenz, daß die Teilgraphensuche für das Source-Objekt

<table>
<tr>
<td>1) zu-n Link<br> 
</td>
<td><pre>
// check To-Many-Link 'professor' between s1 and p1
Iterator fujaba_IterS1ProfP1 = s1.iteratorOfProfessor ();
while (fujaba_IterS1ProfessorP1.hasNext ())
{
   Professor fujaba_TmpObjectProf = (Professor) fujaba_IterS1ProfP1.next();

   JavaSDM.ensure(!p1.contains(fujaba_TmpObjectProf));
}  // while end
</pre></td>
</tr>
<tr>
<td>2) zu-n Link<br> 
</td>
<td><pre>
// check To-Many-Link 'student' between p1 and s1
Iterator fujaba__IterP1StudentS1 = p1.iteratorOfStudent (359);
while (fujaba_IterP1StudentS1.hasNext ())
{
    Student fujaba_TmpObjectStudent = (Student) fujaba_IterP1StudentS1.next ();

    JavaSDM.ensure(!s1.contains(fujaba_TmpObjectStudent));

}  // while end
</pre></td>
</tr>
<tr>
<td>3) zu-n Link<br> 
</td>
<td><pre>
// check To-Many-Link 'professor' between s1 and p1
Iterator fujaba_IterS1ProfP1 = s1.iteratorOfProfessor ();
while (fujaba_IterS1ProfP1.hasNext ())
{
  Professor fujaba_TmpObjectProf = (Professor) fujaba_IterS1ProfP1.next ();

  if( p1.contains(fujaba__TmpObjectProf) )
  {
     JavaSDM.ensure(!fujaba__TmpObjectProf.hasInStudent(234, s1))

  }  // boundTargetSet && reversedQualifiedLink

 }  // while end
</pre></td>
</tr>
</table>

Tabelle 5.13: Negativer-Link an einer gebundenen Menge

*s1* ohne Erfolg beendet wird.

Ist der negative Link vor- oder rückwärts-qualifiziert, so muß entsprechend der Semantik der Qualifizierer mit berücksichtigt werden. Die letzten zwei Beispiele zeigen den Code für einen vor- und rückwärts-qualifizierten negativen Link an einer gebundenen Menge. Im zweiten Beispiel wird in einer `while`-Schleife nur durch die Studenten-Objekte iteriert, zu denen die Source-Variable *p1* einen qualifizierten Link mit dem Schlüssel „359“ besitzt. Befindet sich eines der assoziierten Studenten-Objekte in der gebundenen Menge *s1*, so wird die Teilgraphensuche ohne Erfolg beendet.

Beim dritten Beispiel handelt es sich um einen rückwärts-qualifizierten negativen Link zu der gebundenen Menge *p1*. In der `while`-Schleife wird durch alle Professoren-Objekte iteriert, die einen Link zum Source-Objekt *s1* besitzen. Befindet sich ein Professoren-Objekt in der Menge *p1*, so muß überprüft werden, ob das entsprechende Professoren-Objekt unter dem Schlüssel „359“ das Source-Objekt *s1* assoziiert hat. Falls ja, wird eine Exception ausgelöst und die Teilgraphensuche erfolglos abgebrochen.

## 5.4 Prioritätsklassen und Kostenbewertung

### Prioritäten *P-Check* und *P-Check-To-Many*

Mit Hilfe der Prioritätsklassen wird in Fujaba das Force-Failure-Prinzip realisiert. Die Prioritätsklassen steuern die Teilgraphensuche, das heißt sie bestimmen die richtige Abarbeitungsreihenfolge der Links mit dem Ziel den Suchaufwand so gering wie möglich zu halten. Damit eine Optimierung der Teilgraphensuche für Negative-Knoten und -Links gewährleistet ist, müssen sie notwendigerweise auch in Prioritätsklassen eingeteilt werden.

Negative Constraints mit restriktiven Bedingungen, die leicht überprüft werden können und mit einem kleinem Suchraum werden als erstes abgearbeitet. Sie bekommen eine entsprechend hohe Priorität und werden in der Abarbeitung bevorzugt. Im Vergleich dazu bekommen negative Constraints mit einem großen Suchraum und weniger restriktiven Bedingungen eine kleinere Priorität. Es werden zuerst alle Negativen-Knoten und -Links einer Prioritätsklasse abgearbeitet, bevor mit der nächst niedrigeren Prioritätsklasse die Teilgraphensuche fortgesetzt wird. Die Reihenfolge in der Negative-Knoten und -Links innerhalb der selben Prioritätsklasse abgearbeitet werden ist nicht fest definiert.

Die Negativen-Knoten und -Links werden in zwei Prioritätsklassen aufgeteilt: *P-Check* und *P-Check-To-Many*. Negative-Knoten und -Links in der Prioritätsklasse *P-Check* werden in der Teilgraphensuche vor der Prioritätsklasse *P-Check-To-Many* abgearbeitet. Die Priorität *P-Check* enthält Negative-Knoten und -Links, deren Überprüfung ohne viel Aufwand sofort erfolgen kann. Darunter fallen alle Negativen-Typen, die einen sehr kleinen Suchraum haben, das heißt einen zu-1 Link zur Target-Variablen besitzen oder mit ziemlich restriktiven Bedingungen versehen sind. Ein Negativer-Knoten mit einem zu-n Link, ohne Attributbedingungen und ohne Vererbung würde zum Beispiel unter diese Priorität fallen.

Entsprechend kommen in der Prioritätsklasse *P-Check-To-Many* alle Negativen-Knoten und -Links mit einem großen Suchaufwand und weniger restriktiven Bedingungen rein. Im Allgemeinen fallen unter dieser Prioritätsklasse alle Negativen-Typen, die alle folgenden Bedingungen erfüllen:

1. *Die Target-Variable muß ungebunden sein.* Diese Voraussetzung ist bei Negativen-Knoten immer erfüllt. Bei einem Negativen-Link muß stattdessen zuerst überprüft werden, ob die Target-Variable gebunden oder

ungebunden ist.

2. *Die Source-Variable muß mit einem zu-n Link mit der Target-Variablen verbunden sein.* Die Teilgraphensuche muß nähmlich bei einem zu-n Link einen größeren Suchraum durchlaufen um die negativen Constraints zu kontrollieren.

Zusätzlich muß eines der Bedingungen erfüllt werden:

- Die Target-Variable ist mit Attributbedingungen verknüpft.
- Der Typ der Target-Variablen besteht aus einem abgeleitetem Typ.
- Beim Negativen-Link handelt es sich um einen rückwärts qualifizierten Link.

Die drei Bedingungen zwingen die Teilgraphensuche alle Target-Objekte im Suchraum einzeln zu überprüfen. Die Prioritätsklasse *P-Check-To-Many* kommt in der Prioritätsliste nach der Priorität *P-To-One*, weil die Teilgraphensuche in dieser Prioritätsklasse einen größeren Suchraum zu überprüfen hat. In der Tabelle 5.14 sind alle Negativen-Typen mit ihrer zugehörigen Prioritätsklasse abgebildet. Negative-Typen mit einer zu-1 Assoziation wurden nicht mit abgedruckt, weil sie alle unter der Prioritätsklasse *P-Check* fallen.

Ist die Source- oder Target-Variable eine Mengenvariable, so fallen die Negativen-Typen automatisch in die Prioritätsklasse *P-Check-To-Many* rein. Denn bei einer Mengenvariable ist die Teilgraphensuche gezwungen jedes Element in der Menge einzeln zu überprüfen.

### Priorität *P-None*

Was bisher unerwähnt blieb ist der Fall, wenn die Target-Variable im Kontext von Negativen-Links mehr als nur einen Link besitzt. Sie besitzt also neben dem Negativen-Link zur Source-Variable noch andere Links zu ihren Nachbarobjekten. Wie die Teilgraphensuche in diesem Fall nun weiterläuft, hängt ganz entscheidend von der Target-Variablen ab. Ist die Target-Variable gebunden, so kann die Teilgraphensuche ganz normal wie bisher gewohnt weiterfahren und die negativen Constraints überprüfen. Also kontrollieren, ob zwischen dem Source- und Target-Objekt ein Link existiert. Bei einer ungebundenen Target-Variable darf die Teilgraphensuche den negativen Link

| | Negativer-Typ (zu-n Link) | | P-Check | P-Check-ToMany |
|---|---|---|---|---|
| mit Attribut-Constraints | Negativer-Knoten | | | ● |
| | Negativer-Knoten mit Vererbung | | | ● |
| | Negativer-Knoten an einer Mengenvariablen | | | ● |
| | Negative-Menge | | | ● |
| | Qualifizierter Negativer-Link | gebundene Target-Variable | ● | |
| | | ungebundene Target-Variable | | ● |
| | Rückwärts-Qualifizierter-Link | gebundene Target-Variable | ● | |
| | | ungebundene Target-Variable | | ● |
| | Negativer-Link an einer gebundenen Menge | | | ● |
| ohne Attribut-Constraints | Negativer-Knoten | | ● | |
| | Negativer-Knoten mit Vererbung | | | ● |
| | Negativer-Knoten an einer Mengenvariablen | | | ● |
| | Negative-Menge | | | ● |
| | Qualifizierter Negativer-Link | gebundene Target-Variable | ● | |
| | | ungebundene Target-Variable | ● | |
| | Rückwärts-Qualifizierter-Link | gebundene Target-Variable | ● | |
| | | ungebundene Target-Variable | | ● |
| | Negativer-Link an einer gebundenen Menge | | | ● |

Tabelle 5.14: Prioritätsklassen von Negativen-Knoten und -Links

nicht sofort auswerten. Wieso das so ist soll zur besseren Veranschaulichung an einem Beispiel erklärt werden. Im Beispiel wird angenommen, daß die Teilgraphensuche den negativen Link auch auswertet, wenn die Target-Variable neben dem negativen Link zur Source-Variable noch andere Links besitzt. Das Beispiel ist in der Abbildung 5.1 abgedruckt.

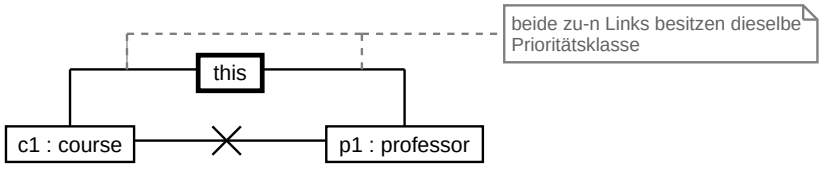



Abbildung 5.1: Ein Beispiel für die Prioritätsklasse *P-None*

Die Teilgraphensuche beginnt mit der *this*-Variablen. Von der Variablen *this* aus führen zwei zu-n Links zu den beiden ungebundenen Variablen *c1* und *p1*. Weil die beiden zu-n Links dieselbe Priorität besitzen, hat die Teilgraphensuche zwei Auswahlmöglichkeiten die Suche zu beginnen. Entscheidet

sich die Teilgraphensuche für den linken weg, so bindet sie zuerst die Variable *c1*. Anschließend wertet sie den negativen Link aus. Sie darf semantisch nur Vorlesungs-Objekte an der Variablen *c1* binden, die von keinem Studenten gehört werden. Nachdem die Variable *c1* erfolgreich gebunden wurde, versucht sie die Variable *s1* zu binden. An der Variablen *s1* darf sie semantisch nur Studenten-Objekte binden, die die Vorlesung *c1* nicht besuchen. Bei erfolgreicher Bindung der Variablen *s1* wird die Teilgraphensuche beendet. Wertet sie beim Beginn der Teilgraphensuche nicht die Variable *c1*, sondern die Variable *s1* zuerst aus so ergibt sich daraus eine andere Semantik für die beiden Variablen. Es können an der Variablen *s1* nur Studenten-Objekte gebunden werden, die keine Vorlesungen hören, während an der Variablen *c1* nur Vorlesungs-Objekte gebunden werden können, die nicht von dem Studenten *s1* besucht werden. Je nachdem in welcher Reihenfolge die Links von der Teilgraphensuche bearbeitet werden, ergibt sich daraus eine unterschiedliche Semantik für die Variablen *c1* und *s1*. Dieses Verhalten ist natürlich nicht erwünscht, weil die Semantik des Negativen-Link unabhängig von seiner Bearbeitungsreihenfolge sein sollte.

Um dies zu erreichen darf ein Negativer-Link nur ausgewertet werden, wenn die Target-Variable bereits gebunden wurde. Ist die Target-Variable ungebunden und besitzt sie mehr als einen Link, so bekommt der Negative-Link die Prioritätsklasse *P-None* zugeordnet. Das heißt der Negative-Link darf nicht ausgewertet werden, solange die Target-Variable noch ungebunden ist. Wird im Beispiel zuerst die Variable *c1* gebunden, so wird anschließend nicht mehr der Negative-Link ausgewertet, weil die entsprechende Target-Variable mehr als einen Link besitzt und noch ungebunden ist. Der Negative-Link wird erst ausgewertet nachdem die Variable *s1* gebunden wurde. Denn in diesem Fall ist die Target-Variable *c1* bereits gebunden. Bei der Auswertung des Negativen-Link wird nun überprüft, ob zwischen den beiden gebundenen Variablen ein Link existiert. Wird bei der Teilgraphensuche die Variable *s1* zuerst bearbeitet, so wird sich jetzt für die Variablen dieselbe Semantik ergeben unabhängig von der Bearbeitungsreihenfolge der Links.

# Kapitel 6

# Codegenerierung für die Multilinks

(Kan-Sing, Wan)

## 6.1 Einleitung

Multilinks erlauben bei sortierten oder geordneten zu-n Assoziationen die Ordnungsbeziehung der Objekte untereinander in einem Story-Pattern zu spezifizieren. Die Sortierung der Objekte entspricht dabei der Sortierung der Objekte innerhalb eines Containers. Die Objekte besitzen selbst keine Verweise auf andere Objekte, um eine Reihenfolgebeziehung zu implementieren. Um den Nachfolger eines Objektes zu ermitteln, muß man deshalb auf die entsprechenden Methoden des Containers zurückgreifen. Multilinks repräsentieren also keine Datenstruktur in den Objekten, die eine Sortierung oder Ordnung realisieren, sondern verdeutlichen lediglich die Ordnungsbeziehungen der Objekte untereinander innerhalb eines Containers. Für das Story-Pattern ist ein Multilink eine zusätzliche Bedingung an das Objekt, das die spezifizierte Ordnungsbeziehung erfüllen muß, bevor es an einer Variablen gebunden werden kann. Deshalb muß die bisher implementierte Teilgraphensuche erweitert werden, um die zusätzlichen Anforderungen der Multilinks zu erfüllen.

Für eine Optimierung der Teilgraphensuche ist die Einschätzung des Suchaufwandes für die einzelnen Multilink-Typen wichtig. Deshalb wird in diesem Abschnitt die implementierte Teilgraphensuche für die einzelnen

Multilink-Typen vorgestellt. Die Multilink-Pfade werden danach besprochen. Hier muß bei der Teilgraphensuche zusätzlich die transitive Eigenschaft der Multilink-Pfade beachtet werden. Die Kostenbewertung der Multilinks und die darauf aufbauende Optimierung der Teilgraphensuche für die Multilinks wird in dem Unterkapitel »Kostenbewertung von Multilinks« behandelt.

## 6.2 Codegenerierung für *first*- und *last*-Multilinks

Die *first*- und *last*-Multilinks beziehen sich im Gegensatz zu den übrigen Multilink-Typen nur auf eine Variable. Ein *first*-Multilink legt fest, daß die Variable mit dem ersten Element des Containers zu binden ist. Ein *last*-Multilink würde die Variable entsprechend mit dem letzten Element des Containers binden. Beide Mutlink-Typen können erfolgreich ausgeführt werden, solange mindestens ein Objekt in dem Container verbleibt. Enthält der Container nur ein Objekt, würden sowohl der *first*- wie auch der *last*-Multilink dasselbe Objekt an die jeweilige Variable binden. Es ist sogar möglich, daß die beiden Multilink-Typen auf denselben Link verweisen. Beide Multilinks werden in diesem Fall als Bedingung an das zu bindende Objekt betrachtet. Das Objekt muß also identisch sein mit dem ersten wie auch dem letzten Element des Containers. Diese Bedingung ist erfüllt, wenn der Container nur ein Element besitzt. Die Teilgraphensuche würde in diesem Fall zuerst versuchen das erste Element eines Containers an die Variable zu binden. Da der Container ein Element enthält, wird dieser zurückgegeben und an die Variable gebunden. Bei der anschließenden Ausführung des *last*-Multilinks ist die Variable bereits gebunden. Das *last*-Multilink ruft deshalb das letzte Element des Containers ab und vergleicht es mit dem gebunden Objekt an der Variablen. Sind beide Objekte identisch wird das Story-Pattern fortgesetzt, ansonsten wird ein Bindungsfehler generiert. Da der Container nur ein Element enthält, ist das letzte Element des Containers identisch mit dem gebundenen Objekt an der Variablen.

Um das erste oder letzte Element einer sortierten zu-n Assoziation zu bekommen, müssen die Zugriffsmethoden erweitert werden. Für den Zugriff auf das erste oder letzte Element eines Containers werden deshalb die folgenden Methoden eingeführt:

```
1 : public ELEMENT_TYPE getFirstOfCONTAINER ()
2 : {
3 :    if (CONTAINER == null)
```

```
4 :    {
5 :       return null;
6 :    }
7 :    else
8 :    {
9 :       return (ELEMENT_TYPE) CONTAINER.getFirst ();
10:    }
11: }
12:
13: public ELEMENT_TYPE getLastOfCONTAINER ()
14: {
15:    if (CONTAINER == null)
16:    {
17:       return null;
18:    }
19:    else
20:    {
21:       return (ELEMENT_TYPE) CONTAINER.getLast ();
22:    }
23: }
```

`CONTAINER` und `ELEMENT_TYP` sind jeweils Platzhalter in den Methoden und werden in der konkreten Implementierung einer geordneten oder sortierten zu-n Assoziation durch das entsprechende Container-Objekt beziehungsweise den entsprechenden Typ der Container-Elemente ersetzt. Bevor auf das Container-Objekt zugegriffen werden kann, muß überprüft werden, ob es bereits erzeugt wurde. Denn solange kein Element in dem Container eingefügt werden soll, wird kein Container-Objekt erzeugt. Diese Überprüfung wird durch den *if*-Zweig der beiden Methoden vorgenommen. Existiert ein Container werden die entsprechenden Methoden des Containers aufgerufen, um das erste beziehungsweise das letzte Element des Containers zu bekommen.

Der Container wird durch ein Objekt der Klasse *FLinkedList* realisiert. Wie der Name der Klasse schon andeutet, implementiert sie die Datenstruktur einer doppelverketteten Liste. Die Methoden `getFirst()` und `getLast()` gehören also der Klasse *FLinkedList* an und sie liefern das erste oder letzte Element einer Liste mit einer Zugriffszeit von O(1) zurück.

Die folgende Tabelle in der Abbildung 6.1 faßt die möglichen Ausprägungen der *first*- und *last*-Multilinks zusammen und zeigt den zugehörigen Code, der von Fujaba erzeugt wird, um die Teilgraphensuche zu implementieren. Die Darstellung der Multilinks wird aus Platzgründen vereinfacht dargestellt. Die Container-Variablen und die Links wurden weggelassen, da sie in jeder der Ausprägungen unverändert bleiben. Die Multilinks werden deshalb direkt an den Variablen eingezeichnet.

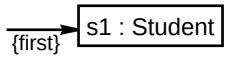
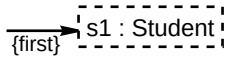
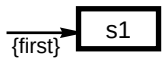
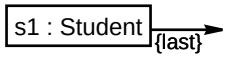
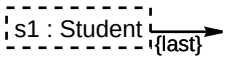
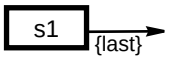
| | |
|---|---|
|  | `// multilink bind s1 : Student`<br>`s1 = this.getFirstOfStudents ();`<br>`JavaSDM.ensure (s1 != null);` |
|  | `// multilink bind s1 : Student`<br>`s1 = this.getFirstOfStudents ();`<br>`...` |
|  | `// check multilink s1 : Students`<br>`JavaSDM.ensure (this.getFirstOfStudents`<br>`().equals (s1));`<br>`...` |
|  | `// multilink bind s1 : Student`<br>`s1 = this.getLastOfStudents ();`<br>`JavaSDM.ensure (s1 != null);` |
|  | `// multilink bind s1 : Student`<br>`s1 = this.getLastOfStudents ();`<br>`...` |
|  | `// check multilink s1 : Students`<br>`JavaSDM.ensure (this.getLastOfStudents ().equals`<br>`(s1));`<br>`...` |

Tabelle 6.1: Implementierung der *first*- und *last*-Multilinks

Die Methoden der Klasse *Uni* `getFirstOfStudents()` und `getLast-OfStudents()` liefern jeweils das erste oder letzte Element des Containers `Students` zurück. Der Qualifizierer vor den Methoden verweist immer auf das entsprechende Container-Objekt des Multilinks. Die Objektreferenz `this` verweist deshalb auf eine Instanz der Klasse *Uni*.Um einen eindeutigen Namen für die Zugriffsmethoden zu generieren, setzt sich der Methodenname aus der auszuführenden Operation und den Container-Namen zusammmen. Der Methodenname für ein *first*-Multilink würde sich in diesem Fall aus der Zeichenkette `getFirst` und den Namen des Containers `Students` zusammensetzen. Die Hilfsmethode `JavaSDM.ensure(..)` wird nach dem Zugriff auf den Container aufgerufen. Sie überprüft, ob die Variable `s1` erfolgreich gebunden wurde, ansonsten löst sie ein Exception aus, um ein Bindungsfehler aufzuzeigen. Diese Situation tritt ein, wenn der Container leer ist und die Zugriffsmethode deshalb den Wert `null` zurückliefert. In den Fällen, wo eine optionale Variable gebunden werden soll, entfällt diese Überprüfung, da optionale Variablen aufgrund ihrer Semantik keine Bindungsfehler erzeugen.

## 6.3 Codegenerierung für *direct*-Multilinks

Der *direct*-Multilink spezifiziert eine direkte Nachfolger- oder Vorgängerbeziehung zwischen zwei Objekten, die derselben Assoziation angehören. Ist die Source-Variable gebunden und die Target-Variable ungebunden, muß für die ungebundene Variable ein Nachfolger in dem Container gefunden werden. Im umgekehrten Fall müßte zu der gebundenen Variablen ein Vorgänger ermittelt werden. Sind beide Variablen gebunden, muß nur überprüft werden, ob der Nachfolger der Source-Variablen identisch ist mit dem gebundenen Objekt der Target-Variablen. Unterscheiden sich die beiden Objekte, liegt ein Bindungsfehler vor, da die spezifizierten Bedingungen des *direct*-Multilinks dadurch verletzt werden. Optionale Variablen können auch zusammen mit den *direct*-Multilinks verwendet werden. Entsprechend ihrer Semantik erzeugen sie keine Bindungsfehler, wenn für das Multilink keinen Nachfolger oder Vorgänger gefunden werden konnte.

Klassen, die sortierte oder geordnete zu-n Assoziationen enthalten, müssen um entsprechende Zugriffsmethoden erweitert werden, damit bei der Anwendung von *direct*-Multilinks ein Zugriff auf den Nachfolger oder Vorgänger einer Variablen möglich ist. Nachfolgend sind die zwei Methoden abgedruckt:

```
1 : public ELEMENT_TYPE getNextOfCONTAINER (ELEMENT_TYPE refObject)
2 : {
3 :    if (CONTAINER == null)
4 :    {
5 :       return null;
6 :    }
7 :    else
8 :    {
9 :       return (ELEMENT_TYPE) CONTAINER.getNextOf (refObject);
10:    }
11: }
12:
13: public ELEMENT_TYPE getPreviousOfCONTAINER (ELEMENT_TYPE refObject)
14: {
15:    if (CONTAINER == null)
16:    {
17:       return null;
18:    }
19:    else
20:    {
21:       return (ELEMENT_TYPE) CONTAINER.getPreviousOf (refObject);
22:    }
23: }
```

Die beiden Methoden besitzen dieselbe Struktur, wie die zuvor vorgestellten Zugriffsmethoden für die *first*- und *last*-Multilinks. Vor dem Zugriff auf dem Container muß überprüft werden, ob ein Container-Objekt existiert. Das Container-Objekt ist ein Objekt der Klasse *FLinkedList*. Es mußte für den Zugriff auf den direkten Nachfolger oder Vorgänger eines Objektes entsprechend erweitert werden. Deshalb wurde die Klasse *FLinkedList* um die beiden Methoden `getNextOf (Object value)` und `getPreviousOf (Object value)` erweitert. Die Methode `getNextOf(..)` liefert zu dem übergebenen Argument den direkten Nachfolger zurück. Analog ermittelt die Methode `getPreviousOf(..)` den direkten Vorgänger.

Um den Nachfolger oder Vorgänger des als Argument übergebenen Objektes zu ermitteln, muß dieses Objekt zuerst in dem Container gefunden werden. Der Zeitaufwand für die Suchoperation kann im schlechtesten Fall O(n) betragen, wobei n gleich der Anzahl der Elemente in der Liste ist. Bei mehrfach Aufrufen dieser Methoden (z.B. während der Teilgraphensuche eines Multilink-Pfades) ergäbe sich damit eine polynomielle Laufzeit. Um die Laufzeit zu reduzieren, wird in der Liste deshalb das zuletzt zugegriffene Element zwischengespeichert. Wird die Methode `getNextOf(..)` zum Beispiel während der Teilgraphensuche für einen Multilink-Pfad mehrmals hintereinander aufgerufen, betragen die Kosten für den ersten Aufruf zwar weiterhin O(n), aber in den nachfolgenden Methodenaufrufen verursacht sie nur noch Kosten von O(1). Die hintereinander Ausführung mehreren *direct*-Multilinks verursachen also insgesamt einen Zeitaufwand von O(n), da die Gesamtzahl der Traversierungen in der Liste durch die Gesamtzahl der Elemente in der Liste beschränkt ist.

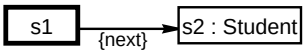
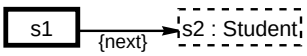
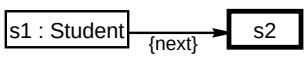
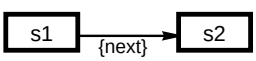
| | |
|---|---|
|  | `// multilink bind s1 : Student`<br>`s2 = this.getNextOfStudents (s1);`<br>`JavaSDM.ensure (s2 != null);` |
|  | `// multilink bind s1 : Student`<br>`s2 = this.getNextOfStudents (s1);`<br>`...` |
|  | `// multilink bind s2 : Student`<br>`s1 = this.getPreviousOfStudents (s2);`<br>`JavaSDM.ensure (s2 != null);` |
|  | `// check multilink s1 to s2`<br>`JavaSDM.ensure (this.getNextOfStudents (s1).equals (s2))`<br>`...` |

Tabelle 6.2: Implementierung des *direct*-Multilinks

Die Tabelle 6.2 zeigt die Implementierung des *direct*-Multilinks anhand einiger Beispiele. Die Methoden `getNextOfStudents()` und `getPreviousOfStudents()` geben den Nachfolger oder Vorgänger eines Objektes zurück. Das Bezugsobjekt wird den Methoden als Argument übergeben. In dem ersten Beispiel der Tabelle ist es die Variable *s1*, deren Nachfolger durch die Methode `getNextOfStudents()` ermittelt werden soll. Die Methoden geben den Wert `null` zurück, wenn kein Nachfolger oder Vorgänger zu dem Bezugsobjekt gefunden werden konnte.

Die komplette Teilgraphensuche des *direct*-Multilinks wird in den Methoden realisiert. Anhand des ersten Beispiels wird die Vorgehensweise der Methode `getNextOfStudents()` erklärt. Zu der gebunden Variablen *s1* soll der Nachfolger ermittelt und an der Variable *s2* gebunden werden. Um den Nachfolger zu ermitteln, wird zuerst das Objekt an der Variablen *s1* in dem Container gesucht. Nachdem es gefunden wurde, wird das darauffolgende Objekt von der Methode zurückgegeben. Falls es keinen Nachfolger gibt, weil das Bezugsobjekt das letzte Element des Containers ist, wird der Wert `null` zurückgegeben. Anschließend überprüft die Hilfsmethode `JavaSDM.ensure(..)`, ob die Teilgraphensuche für den Nachfolger erfolgreich war. Um den Vorgänger einer Variablen zu ermitteln, wird auf ähnlicher Weise verfahren. Sobald das Bezugsobjekt in dem Container gefunden wurde, gibt die Methode den direkten Vorgänger des Bezugsobjektes zurück. Der Rückgabewert der Methode ist `null`, wenn das Bezugsobjekt auf das erste Element des Containers verweist. Bei optionalen Variablen ändert sich die implementierte Teilgraphensuche nicht. Nur die Überprüfung mit der Hilfsmethode `JavaSDM.ensure(..)` kann in diesem Fall entfallen, wie im zweiten Beispiel der Tabelle zu sehen ist.

## 6.4 Codegenerierung für *index*-Multilinks

Der *index*-Multilink ist eine Verallgemeinerung des *direct*-Multilinks. Der *direct*-Multilink spezifizierte immer eine Ordnungsbeziehung zwischen zwei direkt benachbarten Objekten. Dagegen erlaubt der *index*-Multilink eine Ordnungsbeziehung zwischen zwei beliebigen Objekten eines Containers zu spezifizieren. Der Abstand zwischen den zwei Objekten in dem Container kann durch den Index des *index*-Multilinks spezifiziert werden. Ist die Source-Variable gebunden und der Index des *index*-Multilinks auf 1 gesetzt worden, so wird versucht den direkten Nachfolger des Source-Objektes an die Target-Variable zu binden. Der Abstand zwischen den beiden Objekten ist in diesem

Fall 0, deshalb mußt der Index um eins abgezogen werden, um den tatsächlichen Abstand zwischen den zwei Objekten zu bekommen.

Die Implementierung der Teilgraphensuche für die *index*-Multilinks wird durch die folgenden zwei Zugriffsmethoden realisiert:

```
1 : public TYPE getNextIndexOfCONTAINER (TYPE refObject, int index)
2 : {
3 :    if (CONTAINER == null)
4 :    {
5 :       return null;
6 :    }
7 :    else
8 :    {
9 :       return (TYPE) CONTAINER.getNextIndexOf (refObject, index);
10:    }
11: }
12:
13: public TYPE getPreviousIndexOfCONTAINER (TYPE refObject, int index)
14: {
15:    if (CONTAINER == null)
16:    {
17:       return null;
18:    }
19:    else
20:    {
21:       return (TYPE) CONTAINER.getPreviousIndexOf (refObject, index);
22:    }
23: }
```

Den beiden Zugriffsmethoden werden als Parameter das Bezugsobjekt und der Index übergeben. Diese Parameter werden an den entsprechenden Methoden des Containers durchgereicht, die in der Liste nach den spezifizierten Objekten suchen sollen. Zu diesem Zweck wurde die Klasse *FLinkedList* um die zwei benötigten Methoden `getNextIndexOf(..)` und `getPreviousIndexOf(..)` erweitert. Um das spezifizierte Objekt im Container zu finden, wird zuerst nach dem übergebenen Bezugsobjekt gesucht, welcher auf jedem Fall im Container ist. Sobald das Bezugsobjekt gefunden wurde, kann über den Index auf das gesuchte Objekt zugegriffen werden. In der Methode `getNextIndexOf(..)` wird vom Bezugsobjekt aus entsprechend dem Index $n$-mal nach rechts über die Objekte traversiert, um das gesuchte Objekt zurückzugeben. Die Methode `getPreviousIndexOf(..)` traversiert folglich in die entgegengesetzte Richtung. Die Methoden geben den Wert `null` zurück, wenn der Index größer als die Anzahl der Elemente in dem Container ist oder das durch den Index referenzierte Objekt „außerhalb“ des Containers liegt.

Die Zeitaufwand für die Methode `getNextOf(..)` beträgt O(n), wenn das übergebene Bezugsobjekt ungleich dem zuletzt gespeicherten Objekt in der Liste ist. Ansonsten kostet die Ausführung dieser Methode O(index) für die Traversierung nach rechts bis zum Zielobjekt. Die Methode `getPreviousOf(..)` besitzt ebenfalls eine Laufzeit von O(index), wenn das Bezugsobjekt identisch mit dem zuletzt zugegriffenen Objekt der Liste ist. Ansonsten kann die Laufzeit im schlechtesten Fall O(2n) betragen. Dieser Fall tritt ein, wenn das Bezugsobjekt das letzte Element der Liste ist und über den Index auf das erste Element in der Liste zugegriffen werden soll. Man traversiert also zweimal über die gesamte Liste, einmal um das Bezugsobjekt zu finden und einmal in entgegengesetzter Richtung bis zum ersten Element. Im Durchschnitt beträgt die Zugriffszeit also O(n).

Die nachstehende Tabelle 6.3 zeigt die Implementierung des *index*-Multilinks anhand von einigen Beispielen:

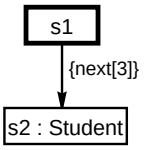
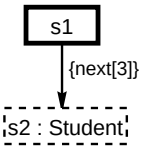
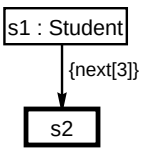
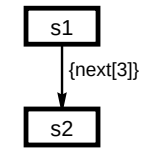
| | |
|---|---|
|  | `// multilink bind s1 : Student`<br>`s2 = this.getNextIndexOfStudents (s1, 3);`<br>`JavaSDM.ensure (s2 != null);` |
|  | `// multilink bind s1 : Student`<br>`s2 = this.getNextIndexOfStudents (s1, 3);`<br>`...` |
|  | `// multilink bind s2 : Student`<br>`s1 = this.getPreviousIndexOfStudents (s2, 3);`<br>`JavaSDM.ensure (s2 != null);` |
|  | `// check multilink s1 to s2`<br>`JavaSDM.ensure (this.getNextIndexOfStudents (s1, 3).equals (s2))`<br>`...` |

Tabelle 6.3: Implementierung des *index*-Multilinks

Die Methoden `getNextIndexOfStudents(..)` und `getPreviousIndexOfStudents(..)` implementieren die Teilgraphensuche für den *index*-Multilink. Die erste Methode wird aufgerufen, wenn die Source-Variable ge-

bunden und die Target-Variable ungebunden ist. Für den umgekehrten Fall wird die zweite Methode aufgerufen. Beiden Methoden werden als Parameter das Bezugsobjekt und der Index `n` übergeben. Der Rückgabewert der Methoden ist `null`, wenn in dem Container kein geeignetes Studenten-Objekt gefunden werden konnte.

## 6.5 Codegenerierung für *indirect*-Multilinks

Der *indirect*-Multilink ist von allen Multilink-Typen am wenigsten restriktiv. Für die spezifizierte Ordnungsbeziehung zwischen zwei Objekten wird nur verlangt, daß das Source-Objekt vor dem Target-Objekt entsprechend der Sortierung in dem Container liegt. Aufgrund der Implementierung des *indirect*-Multilinks, werden die Variablen immer von links nach rechts gebunden. Deshalb kann für die Betrachtung des *indirect*-Multilinks immer davon ausgegangen werden, daß die Source-Variable gebunden ist. Ist die Target-Variable ungebunden, mußt in dem Container von dem Source-Objekt ausgehend, ein geeignetes Objekt gefunden werden. Im Gegensatz zu den übrigen Multilink-Typen ist die Target-Variable nicht eindeutig bestimmt, sobald die Source-Variable gebunden wurde. Jedes Objekt in dem Container, das hinter dem Source-Objekt einsortiert worden ist, kann ein möglicher Kandidat für die Target-Variable sein. Deshalb mußt für die Teilgraphensuche jedes dieser Objekte betrachtet werden, um zu garantieren, daß für eine erfolgreiche Ausführung des Story-Pattern wirklich jede Möglichkeit betrachtet wurde. Sobald aber ein geeigneter Teilgraph in der Laufzeitobjetkstruktur gefunden wurde, brauchen die restlichen Objekte des Containers nicht mehr betrachtet zu werden. In diesem Fall wird die Suche vorzeitig beendet.

Die nachstehende Tabelle 6.4 zeigt die Implementierung des *indirect*-Multilinks anhand von einigen Beispielen. Anhand des ersten Beispiels soll die implementierte Teilgraphensuche für ein *indirect*-Multilink erklärt werden. Das Beispiel enthält eine ungebundene Target-Variable mit zwei Attribut-Constraints. Für die Variable soll ein Objekt der Klasse *Student* in dem Container gefunden werden, dessen Semesterzahl kleiner als zehn und die Fachschaft gleich 17 ist. Hierzu müssen alle Objekte, die hinter dem Source-Objekt einsortiert worden sind, betrachtet werden. Zu diesem Zweck wird die Methode `iteratorOfStudents(..)` des Container-Objektes aufgerufen, die einen Iterator [12] auf dem Container zurückliefert. Mit dem Iterator ist es möglich, die einzelnen Elemente des Containers schrittweise zu betrachten. Damit der Iterator nicht am Anfang des Containers beginnt, wird der Metho-

de die Source-Variable `s1` als Argument übergeben. Der Parameter bestimmt den Startpunkt des Iterators innerhalb des Containers. Die Zugriffsmethoden für Iteratoren wurden für sortierte oder geordnete zu-n Assoziationen entsprechend erweitert. Die *while*-Schleife benutzt nun diesen Iterator, um durch die Elemente des Containers zu iterieren. Die Variable `fujaba_Success` signalisiert die erfolgreiche Ausführung eines Story-Pattern. Sie wird auf `true` gesetzt, wenn die Modifikationen durchgeführt worden sind, also ein Teilgraph in der Laufzeitobjektstruktur gefunden wurde. Die *while*-Schleife kann deshalb vorzeitig beendet werden, wenn diese Variable auf `true` gesetzt worden ist oder der Iterator alle Elemente des Containers durchlaufen hat. Innerhalb der *while*-Schleife wird der Variablen `s2` ein Objekt der Klasse *Student* aus dem Container zugewiesen. Dies geschieht durch den Aufruf der Methode `next()` des Iterators, die das nächste Objekt der Iteration zurückgibt. Nachdem die Variable `s2` gebunden wurde, werden die Attribut-Constraints des Objektes überprüft. Die Attribute des Objektes können über entsprechende Zugriffsmethoden abgefragt werden. Die Überprüfung der Constraints geschieht mit Hilfe der Methode `JavaSDM.ensure(..)`, der jeweils ein Attribut-Constraint als Argument übergeben wird. Sie erzeugt ein Exception, wenn das Attribut-Constraint nicht erfüllt wird, um hiermit ein Bindungsfehler zu signalisieren. Damit die Schleife aber die Möglichkeit hat durch alle Objekte des Containers zu iterieren, mußt in der *while*-Schleife ein neuer *try-catch*-Block angelegt werden, der den gesamten Schleifenrumpf umfaßt. Wird nun eine Exception im weiteren Verlauf der Teilgraphensuche ausgelöst, kann diese Exception innerhalb der *while*-Schleife abgefangen werden. Die Suche beginnt dann wieder am Anfang der Schleife. Ist die Variable `fujaba_Success` immer noch `false` und hat der Iterator noch weitere Elemente, wird die Suche mit einem neuen Objekt an der Variable `s2` fortgesetzt.

Der Code, der für ein *indirect*-Multilink mit einer optionalen Variable erzeugt wird, unterscheidet sich nur minimal von dem ersten Beispiel. Anstelle einer *while*-Schleife wird eine *do-while*-Schleife verwendet. Ist der Container leer, wird die Teilgraphensuche dadurch nicht abgebrochen, sondern kann entsprechend der Semantik einer optionalen Variablen fortgesetzt werden. Dasselbe gilt, wenn kein passendes Element in dem Container für die Variable gefunden werden konnte. Die Schleife wird nicht beendet, sondern fortgesetzt bis die Variable `fujaba_Success` auf `true` gesetzt worden ist. Vor der Überprüfung der Attribut-Constraints mußt die Variable selbst überprüft werden, ob sie gebunden wurde, da eine ungebundene optionale Variable keine Bindungsfehler erzeugt.

In dem dritten Beispiel sind beide Variablen gebunden. In diesem Fall

mußt nur überprüft werden, daß das Source-Objekt vor das Target-Objekt in dem Container einsortiert worden ist. Zu diesem Zweck werden zwei neue Zugriffsmethoden eingeführt:

```
1 : public boolean isBeforeOfCONTAINER (TYP leftObject, TYP rightObject)
2 : {
3 :    if (CONTAINER == null)
4 :    {
5 :       return false;
6 :    }
7 :    else
8 :    {
9 :       return CONTAINER.isBefore (leftObject, rightObject);
10:    }
11: }
12:
13: public boolean isAfterOfCONTAINER (TYP leftObject, TYP rightObject)
14: {
15:    if (CONTAINER == null)
16:    {
17:       return false;
18:    }
19:    else
20:    {
21:       return CONTAINER.isAfter (leftObject, rightObject);
22:    }
23: }
```

Die Methode `isBeforeOfCONTAINER` liefert `true` zurück, wenn das linke Objekt vor dem rechten Objekt einsortiert worden ist. Die zweite Zugriffsmethode gibt den Wert `true` zurück, wenn das linke Objekt nach dem rechten Objekt in dem Container einsortiert worden ist. Die Überprüfung, ob die spezifizierte Ordnungsbeziehung zwischen den zwei Objekten gültig ist, wird durch die neu eingeführten Methoden `isBefore(..)` und `isAfter(..)` der Klasse *FLinkedList* realisiert. Beide Methoden besitzen eine Laufzeit von O(n), da für die Überprüfung die Liste einmal durchlaufen werden mußt.

Im diesem Beispiel wird Überprüfung von der Methode `isBeforeOfStudents (..)` vorgenommen, das eine Methode der Klasse *Uni* ist. Der Methode werden als Parameter die beiden Variablen *s1* und *s2* übergeben. Ist *s1* vor *s2* liefert die Methode den Wert `true` zurück, sonst `false`. Mit Hilfe der Methode `JavaSDM.ensure(..)` wird ein Exception ausgelöst, wenn die Ordnungsbeziehung zwischen den beiden Objekten verletzt wurde.

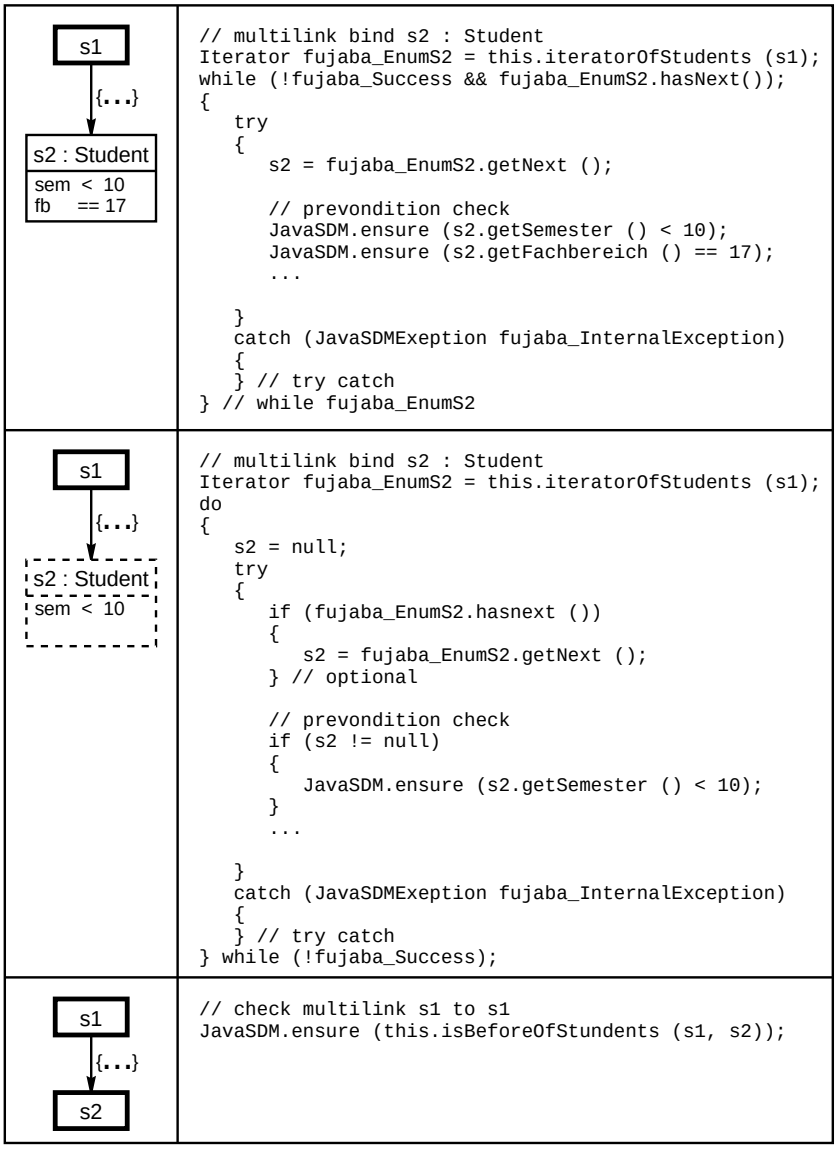



Tabelle 6.4: Implementierung des *indirect*-Multilinks

## 6.6 Codegenerierung für Multilink-Pfade

In den vorherigen Abschnitten wurde die Teilgraphensuche für die verschiedenen Multilink-Typen vorgestellt. Hierbei wurde immer nur die Teilgraphensuche für ein einzelnes Multilink betrachtet. Kombiniert man aber mehrere Multilinks zu einem Multilink-Pfad, um komplexere Ordnungsbeziehungen

zwischen den Objekten darzustellen, muß für die Teilgraphensuche der gesamte Pfad betrachtet werden. Betrachtet man die Multilinks isoliert, unabhängig von den übrigen Multilinks desselben Pfades, besteht die Gefahr, daß man beim Binden der einzelnen Variablen die transitive Eigenschaft des Multilink-Pfades verletzt. Die folgende Abbildung 6.1 veranschaulicht dieses Problem.

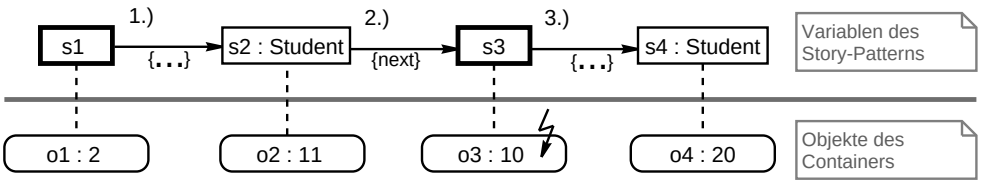



Abbildung 6.1: Beispiel eines fehlerhaft gebundenen Multilink-Pfades

In der oberen Ebene werden die Variablen des Story-Patterns und die dazugehörigen Multilinks dargestellt. Die untere Ebene enthält die Objekte des Containers. Neben den Namen besitzt jedes Objekt noch eine Ordnungszahl, die seine Position innerhalb des Containers angibt. Zu Beginn der Teilgraphensuche sind die Variablen *s1* und *s3* jeweils gebunden. Die Variable *s1* ist mit den Objekt *o2* gebunden worden und die Variable *s3* mit dem Objekt *o3*. Weil die Multilinks unabhängig voneinander betrachtet werden, ist auch ihre Bearbeitungsreihenfolge beliebig. Die Teilgraphensuche beginnt zuerst mit dritten Multilink und bindet das Objekt *o4* an die Variable *s4*. Die Ordnungsrelation zwischen den beiden Objekten wird eingehalten, da das Objekt *o3* mit seiner Ordnungszahl 10 vor das Objekt *o4* liegt, das die Position 20 in dem Container einnimmt. Anschließend wird das erste Multilink ausgeführt. Die Variable *s2* wird mit dem Objekt *o2* gebunden. Die Ordnungsrelation wird ebenfalls eingehalten. Bei der anschließenden Ausführung des zweiten Multilinks sind die Source- und Target-Variable bereits gebunden. Es mußt lediglich überprüft werden, ob das Objekt *o2* vor das Objekt *o3* in dem Container liegt. Da das Objekt *o2* mit der Position 11 hinter dem Objekt *o3* mit der Position 10 liegt, wird die spezifizierte Ordnungsrelation zwischen den beiden Objekten verletzt.

Es ist offensichtlich das dieses Problem behoben werden kann, wenn der Suchbereich für die Variable *s2* eingegrenzt wird. In diesem Fall also sicherstellt, daß *s2* nur an Objekte des Containers gebunden werden kann, die zwischen den Objekten *o1* und *o3* liegen. In den folgenden Abschnitten wird darauf eingegangen, wie man den Suchbereich einer Variable in einem Multilink-Pfad bestimmt.

Um die Teilgraphensuche für die Multilink-Pfade zu vereinfachen, werden die einzelnen Multilinks eines Multilink-Pfades vom *entry*-Link beginnend von links nach rechts abgearbeitet. Fälle, die sich aus der optimierten Teilgraphensuche für Multilinks ergeben, werden hier nicht betrachtet. Es ist nicht ausgeschlossen, daß ein Multilink-Pfad auch optionale oder mengenwertige Variablen besitzt. In Anlehnung an die Bindungsreihenfolge von Variablen bei den normalen Links, werden deshalb normale Variablen vor optionale und mengenwertige Variablen gebunden. Die optionalen und mengenwertigen Variablen werden gleichberechtigt behandelt und entsprechend ihrer Reihenfolge in einem Multilink-Pfad abgearbeitet. Die Teilgraphensuche für ein Multilink-Pfad wird also in zwei Phasen unterteilt. In der ersten Phase werden alle normalen Variablen von links nach rechts gebunden. In der zweiten Phase werden dann alle optionalen und mengenwertigen Variablen von links nach rechts abgearbeitet.

Um die transitive Eigenschaft von Multilink-Pfaden zu erhalten, werden für die ungebundenen Variablen Suchbereiche festgelegt. Ein Suchbereich enthält dabei alle möglichen Elemente eines Containers, die an der Variablen gebunden werden kann, ohne die transitive Eigenschaft des Multilink-Pfades zu verletzen. Ein Suchbereich wird durch eine untere und obere Schranke definiert, die jeweils aus einem Element des Containers gebildet werden. Sie legen den Start- und Endpunkt für die Suche fest. Ein Suchbereich wird für eine Variable immer dann festgelegt, wenn die Variable nicht eindeutig gebunden werden kann. Dies ist zum Beispiel immer bei den *indirect*-Multilinks der Fall, wenn dessen Target-Variable noch ungebunden ist. Für eine Variable, die über einem *next*-Multilink gebunden wird, mußt kein Suchbereich angegeben werden, da das zu bindende Objekt eindeutig durch den *next*-Multilink bestimmt ist.

Eine untere und obere Schranke äußert sich immer in Form einer gebundenen Variable, dessen Objekt die konkrete Schranke im Container bildet. Es spielt dabei keine Rolle, ob eine Variable während der Abarbeitung des Story-Patterns gebunden wurde oder sie in dem Story-Pattern als gebunden spezifiziert wurde. Eine Variable, die in einem Story-Pattern als gebunden spezifiziert wurde, bezeichnet man als *statisch gebunden*. Wird die Variable dagegen während der Abarbeitung des Story-Patterns gebunden, bezeichnet man sie als *dynamisch gebunden*. Existiert keine untere Schranke, beginnt die Suche mit dem ersten Element des Containers. Die Suche endet mit dem letzten Element des Containers, wenn es keine obere Schranke gibt.

Für das Verständnis der Multilink-Pfade ist es nun wichtig zu wissen, welche Variablen in einem Multilink-Pfad als untere und obere Schranke in

Frage kommen. Die Abarbeitungsreihenfolge der Multilinks und der Typ der Variablen bestimmen maßgeblich die untere und obere Schranke eines Suchbereichs. Variablen oder Links mit einem ≪*create*≫-Modifikator werden nicht berücksichtigt, da sie aus der Sicht des Containers noch nicht existieren.

Für eine normale Variable *v1* soll die untere und obere Schranke in einem Multilink-Pfad bestimmt werden. Weil optionale und mengenwertige Variablen erst nach den normalen Variablen gebunden werden, braucht man sie für die Bestimmung der Schranken nicht zu beachten. Eine Ausnahme bilden statisch gebundene mengenwertige Variablen. Statisch gebundene optionale Variablen schließen sich aufgrund ihrer Semantik aus. Für die Variable *v1* kommen als untere Schranke also nur normale Variablen und statisch gebundene Variablen in Frage. Weil ein Multilink-Pfad von links nach rechts gebunden wird, sind alle normalen Variablen auf der linken Seite von *v1* bereits gebunden. Andernfalls wäre das Story-Pattern schon längst abgebrochen worden. Aus der Sicht der Variable *v1*, ergibt sich die untere Schranke also aus der ersten normalen oder statisch gebundenen Variable, die *v1* auf ihrer linken Seite trifft. Für die obere Schranke kommen nur statisch gebundene Variable in Betracht, da die normalen Variablen zu diesem Zeitpunkt noch ungebunden sind. Aus der der Sicht der Variablen *v1*, wird die obere Schranke aus der ersten statisch gebundene Variable gebildet, die es auf der rechten Seite trifft[1]. Die folgende Abbildung 6.2 veranschaulicht die oben geschilderte Situation durch ein Beispiel.

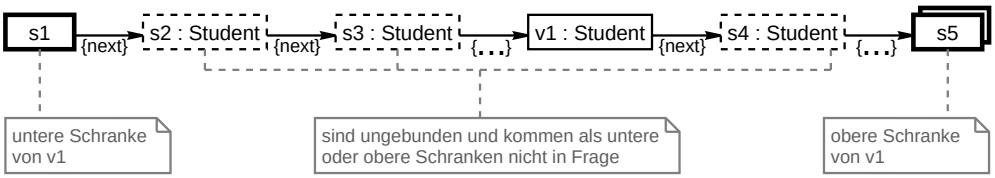



Abbildung 6.2: Untere und obere Schranke für die normale Variable *v1*

Wird eine optionale oder mengenwertige Variable *v1* gebunden, sind sowohl die normalen als auch die optionalen und mengenwertigen Variablen für die Bestimmung der unteren und oberen Schranken von Bedeutung. Alle normalen Variablen sind zu diesem Zeitpunkt bereits gebunden worden, deshalb stellen sie alle als potentielle Schranken dar. Ansonsten hätte ein

[1]Diese Aussage ist richtig, solange man die optimierte Teilgraphensuche für Multilinks nicht betrachtet. Es ist durchaus möglich, daß sich auf der rechten Seite der Variable *v1* auch dynamisch gebundene normale Variablen befinden. Für das Verständnis der Multilink-Pfade spielt es aber keine Rolle, wenn diese Fälle hier nicht betrachtet werden.

Bindungsfehler längst zum Abbruch des Story-Patterns geführt. Weil auch die optionalen und mengenwertigen Variablen von links nach rechts gebunden werden, können die optionalen und mengenwertigen Variablen auf der linken Seite von *v1* gebunden sein. Deshalb müssen sie ebenfalls betrachtet werden. Für die untere Schranke kommen also alle Variablen auf der linken Seite von *v1* in Frage. Da optionale und mengenwertige Variablen keine Bindungsfehler erzeugen, mußt jeweils überprüft werden, ob die Variable erfolgreich gebunden wurde oder nicht. Ist die Variable ungebunden, braucht sie nicht berücksichtigt zu werden. Andernfalls mußt die Variable weiterhin als potentielle Schranke betrachtet werde. Die erste statisch oder dynamisch gebundene normale, optionale oder mengenwertige Variable links von der Variablen *v1* bildet deshalb die unter Schranke für den Suchbereich. Für die obere Schranke müssen die optionalen oder mengenwertigen Variablen nicht betrachtet werden, da sie zu diesem Zeitpunkt noch ungebunden sind. Eine Ausnahme bilden statisch gebundene mengenwertige Variablen. Die erste statisch gebundene oder normale Variable auf der rechten Seite von *v1* bildet deshalb die obere Schranke des Suchbereichs. Das Beispiel in der unteren Abbildung 6.3 zeigt die potentiellen unteren und oberen Schranken für eine optionale Variable an.

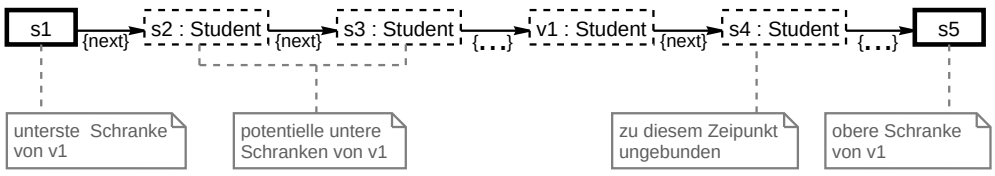



Abbildung 6.3: Untere und obere Schranke für die optionale Variable *v1*

Die normalen Variablen *s1* und *s5* sind zum betrachteten Zeitpunkt bereits gebunden worden . Weil die Variable *v1* als nächstes gebunden werden soll, kann davon ausgegangen werden, daß die optionale Variable *s4* noch ungebunden ist. Deshalb bildet die Variable *s5* die obere Schranke für den Suchbereich von *v1*. Die optionalen Variablen *s2* und *s3* sind potentielle untere Schranken. Ist die Variable *s3* gebunden, bildet sie die untere Schranke zu *v1*. Ansonsten mußt überprüft werden, ob nicht die Variable *s2* gebunden ist. In diesem Fall würde sie die untere Schranke bilden. Sind beide optionale Variablen ungebunden, kommt nur noch die Variable *s1* in Betracht. Sie ist die unterste Schranke von *v1*.

Anhand des folgenden Beispiels in der Abbildung 6.4 wird die implementierte Teilgraphensuche ein für ein Multilink-Pfad vorgestellt:

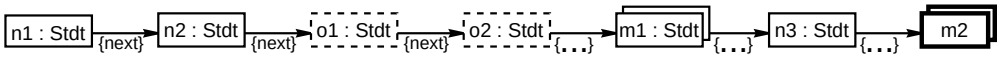



Abbildung 6.4: Beispiel eines Multilink-Pfades

Das Beispiel wurde bewußt so konstruiert, daß die optimierte Teilgraphensuche für Multilinks nicht zum Zuge kommt. Die Teilgraphensuche beginnt mit der Source-Variable des *entry*-Links. Wäre die Variable *n1* optional, hätte man mit der normalen Variable *n2* begonnen. In der ersten Phase werden werden nur die normalen Variablen gebunden. Die Variable *n1* ist noch ungebunden und mußt deshalb noch gebunden werden. Weil das Objekt, das an der Variable *n1* gebunden werden soll, nicht eindeutig bestimmt ist, wird ein Suchbereich für die Variable ermittelt. Für *n1* gibt es offensichtlich keine untere Schranke, deshalb beginnt die Suche mit dem erstem Element des Containers. Für die obere Schranke kommen nur statisch gebundene Variablen in Frage, deshalb bildet die statisch gebundene Variable *m2* die obere Schranke des Suchbereichs. Eine Schranke wird durch ein Objekt des Containers repräsentiert, die Mengenvariable *m2* enthält aber eine Menge von Objekten. Für die obere Schranke wird deshalb das erste Element der Menge genommen und nicht die Variable *m2* selbst. Die Abbildung 6.5 zeigt die derzeitige Situation für die Variable *n1*. Analog würde man für eine untere Schranke das letzte Element einer Mengenvariable nehmen. Die Sortierung der Mengenvariable mußt mit der Sortierung des Containers übereinstimmen, damit das erste Element der Mengenvariable als obere Schranke genommen werden kann. Erfüllt die statisch gebundene Mengenvariable diese Voraussetzung nicht, kann die transitive Eigenschaft der Multilink-Pfade nicht garantiert werden. Die Mengenvariable mußt also vor der Verwendung mit einem Multilink durch eine entsprechende Funktion sortiert worden sein. Diese Funktion mußt vom Anwender implementiert und vorher aufgerufen werden. Mengenvariablen die in einem Multilink-Pfad dynamisch gebunden wurden, halten die Sortierung des Containers automatisch ein. Die normalen sowie die optionalen und mengenwertigen Variablen rechts von *n1* sind zu diesem Zeitpunkt noch ungebunden und müssen deshalb nicht berücksichtigt werden. Um die Variable *n1* zu binden würde Fujaba den folgenden Java-Code generieren:

```
1 : // multilink bind n1 : Students
2 : Iterator fujaba_IterN1 = this.iterOfStudents ();
3 : boolean fujaba_UpperBound_N1 = false;
4 : while (!fujaba_Success && fujaba_IterN1.hasNext () &&
           !fujaba_UpperBound_N1)
```

```
5 : {
6 :    try
7 :    {
8 :       n1 = (Student) fujaba_IterN1.next ();
9 :
10:       if (n1.equals(m2.first ()));
11:       {
12:          fujaba_UpperBound_N1 = true;
13:          throw new JavaSDMException ();
14:       }
15:       ...
16:    }
17:    catch (JavaSDMException fujaba_InternalException)
18:    {
19:    }  // try catch
20: }  // while fujaba_IterN1
```

Gewöhnlich wird die untere Schranke eines Suchbereichs der Methode `iterOfStudents (..)` als Parameter übergeben. Damit wird der zurückgelieferte Iterator der Methode initialisiert und veranlaßt mit diesem Objekt bei der Iteration zu beginnen. Weil für die Variable *n1* keine untere Schranke existiert, wird die überladene Methode `iterOfStudents ()` aufgerufen. Diese Methode liefert einen Iterator zurück, der grundsätzlich mit dem ersten Element des Containers beginnt. Die Einhaltung der oberen Schranke wird innerhalb der Schleife überprüft. Bei jeder Iteration wird überprüft, ob nicht die obere Schranke erreicht wurde. Falls das zurückgelieferte Objekt des Iterators identisch ist dem Objekt der oberen Schranke, wird eine Exception ausgelöst und die Variable `fujaba_UpperBound_N1` auf `true` gesetzt. Die Exception wird innerhalb der Schleife abgefangen. Dadurch wird der reguläre Ablauf der *while*-Schleife unterbrochen und die Ausführung beginnt wieder am Anfang der Schleife. Weil die Variable `fujaba_UpperBound_N1` zuvor auf `true` gesetzt worden ist, wird die Schleife damit abgebrochen. Das Story-Pattern wird damit beendet oder die Teilgraphensuche beginnt weiter oben mit einem neuen Pfad.

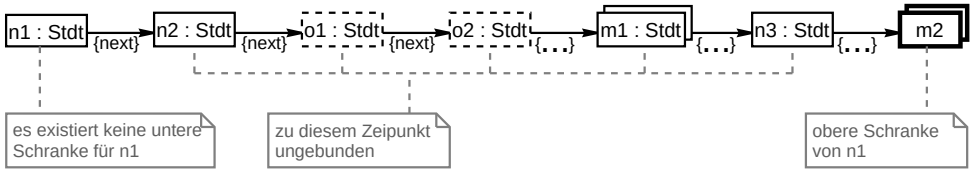



Abbildung 6.5: Implementierung der Teilgraphensuche für die Variable *n1*

Nachdem die Variable *n1* erfolgreich gebunden wurde, wird als nächstes

versucht die Variable *n2* zu binden. Weil das zu bindende Objekt an die Variable *n2* durch den *direct*-Multilink und die gebundene Source-Variable eindeutig bestimmt ist, mußt für *n2* kein Suchbereich ermittelt werden. An *n2* wird das Nachfolgerobjekt des Source-Objektes gebunden. Die optionalen Variablen *o1* und *o2* und die Mengenvariable *m1* werden erst in der zweiten Phase gebunden und brauchen zu diesem Zeitpunkt deshalb noch nicht berücksichtigt zu werden.

Die Teilgraphensuche wird deshalb mit der Variablen *n3* fortgesetzt. Aufgrund des *indirect*-Multilinks, das auf die Variable verweist, mußt für *n3* ein Suchbereich ermittelt werden. Als untere Schranke wird die Variable *n2* bestimmt, die zuvor dynamisch gebunden wurde. Der obere Suchbereich wird durch die gebundene Mengenvariable *m2* eingegrenzt, beziehungsweise das erste Element der Menge. Die folgende Abbildung 6.6 veranschaulicht die derzeitige Situation für die Variable *n3*. Auf einem Ausdruck des Quellcodes wurde verzichtet, da sich der Code bis auf die Variable für die untere Schranke im Vergleich zu dem vorherigen Code nur unwesentlich geändert hat.

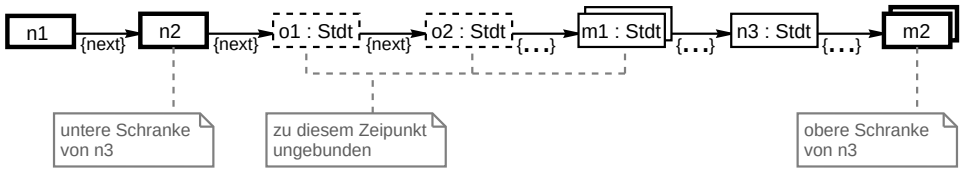



Abbildung 6.6: Aktueller Stand der Teilgraphensuche für die Variable *n3*

In dem Multilink-Pfad sind jetzt alle normalen Variablen gebunden. In der zweiten Phase können jetzt deshalb die restlichen optionalen und mengenwertigen Variablen gebunden werden. Die Teilgraphensuche beginnt wieder mit dem *entry*-Link. Das *entry*-Link enthält keine optionalen oder mengenwertige Variablen, deshalb wird die Suche mit dem nächsten Multilink fortgesetzt. Der *direct*-Multilink verweist auf die optionale Variable *o1*. Für die *o1* mußt kein Suchbereich ermittelt werden, deshalb kann das Multilink direkt ausgeführt werden. Unabhängig davon, ob die Variable erfolgreich gebunden werden konnte oder nicht, kann die Suche mit dem nächsten Multilink fortgesetzt werden. Als nächstes wird also versucht die optionale Variable *o2* zu binden. Auch hier mußt kein Suchbereich für die Variable bestimmt werden. Das besondere an diesem *direct*-Multilink ist, daß sowohl die Source-Variable optional ist. An der eigentlichen Semantik des *direct*-Multilinks wird sich dadurch nichts ändern. Der Unterschied ist, daß die Ausführung des Multilinks

nun davon abhängt, ob die optionale Source-Variable erfolgreich gebunden wurde oder nicht. Ist die Source-Variable gebunden, wird das Multilink ausgeführt und es wird versucht ein Nachfolger an die Variable *o2* zu binden. Ansonsten unterbleibt die Ausführung des Multilinks und die Teilgraphensuche wird fortgesetzt. Ein Bindungsfehler wird in keinem der beiden Fälle erzeugt. Die Implementierung dieses Multilinks sieht wie folgt aus:

```
1 : // multilink bind o2 : Student
2 : if (o1 != null)
3 : {
4 :    o2 = (Student) this.getNextOfStudents (o1);
5 : }
```

Die Ausführung des Multilinks ist nun davon abhängig, ob die Variable `o1` zuvor erfolgreich gebunden wurde oder nicht. Ist die Variable gebunden, wird die Methode getNextOfStudents(..) mit dieser Variablen als Argument ausgeführt. Die Methode gibt den Nachfolger zu `o1` zurück, fall er existiert. Weil die Variable `o2` ebenfalls optional ist, entfällt die anschließende Überprüfung, ob `o2` erfolgreich gebunden wurde. Die Methode getNextOfStudents(..) macht deutlich, wieso die Ausführung eines *direct*-Multilinks abhängig von der Existenz einer gebundenen Source-Variable ist. Ohne ein Argument kann diese Methode nicht ausgeführt werden. Um den direkten Nachfolger einer Variablen zu bestimmen, benötigt man dessen Vorgänger.

Diese Einschränkung gilt nicht für die *indirect*-Multilinks. Ersetzt man in diesem Beispiel den *direct*-Multilink zwischen den beiden optionalen Variablen durch einen *indirect*-Multilink, würde man für die Variable *o2* einen Suchbereich ermitteln und anhand diesen versuchen zu binden. Der *indirect*-Multilink wird also in jedem Fall ausgeführt, unabhängig davon, ob die optionale Source-Variable erfolgreich gebunden wurde oder nicht.

Die Teilgraphensuche wird mit der Mengenvariable *m1* fortgesetzt. An ihr werden alle Elemente eines Suchbereichs gebunden, die nicht die isomorphie Eigenschaft des Graphen verletzen. Die obere Schranke des Suchbereichs wird durch die Variable *n3* gebildet. Für die untere Schranke kommen die Variablen *o1*, *o2* und *n2* in Frage. Die unterste Schranke des Suchbereichs wird von der normalen Variablen *n2* gebildet. Diese Schranke wird genommen, wenn die optionalen Variablen *o1* und *o2* nicht gebunden werden konnten. Ansonsten wird die erste gebundene optionale Variable als untere Schranke festgelegt. Die Abbildung 6.7 zeigt eine Übersicht der geschilderten Situation.

Die Implementierung der Teilgraphensuche für die Mengenvariable *m1* wird in drei Schritten unterteilt. Zuerst wird die untere Schranke ermittelt,

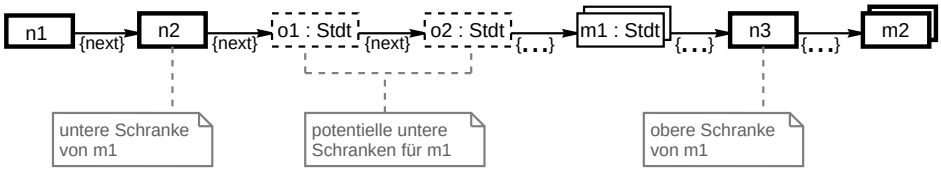



Abbildung 6.7: Aktueller Stand der Teilgraphensuche für die Variable *m1*

anschließend wird der Iterator initialisiert und zuletzt wird die Mengenvariable gebunden. Die untere Schranke kann nur während der Ausführungszeit des Story-Patterns bestimmt werden. Da nur zur Laufzeit überprüft werden kann, ob die optionalen Variablen gebunden wurden oder nicht. Für die Ermittlung der unteren Schranke von *m1* wird folgender Quellcode erzeugt:

```
1 : // find lower bound for m1
2 : Student fujaba_LowerBoundForM1 = null;
3 : if (o2 != null)
4 : {
5 :    fujaba_LowerBoundForM1 = o2;
6 : }
7 : else if (o1 != null)
8 : {
9 :    fujaba_LowerBoundForM1 = o1;
10: }
11: else
12: {
13:    fujaba_LowerBoundForM1 = n2;
14: }
```

In der Variablen `fujaba_LowerBoundForM1` wird die untere Schranke abgelegt. Am Anfang wird sie mit `null` initialisiert. Die geschachtelten *if*-Anweisungen ermitteln die untere Schranke, in dem sie jeweils abfragen, ob die Variable ungleich `null` sind. Die Variablen werden dabei von rechts nach links von *m1* beginnend abgefragt. Ist einer der optionalen Variablen ungleich `null`, wird die Variable `fujaba_LowerBoundForM1` damit aktualisiert. Ansonsten wird ihr *n2* zugewiesen.

Nachdem die untere Schranke ermittelt worden ist, kann der Iterator initialisiert werden. Hierzu wird der Methode `iteratorOfStudents(..)` als Argument die Variable `fujaba_LowerBoundForM1` übergeben. Der zurückgelieferte Iterator dieser Methode beginnt dann mit der unteren Schranke die Elemente des Containers aufzuzählen. In diesem Beispiel existiert mit der normalen Variablen *n2* immer eine untere Schranke, deshalb kann die

Variable `fujaba_LowerBoundForM1` auch niemals den Wert `null` annehmen. Würde man in diesem Beispiel alle normalen Variablen links von *m1* entfernen, kann es durchaus passieren, daß man keine untere Schranke für *m1* findet. Diese Situation tritt ein, wenn für die beiden optionalen Variablen links von *m1* keine passenden Objekte in dem Container gefunden werden konnten. Die Methode `iteratorOfStudents(..)` liefert deshalb einen Iterator zurück, der am Anfang des Containers beginnt, wenn das übergebene Argument `null` ist. Im folgenden wird der Rest des Quellcodes für *m1* abgedruckt:

```
1 : Iterator fujaba_IterM1 = this.iteratorOfStudents (lowerBoundForM1);
2 : while (fujaba_IterM1.hasNext ())
3 : {
4 :    try
5 :    {
6 :       fujaba_TmpObject = fujaba_IterM1.next();
7 :       if(fujaba_TmpObject.equals(n3))
8 :       {
9 :          break;
10:       }  // fi
11:
12:        m1.add (fujaba_TmpObject);
13:    }
14:    catch (JavaSDMException fujaba_InternalException)
15:    {
16:    } // try catch
17: } // while
```

In den Zeilen 2 bis 20 wird die Mengenvariable *m1* gebunden. Die *while*-Schleife iteriert durch alle Elemente des Suchbereichs und fügt sie der Mengenvariable hinzu. Die Schleife wird abgebrochen, wenn das vom Iterator zurückgelieferte Objekt identisch ist mit dem Objekt der oberen Schranke. Nach der *while*-Schleife werden alle Objekte aus der Menge entfernt, die die Isomorphie Eigenschaft des Graphen verletzen. Dieser Code wurde hier nicht mehr mit abgedruckt.

## 6.7 Codegenerierung für das Einfügen von Objekten mit Multilinks

Multilinks erlauben die Einfügeposition beim Hinzufügen von Objekten in dem Container zu spezifizieren. Dazu müssen die Klassen, die sor-

tierte oder geordnete zu-n Assoziationen besitzen, durch entsprechende Zugriffsmethoden erweitert werden. Im folgenden wird die Methode `addBeforeOfCONTAINER(..)` abgedruckt:

```
1 : public boolean addBeforeOfCONTAINER (TYPE refObject, TYPE newObject)
2 : {
2 :    boolean changed;
3 :    if (CONTAINER == null)
4 :    {
5 :       changed = false;
6 :    }
7 :    else
8 :    {
9 :       int index = CONTAINER.indexOf (refObject);
10:       try
11:       {
12:          CONTAINER.add (index, newobject);
13:          changed = true;
14:       }
15:       catch (IndexOutOfBoundsException e)
16:       {
17:          changed = false;
18:       }
19:    }
20:    return changed;
21: }
```

Die Methode besitzt zwei Parameter, `refObject` und `newObject`. Mit `refObject` wird ein Bezugsobjekt angegeben vor dem das neue Objekt `newObject` eingefügt werden soll. Die Methode liefert `true` zurück, wenn die Einfügeoperation erfolgreich ausgeführt werden konnte. Um das neue Objekt in die Liste einzufügen, wird zuerst der Index des Bezugsobjektes in der Liste ermittelt. Mit der `add(..)`-Methode der Liste wird nun das neue Objekt eingefügt, wobei die Stelle an der das Objekt eingefügt werden soll, durch den Index des Bezugsobjektes bestimmt wird. Ist die Anzahl der Elemente in der Liste gleich n, so beträgt der Zeitaufwand für die Methode O(n).

Die zweite Zugriffsmethode `addAfterOfCONTAINER(..)` fügt ein Objekt hinter dem übergebenen Bezugsobjekt ein. Die Methode unterscheidet sich von der `addBeforeOf(..)` Methode nur durch die Zeile 22 und wird deshalb hier aus Platzgründen nicht mehr abgedruckt. Die Variable `index` wird vor der Übergabe einmal inkrementiert. Dadurch wird das Objekt hinter dem Bezugsobjekt eingefügt. Die Laufzeit der Methode ist ebenfalls O(n). Die nachfolgende Tabelle 6.5 zeigt zwei Beispiele und den zugehörigen Java-Code:

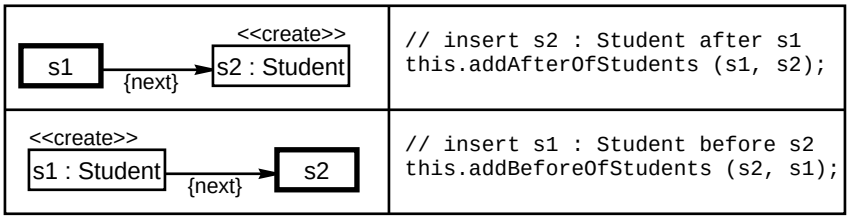



Tabelle 6.5: Codeerzeugung für die Einfügeoperation eines Multilinks

## 6.8 Kostenbewertung von Multilinks

In diesem Abschnitt werden die Optimierungsstrategien für die Teilgraphensuche von Multilinks vorgestellt. Durch eine günstige Abarbeitungsreihenfolge der Multilinks soll der Suchaufwand verringert werden. Diese Reihenfolge ergibt sich aus dem Force-Failure-Prinzip, bei dem es gilt, möglichst früh Bindungsfehler zu erzwingen. Dadurch können ungünstige Suchpfade schneller erkannt und abgebrochen werden. Multilinks mit restriktiven Bedingungen oder einem kleinen Suchbereich werden also als erstes abgearbeitet. Die Bewertung der Multilinks erfolgt durch Prioritäten, die seinen Suchaufwand wiederspiegeln. Multilinks mit einem geringen Suchaufwand (Kosten) besitzen eine hohe Priorität und werden deshalb bei der Ermittlung der Abarbeitungsreihenfolge begünstigt. Die Multilinks werden zu diesem Zweck in Prioritätsklassen zusammengefaßt. Innerhalb einer Klasse besitzen die Multilinks dieselbe Priorität. Es werden stets alle Multilinks einer Prioritätsklasse abgearbeitet, bevor mit der nächsten niedrigeren Prioritätsklasse die Teilgraphensuche fortgesetzt wird.

Der *Check-Multilink* ist von allen Multilinks der kostengünstigste Multilink und wird deshalb als erstes ausgeführt. Ein Check-Multilink ist ein Multilink, bei dem die Source- und Target-Variable gebunden ist. Eine solche Situation kann auftreten, wenn die beiden Variablen über einem günstigeren Weg gebunden worden sind und der Multilink zu diesem Zeitpunkt noch nicht abgearbeitet wurde. Bei der Teilgraphensuche muß nun überprüft werden, ob die durch den Multilink spezifizierte Ordnungsbeziehung zwischen den beiden Variablen eingehalten wird. Für die Überprüfung wird eine entsprechende Methode aufgerufen, die je nach dem Multilink-Typ variieren kann. Eine Suche in der Laufzeitobjektstruktur entfällt, deshalb sind die Kosten für ein Check-Multilink von allen Multilinks am geringsten. Die Abbildung 6.8 zeigt die möglichen Ausprägungen von Check-Multilinks und deren zugehörigen Quellcode. Zur Vereinfachung der Darstellung wurde die Container-Variable

weggelassen und der Multilink zwischen den Variablen gezeichnet. Alle Variablen in der Abbildung sind gebunden. Auf eine dickere Umrandung wurde aber verzichtet, um den Typ der Variablen besser darstellen zu können. Wie man anhand des vierten Beispiels sehen kann, enfällt eine Überprüfung sobald einer der beteiligten optionalen oder mengenwertigen Variablen ungebunden ist.

Zu derselben Prioritätsklasse gehören auch die *Check-first-* und *Check-last*-Multilinks. Alle gebundenen normalen Variablen, die durch einen *first-* oder *last*-Multilink markiert worden sind, gehören dieser Klasse an (siehe Abbildung 6.8). Die Teilgraphensuche muß nun überprüfen, ob das an die Variable gebundene Objekt identisch ist mit dem ersten beziehungsweise letzten Element des Containers. Unterscheiden sich die beiden Objekte wird ein Bindungsfehler erzeugt.

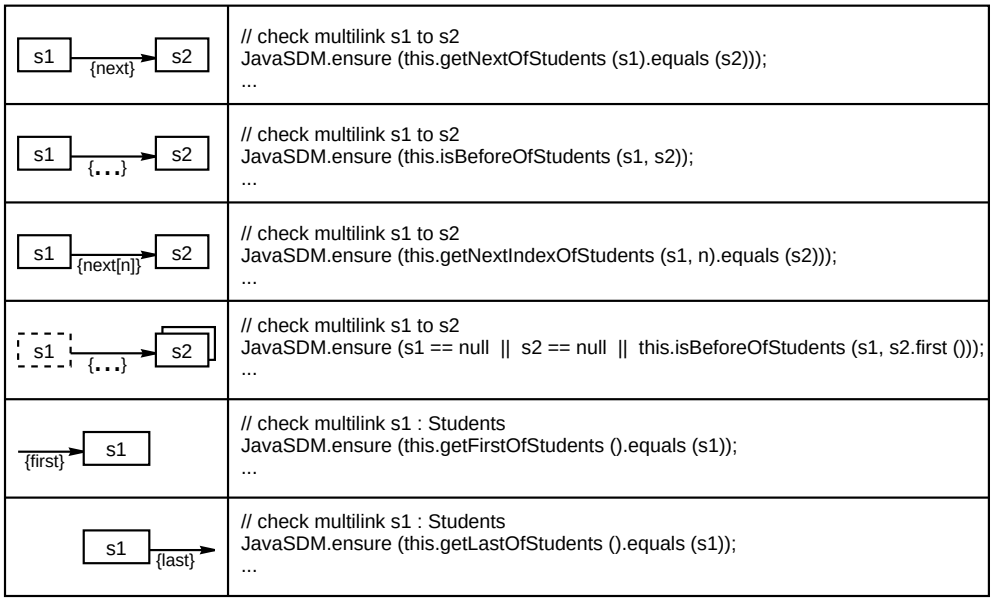



Abbildung 6.8: Check-Multilinks

Die nächste Prioritätsklasse umfaßt die *first-* und *last*-Multilinks. Variablen die durch einen *first-* oder *last*-Multilink ausgezeichnet worden sind, können eindeutig gebunden werden. Der Suchraum enthält jeweils genau ein Element. Bei dem *first*-Multilink ist es das erste Element des Containers und bei dem *last*-Multilink entsprechend das letzte Element. Auf beide Elemente kann man über entsprechende Methoden des Containers direkt zugreifen. Weil in der konkreten Implementierung des Containers eine Liste verwendet

wurde, ist der Zeitaufwand hierfür O(1). Ein Bindungsfehler wird erzeugt, wenn das erste beziehungsweise letzte Element eines Containers nicht an einer Variable gebunden werden kann. Dieser Fall kann nur eintreten, wenn der Container leer ist. Weil die Wahrscheinlichkeit sehr gering ist, daß ein Container einer zu-n Assoziation keine Elemente enthält, ist auch die Wahrscheinlichkeit für einen Bindungsfehler sehr gering. Nach dem Force-Failure-Prinzip müßte man diese Prioritätsklasse eigentlich später behandeln. Der Grund weshalb diese Prioritätsklasse trotzdem an zweiter Stelle behandelt wird, ist die Absicht Variablen schon möglichst früh eindeutig zu binden und das mit einem geringen Suchaufwand. Diese Voraussetzungen werden von den *first*- und *last*-Multilinks erfüllt. Durch diese Vorgehensweise sollen Bindungsfehler möglichst früh „provoziert“ werden. Dies geschieht im Zusammenhang mit den Multilinks der nun folgenden dritten Prioritätsklasse.

Die dritte Prioritätsklasse umfaßt alle Multilinks, die den Typ *next* oder *index* haben und bei denen entweder die Source- oder die Target-Variable gebunden ist. Zusätzlich gilt, daß die beiden Variablen den Typ *normal* haben (siehe Abbildung 6.9). Das gesuchte Objekt ist durch den Multilink-Typ *next* oder *index* und der gebundenen Variable eindeutig bestimmt. Der Suchraum enthält also genau ein Element. Über die entsprechenden Zugriffsmethoden kann direkt auf das Objekt zugegriffen werden. Erfüllt das Objekt die an der Variablen spezifizierten Bedingungen nicht oder es ist bereits gebunden worden, kann sofort ein Bindungsfehler ausgegeben werden. Eine Fortsetzung der Suche ist nicht nötig, da der Suchraum nur ein Element enthält. Weil Fujaba für den Container eine Liste verwendet, ist der Zeitaufwand im schlechtesten Fall O(n). Um die Zugriffszeiten zu verbessern, speichert die Liste jeweils das letzte Element auf das zugegriffen wurde. Wird die Zugriffsmethode jetzt mehrmals hintereinander aufgerufen, ist der benötigte Zeitaufwand nur noch O(1). Dies ist genau dann der Fall, wenn mehrere *direct*- oder *index*-Multilinks einen zusammenhängenden Pfad bilden, der nur aus normalen Variablen besteht. So ein „fester“ Pfad ist eindeutig bestimmbar, sobald nur eine Variable gebunden wurde. Aufgrund dieser Eigenschaft eines festen Pfades, kann man über einem festen Pfad auch rückwärts traversieren, um die Variablen zu binden, ohne die Transitivität zu verletzen. Mit dieser Prioritätsklasse kann so ein Pfad sukzessiv vollständig gebunden werden, sofern auch nur eine gebundene Variable als Einstiegspunkt existiert. Das ist der Grund, warum diese Klasse an dritter Stelle behandelt wird. Bindungsfehler können in einem festen Pfad sehr schnell erkannt werden.

Die nächsten Prioritätsklassen behandeln die restlichen Multilinks, die bisher nicht eindeutig gebunden werden konnten. Diese Prioritätsklassen


Abbildung 6.9: Multilinks der dritten Prioritätsklasse

realisieren die in dem Unterkapitel 6.6 »Teilgraphensuche für Multilink-Pfade« besprochene Teilgraphensuche für ein Multilink-Pfad. Die erste Phase bindet deshalb von links nach rechts die bisher noch ungebundenen normalen Variablen des Multilink-Pfades. Die anschließende zweite Phase bindet dann die restlichen optionalen und mengenwertigen Variablen, die noch ungebunden sind.

Optionale und mengenwertige Variablen werden bei den Multilinks gleichberechtigt behandelt. Die mengenwertigen Variablen sind den optionalen Variablen nicht untergeordnet, wie bei den Links. Aufgrund des Force-Failure-Prinzips werden sie aber erst nach den normalen Variablen gebunden, weil sie aufgrund ihrer Semantik keine Bindungsfehler erzeugen. Dies ist auch der Grund, weshalb die Teilgraphensuche für einen Multilink-Pfad in zwei Phasen eingeteilt werden mußte.

In den zwei Phasen werden die Multilinks jeweils mit dem *entry*-Link beginnend von links nach rechts abgearbeitet. Der Grund für diese Vorgehensweise ist, daß für das Matchen einer Variablen ein Iterator benutzt wird, um die Elemente des Containers zu durchlaufen. Der verwendete Iterator der Java Bibliothek durchläuft den Container dabei grundsätzlich von links nach rechts. Würde man die Multilinks in einer beliebigen Reihenfolge abarbeiten, kann es im ungünstigsten Fall passieren, daß der letzte Multilink eines Multilink-Pfades zuerst bearbeitet wird. Beim Matchen des Multilinks werden die Variablen mit den ersten noch ungebundenen Objekten des Containers gebunden. Versucht man jetzt die übrigen Multilinks des Multilink-Pfades zu matchen, so erzeugen sie nur Bindungsfehler, da die gebundenen Objekte die Transitivität verletzen. Der ungünstigste Fall tritt also ein, wenn der Multilink-Pfad komplett von rechts nach links gebunden wird. Besteht ein Multilink-Pfad aus $n$ *indirect*-Multilinks und damit $n+1$ ungebundenen Variablen, hätte man bei einem Container mit $m$ Elementen im schlechtesten Fall eine Laufzeit von $O(m^n)$. Wird ein Multilink-Pfad dagegen von links nach rechts abgearbeitet, kann so eine Situation vermieden werden.

Die vierte und fünfte Prioritätsklasse realisieren zusammen die erste

Phase der Teilgraphensuche für Multilink-Pfade. Die vierte Prioritätsklasse enthält die *entry*-Links der Multilink-Pfaden. Jeder Multilink-Pfad besitzt dabei genau einen *entry*-Link. Der *entry*-Link wird benutzt, um als Startpunkt eines Multilink-Pfades alle Multilinks von links nach rechts zu durchlaufen. Nachdem ein *entry*-Link abgearbeitet wurde, wird die Variable *bindOptionalAndSet* des *entry*-Links `true` auf gesetzt. Ein damit gekennzeichneter *entry*-Link kann jetzt nun in der zweiten Phase bearbeitet werden. Nachdem das *entry*-Link abgearbeitet wurde, wird jeder Multilink des Multilink-Pfades von links nach rechts abgearbeitet. Jede noch ungebundene normale Variable wird auf diese Weise gebunden. Diese Multilinks werden in der fünften Prioritätsklasse zusammengefaßt. Wichtig ist, daß die Multilinks entsprechend ihrem vorkommen in dem Pfad in diese Prioritätsklasse hinzugefügt werden. Multilinks mit derselben Priorität werden nach *FIFO*-Prinzip abgearbeitet.

Dadurch, daß die einzelnen Multilinks eine eigene Priorität bekommen, ist es nicht notwendig, daß die Multilinks durchgehend von links nach rechts abgearbeitet werden müssen. Ergeben sich beim Binden der Multilinks günstigere Wege, können diese zuerst verfolgt werden. Die Teilgraphensuche für den Multilink-Pfad wird dann später fortgesetzt.

Die restlichen zwei Prioritätsklassen realisieren die zweite Phase der Teilgraphensuche. In dieser Phase werden die noch ungebundenen optionalen und mengenwertigen Variablen gebunden. Die Arbeitsweise ist identisch mit der ersten Phase. Die sechste Prioritätsklasse enthält alle *entry*-Links bei denen die Variable *bindOptionalAndSet* auf true gesetzt wurde. Mit der siebten Prioritätsklasse wird die Traversierung der Multilinks realisiert. Die Tabelle 6.6 zeigt die zusammengefaßten Prioritätsklassen in absteigender Reihenfolge an.

Um die Teilgraphensuche in einem Story-Pattern zu vervollständigen, müssen diese Prioritätsklassen noch in die Prioritätenskala (siehe Kapitel »Einführung in die Teilgraphensuche«) eingeordnet werden. Multilinks werden nur bei sortierten zu-n Assoziationen angewendet, deshalb sind sie auf jedem Fall nach den zu-1 Links einzugliedern. Aufgrund der restriktiveren Bedingungen der Multilinks und der Effizienz werden sie vor den zu-n Links ausgeführt. Dies entspricht dem Force-Failure-Prinzip. Würde man zuerst die zu-n Links und erst anschließend die Multilinks ausführen, hätte man im schlechtesten Fall eine exponentielle Laufzeit. Statt die Multilinks für eine Optimierung der Laufzeit zu nutzen, werden die Variablen unabhängig voneinander gebunden und erst anschließend würde man die Einhaltung der spezifizierten Ordnungsbeziehung überprüfen. Die Prioritätsklassen sechs und

| Klasse | Name der Prioritäten | Beschreibung |
|---|---|---|
| 1 | P-Multilink-Check | Multilinks, deren Source- und Target-Variablen gebunden sind - bei first- und last-Multilinks entsprechend nur eine Variable |
| 2 | P-Multilink-First | first-Multilinks |
| | P-Multilink-Last | last-Multilinks |
| 3 | P-Multilink-Bound-to-Unbound | direct-Multilinks mit normalen Variablen, wobei die Source-Variable gebunden ist |
| | P-Multilink-Unbound-to-Bound | direct-Multilinks mit normalen Variablen, wobei die Target-Variable gebunden ist |
| 4 | P-Multilink-Entry | entry-Links |
| 5 | P-Multilink-Path | Multilinks eines Multilink-Pfades in der ersten Phase |
| 6 | P-Multilink-Entry-Optional | entry-Links, deren Variablen bindOptional-AndSet auf true gesetzt worden sind |
| 7 | P-Multilink-Path-Optional | Multilinks eines Multilink-Pfades in der zweiten Phase |

Tabelle 6.6: Multilink Prioritäten in absteigender Reihenfolge

sieben binden die optionalen und mengenwertigen Variablen eines Multilink-Pfades. Sie werden deshalb entsprechend dem Force-Failure-Prinzip nach den optionalen zu-1 Links eingeordnet. Die nachfolgende Tabelle 6.7 zeigt die sich daraus ergebende Prioritätenskala.

Das Beispiel in der Abbildung 6.10 veranschaulicht die Teilgraphensuche für die Multilinks. Es zeigt zwei Multilink-Pfade und eine Variable, die jeweils mit derselben Container-Variable verbunden sind. Die Multilinks sind von eins bis zehn durchnummeriert, damit nachher auf sie referenziert werden kann. Nachdem die Container-Variable gebunden wurde, erfolgt eine erste Bewertung der Multilinks. Das Multilink 1 erhält die Priorität *P-First*, Multilink 7 die Priorität *P-Bound-to-Unbound*, die Multilinks 6 und 8 die Priorität *P-Unbound-to-Bound* und die Multilinks 2 und 6 erhalten die Priorität *P-Entry*. Die restlichen Multilinks werden vorerst nicht betrachtet und erhalten deshalb die Priorität *P-None*. Der zu-n Link zu der Variablen *r1* erhält die entsprechende Priorität *P-To-Many*. Der Multilink 7 mit der höchsten Priorität wird zuerst ausgeführt. Die Variable *t3* wird dadurch gebunden. Das Binden der Variablen *t3* bewirkt eine Neubewertung des achten Multilinks. Weil die Source-Variable gebunden ist und die beiden Variablen normal sind, bekommt es ebenfalls die Priorität *P-Bound-to-Unbound* und wird deshalb als nächstes ausgeführt. Als nächstes werden die beiden Multilinks 6 und

| |
|---|
| P-Check |
| P-To-One |
| P-Multilink-Check |
| P-Multilink-First |
| P-Multilink-Last |
| P-Multilink-Bound-to-Unbound |
| P-Multilink-Unbound-to-Bound |
| P-Multilink-Entry |

| |
|---|
| P-Multilink-Path |
| P-To-Many |
| P-Optional-Check |
| P-Optionale-To-One |
| P-Multilink-Entry-Optional |
| P-Multilink-Path-Optional |
| P-Optional-To-Many |
| P-Set |

Tabelle 6.7: Prioritäten in absteigender Reihenfolge

10 mit der Priorität *P-Unbound-to-Bound* bearbeitet. Die Reihenfolge der Ausführung ist nicht genau bestimmbar, da sie aufgrund des FIFO-Prinzips davon abhängt, welcher der beiden Multilinks zuerst bewertet worden ist. Dies ist bei der erstmaligen Bewertung der Multilinks nicht eindeutig, da die Reihenfolge in der Multilinks zunächst bewertet werden beliebig ist. Es wird davon ausgegangen, daß zuerst Multilink 6 ausgeführt wird. Das Multilink bindet dementsprechend die Variable *t1*. Anschließend wird durch das Multilink 10 die Variable *t5* gebunden. Dies führt zu einer Neubewertung des benachbarten *indirect*-Multilinks 9. Die beiden Variablen des *indirect*-Multilinks sind bereits gebunden worden, deshalb handelt es sich bei diesem Multilink um einem Check-Multilink. Weil das Check-Multilink in diesem Fall die höchste Priorität hat, wird es als nächstes ausgeführt. Dieser Multilink-Pfad ist damit komplett gebunden.

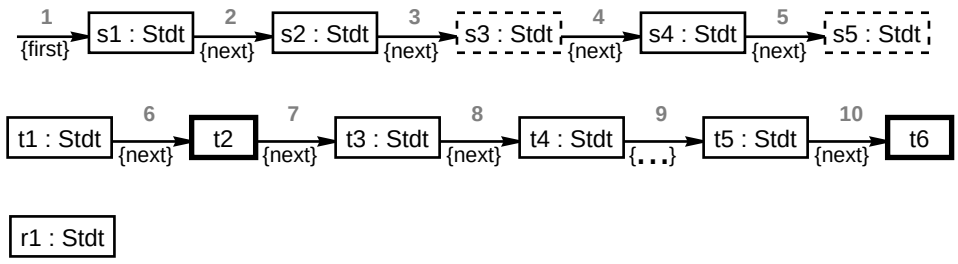



Abbildung 6.10: Beispiel für die optimierte Teilgraphensuche

Übrig bleiben noch die drei Multilinks 1, 2 und 6 und das zu-n Link. Das Multilink 1 besitzt die Priorität *P-First* und damit die höchste Priorität, weshalb es als nächstes ausgeführt wird. Die Variable *s1* wird mit dem er-

sten Element des Containers gebunden. Dadurch kann die Variable *s2* über den Multilink 2 gebunden werden. Damit bleiben nur noch das zu-n Link und die beiden *entry*-Links übrig, die noch ausgeführt werden müssen. Die *entry*-Links besitzen die höhere Priorität und werden deshalb zuerst ausgeführt. Die Reihenfolge der Ausführung ist ebenfalls nicht genau bestimmt. Der *entry*-Link 6 löst die zweiphasige Teilgraphensuche für den entsprechenden Multilink-Pfad aus. Dieser Multilink-Pfad ist aber bereits vollständig gebunden worden, deshalb wird nur über diesem Multilink-Pfad traversiert ohne eine Variable zu binden. Erst danach kann die Teilgraphensuche davon ausgehen, das wirklich jede Variable in diesem Multilink-Pfad gebunden wurde.

Danach wird das *entry*-Link 2 ausgeführt. Hierbei wird interne Variable *bindOptionalAndSet* des *entry*-Links auf `true` gesetzt und alle Multilinks des entsprechenden Multilink-Pfades werden erneut bewertet. Die Multilinks 3, 4 und 5 erhalten die Priorität *P-Path*. Das *entry*-Link 2 bekommt durch das setzen der internen Variable die neue Priorität *P-Entry-Optional*. Die Multilinks 3, 4 und 5 besitzen damit die höchste Priorität. Aufgrund der festgelegten Bewertungsreihenfolge und des FIFO-Prinzips wird der Multilink 3 vor den Multilinks 4 und 5 ausgeführt. Die einzige ungebundene Variable in diesem Multilink ist aber optional, deshalb wird dieser Multilink vorerst nicht ausgeführt. Die Teilgraphensuche wird mit dem Multilink 4 fortgesetzt. Dieser Multilink besitzt eine ungebundene Variable vom Typ *normal*, die daraufhin gebunden wird. Der Multilink 5 verweist ebenfalls nur auf eine optionale Variable und wird deshalb nicht ausgeführt. Die erste Phase der Teilgraphensuche ist damit abgeschlossen.

Bevor aber mit der zweiten Phase der Teilgraphensuche begonnen werden kann, wird der zu-n Link zu der Variablen *r1* ausgeführt. Der zu-n Link verweist auf eine Variable vom Typ *normal* und wird deshalb höher bewertet als der *entry*-Link. Nachdem die Variable gebunden wurde, kann die zweite Phase mit dem *entry*-Link begonnen werden. Die Multilinks 2, 3, 4 und 5 werden mit der Priorität *P-Path-Optional* in dieser Reihenfolge bewertet. Weil keine weiteren Links oder Multilinks existieren, werden diese Multilinks entsprechend der Reihenfolge abgearbeitet. Zuerst wird also Multilink 2 ausgeführt. In diesem Multilink sind aber beide Variablen bereits gebunden, weshalb die Teilgraphensuche mit dem Multilink 3 fortgesetzt wird. In der zweiten Phase kann nun die optionale Target-Variable *s3* des Multilinks gebunden werden. Multilink 4 besitzt keine ungebundenen Variablen und wird deshalb übersprungen. Mit dem Multilink 5 wird die letzte optionale Variable gebunden und die Teilgraphensuche für dieses Beispiel ist damit komplett.

# Kapitel 7

# Beispielsitzung

(Kan-Hung Wan, Kan-Sing Wan)

Anhand eines Beispiels sollen einige der in dieser Arbeit eingeführten Sprachkonstrukte in Fujaba veranschaulicht werden. Für das Beispiel soll eine Methode entwickelt werden, die für eine Vorlesungsprüfung zwei freie Räume sucht in der die Prüfung abgehalten wird. Durch die Angabe eines Nachnamens eines Studenten sollen automatisch zwei Gruppen gebildet und auf die zwei Räume verteilt werden.

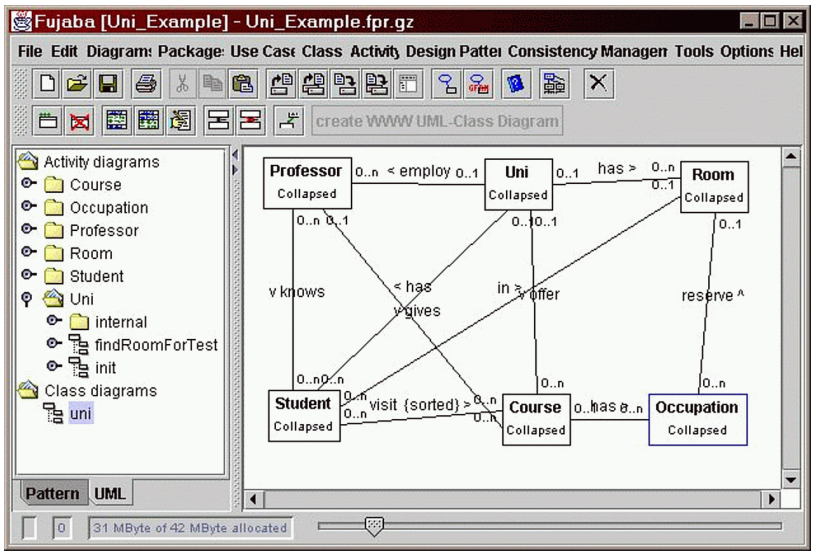



Abbildung 7.1: Klassendiagramm in Fujaba

Zu diesem Zweck wird die Klasse *Uni* um die Methode `findRoomForTest(..)` erweitert. Als Parameter werden der Methode die zu prüfende Vorlesung, der Prüfungszeitraum und der Name eines Studenten übergeben. Zuerst wurde also mit Fujaba das Klassendiagramm aus dem ersten Kapitel modelliert (siehe Abbildung 7.1). Die Methode wird mit Hilfe eines Story Diagramms implementiert. Das Story Diagramm besteht aus zwei Story-Pattern (siehe Abbildung 7.2). In dem ersten Story Pattern wird versucht zwei freie Räume für die Zeitspanne von *pStart* bis *pEnd* zu finden. Bei der Suche nach einem freien Raum muß darauf geachtet werden, daß es keine zeitliche Überschneidungen mit anderen Raumbelegungen gibt. Ein Raum gilt als frei, wenn innerhalb der angegebenen Zeitspanne keine anderen Raumbelegungen beginnen, enden oder die angegebene Zeitspanne vollständig überdecken. Diese drei Bedingungen werden jeweils durch einen negativen Knoten spezifiziert. An den Variablen *r1* und *r2* hängen deshalb jeweils drei negative Knoten. Die drei negativen Knoten werden dabei nacheinander überprüft.

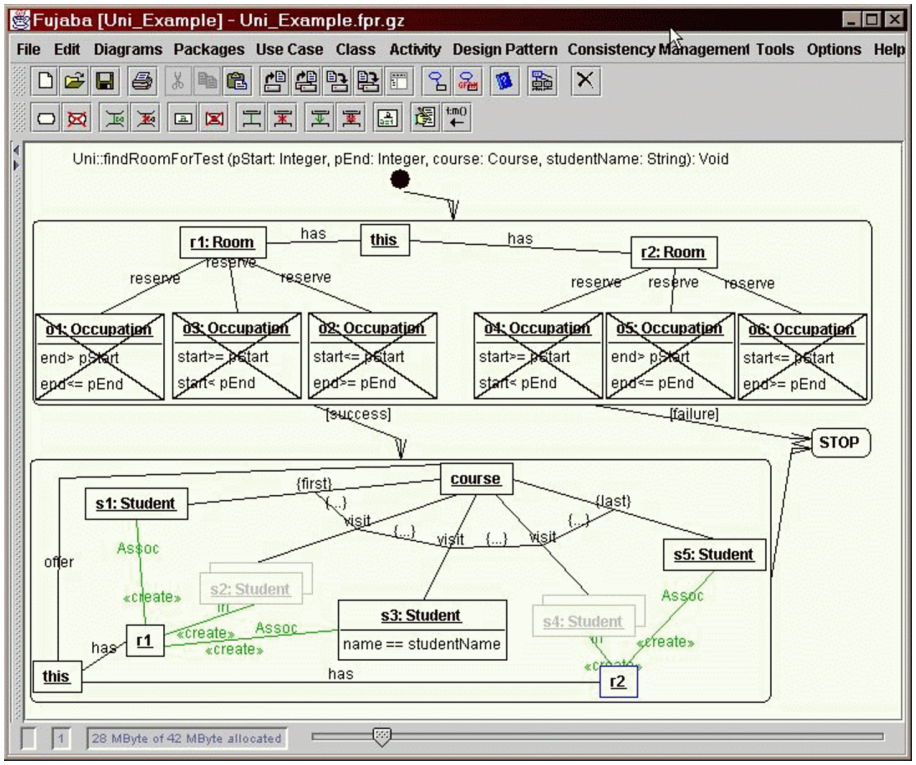



Abbildung 7.2: Story Diagramm der Methode `findRoomForTest()`

Konnten im ersten Story-Pattern zwei freie Räume gefunden werden, so wird das zweite Story-Pattern ausgeführt. Über die als Parameter übergebene

Vorlesung *course* können die Studenten dieser Vorlesung bestimmt werden. Die Studenten sind in der Vorlesung alphabetisch nach ihrem Namen sortiert (siehe Klassendiagramm). Die Aufteilung der Studenten in zwei Gruppen geschieht über den als Parameter übergebenen Namen *studentName*. Die erste Gruppe besteht aus dem alphabetisch ersten Studenten bis zum Studenten mit dem Namen *studentName*. Die zweite Gruppe enthält die übrigen Studenten der Vorlesung. Die Aufteilung der Studenten wird durch Multilinks realisiert. Zuerst werden die *first*- und *last*-Multilinks ausgeführt. Anschließend wird versucht die Variable *s3* zu binden. Die Mengenvariablen *s2* und *s4* werden zuletzt ausgeführt. Nachdem alle Variablen erfolgreich gebunden werden konnten, werden die Modifikationen ausgeführt. Die Studenten werden dadurch ihren Räumen zugeordnet.

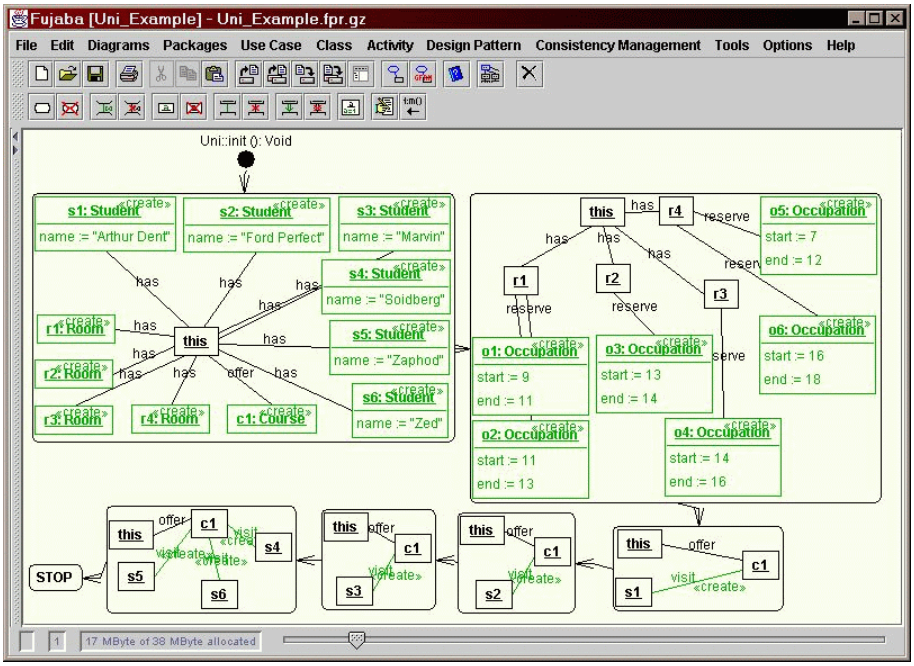



Abbildung 7.3: Story Diagramm der Methode `init()`

Aufgrund einiger Probleme mit den Displaystrukturen konnten die *first*- und *last*-Multilinks nicht richtig dargestellt werden. Deshalb werden die *first*- und *last*-Multilinks zur Zeit nur durch die Beschriftungen {`first`} und {`last`} über den entsprechenden Links kenntlich gemacht. Wegen der fehlenden graphischen Repräsentation der *first*- und *last*-Multilinks in den Story-Patterns, können sie deshalb auch noch nicht direkt gelöscht werden. Um ein *first*- oder *last*-Multilink aus dem Story-Pattern zu entfernen, muß der Link gelöscht werden auf dem das *first*- oder *last*-Multilink verweist.

Dadurch werden sie indirekt gelöscht.

Um zu zeigen das die Methode `findRoomForTest(..)` mit den neuen Sprachelementen auch funktioniert, wurde das Tool Mr. Dobs (Dynamic Object Browsing System) zur Hilfe genommen. Mit Hilfe von Dobs kann eine Laufzeitobjektstruktur visualisiert werden. Zu Testzwecken wurde deshalb die Klasse *Uni* um die Methode `init()` (siehe Abbildung 7.3) erweitert, die eine Laufzeitobjektstruktur erstellt, um die Methode `findRoomForTest(..)` zu testen. Mit Hilfe von Dobs wird nun der Zustand der Laufzeitobjektstruktur vor und nach Ausführung der Methode `findRoomForTest(..)` festgehalten, um die Ausführung der Methode zu dokumentieren.

Die folgende Abbildung 7.4 zeigt die Laufzeitobjektstruktur direkt nach dem Aufruf der Methode `init()`:

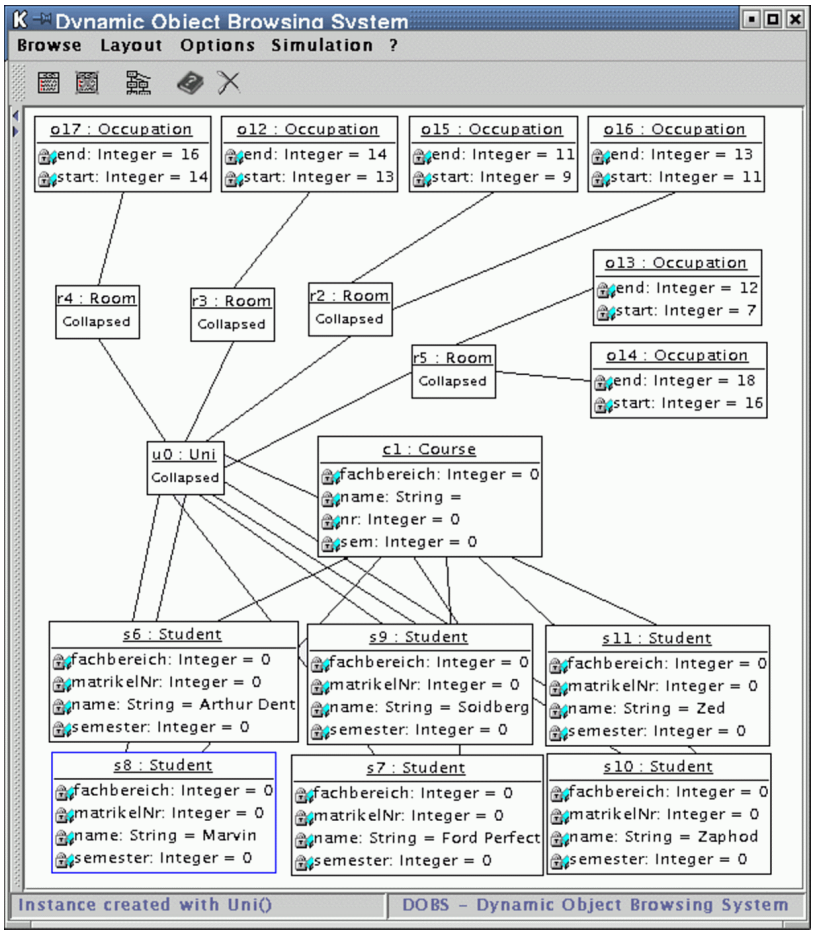



Abbildung 7.4: Laufzeitobjektstruktur nach Ausführung der Methode `init()`

Zu diesem Zeitpunkt sind die Studenten noch keinem Raum zugeordnet. Die Methode `findRoomForTest(..)` wird nun mit der Vorlesung *c1*, dem Prüfungszeitraum von 9 bis 11 Uhr und dem Studentennamen Marvin aufgerufen. Die Abbildung 7.5 zeigt das Ergebnis nach Ausführung der Methode. Zur besseren Übersicht wurden in der Abbildung einige Klassen ausgeblendet. Man erkennt das die Studenten in zwei Gruppen aufgeteilt und jeweils den Räumen *r3* und *r4* zugeordnet worden sind. Die Methode `findRoomForTest(..)` konnte also erfolgreich ausgeführt werden. Im Anhang A befindet sich der von Fujaba generierte Sourcecode für die Methode `findRoomForTest(..)`.

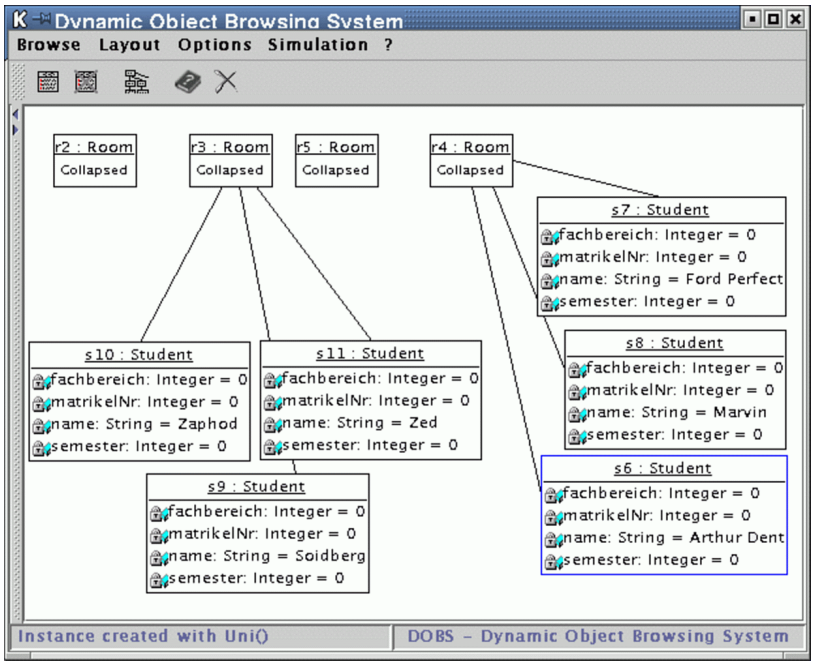



Abbildung 7.5: Laufzeitobjektstruktur nach Ausführung der Methode `findRoomForTest()`

# Kapitel 8

# Zusammenfassung und Ausblick

(Kan-Hung Wan, Kan-Sing Wan)

Ziel dieser Studienarbeit ist die Erweiterung der Java Codegenerierung der Entwicklungsumgebung Fujaba um Attributbedingungen für negative Knoten und Multilinks. Dies umfat ebenso eine Erweiterung der Zugriffsmethoden für Beziehungen zwischen Klassen, sowie eine Änderung an der optimierten Teilgraphensuche für Story-Diagramme.

In dieser Arbeit wurde die Syntax und Semantik von Negativen-Knoten und Negativen-Links näher erläutert. Insbesondere wurde auf die vielen Sonderfälle von negativen Knoten und Links eingegangen, wie zum Beispiel die Semantik vom gebundenen negativen Knoten mit Attributbedingungen oder die Semantik von einem negativen Link an einer gebundenen Menge. Es wurde erklärt wie Fujaba die Teilgraphensuche intern durch Prioritätsklassen realisiert und nach welchen Strategien es vorgeht, um eine Optimierung der Suche zu erreichen. Anschließend wurde die Codegenerierung speziell für Negative-Knoten und -Links erklärt. Dabei wurde zum Beispiel auf die negativen rückwärts qualifizierten Links eingegangen.

Die Codegenerierung für die unterschiedlichen negativen Typen bildet auch die Grundlage für ihre Einordnung in die entsprechenden Prioritätsklassen. Dabei wurden die Prioritätsklassen *P_CHECK* und *P_CHECK_TO_MANY* eingeführt. In der Prioritätsklasse *P_CHECK* werden negative Sprachelemente eingeordnet, die mit einem geringem Suchaufwand verbunden sind. Im Gegensatz dazu fallen in die Klasse *P_CHECK_TO_MANY* negative Knoten und Links mit weniger restriktiven Bedingungen rein. Speziell für die negativen Links gibt es noch die Prioritätsklasse *P_NONE*. Besitzt eine ungebundene Target-Variable mehr als

einen Link, so darf sie noch nicht bearbeitet werden was durch die Vegabe der Priorität *P_NONE* erreicht wird. Dadurch wird die Semantik des negativen Link eindeutigt definiert unabhängig von der Bearbeitungsreihenfolge der Links in der Teilgraphensuche.

In UML-Klassendiagrammen können zu-n Assoziation über Constraints als sortiert oder geordnet spezifiziert werden. Um dies in einem Story-Pattern zu verdeutlichen, werden Multilinks benutzt, die Ordnungsbeziehungen zwischen Elementen derselben Assoziation beschreiben. In Kapitel 3 wurde die Semantik und die Syntax der einzelnen Multilink-Typen vorgestellt. Der *direct*-Multilink spezifiziert eine direkte Nachfolger- oder Vorgängerbeziehung zwischen zwei Variablen. Der *index*-Multilink ist eine Verallgemeinerung des *next*-Multilinks. Über einem Index kann der Abstand zwischen den zwei Variablen innerhalb der Sortierung angegeben werden. Der *indirect*-Multilink modelliert eine indirekte Ordnungsbeziehung zwischen zwei Variablen und die *first*- und *last*-Multilinks binden das erste oder letzte Element eines Containers an einer Variablen.

Es ist möglich die einzelnen Multilinks miteinander zu verketten. Das Ergebnis ist ein Multilink-Pfad. Beim Binden der Variablen eines Multilink-Pfades kommt hinzu, daß die Transitivität eingehalten werden muß. Zum Beispiel muß das erste Element eines Containers in einem Multilink-Pfad immer vor allen anderen Elemente des Containers liegen.

Kapitel 6 behandelt die Codegenerierung der Multilinks und der Multilink-Pfade. Für die Multilinks wurden dafür neue Zugriffsmethoden eingeführt und die Klasse *FLinkedList*, das als Container für die sortierten oder geordneten zu-n Assoziationen verwendet wird, wurde entsprechend erweitert. Die bisher implementierte Teilgraphensuche in Fujaba mußte erweitert werden, da die Behandlung von Multilinks noch nicht implementiert wurde. Hierzu wurden neue Prioritätsklassen für die Multilinks eingeführt, die neben einer Optimierung der Teilgraphensuche nach dem Force-Failure-Prinzip auch Transitivität der Multilink-Pfade berücksichtigen. Für die implementierte Teilgraphensuche der Multilink-Pfade wurde zudem noch das Konzept der unteren und oberen Schranken entwickelt, die beim Binden der einzelnen Variablen des Multilink-Pfades die transitive Eigenschaft garantieren sollen. Am Ende wurde die Prioritätenskala um die neuen Prioritätsklassen ergänzt, wobei die Einordnung nach dem Force-Failure-Prinzip geschah.

Die in dieser Arbeit vorgestellten Sprachelemente und Konzepte wurden in die Entwicklungsumgebung Fujaba vollständig implementiert. Was bisher noch fehlt ist eine Kombination der beiden Sprachelemente Negative-Knoten

und Multilinks. Bei der Anwendung eines *indirect*-Multilinks ist es zum Beispiel manchmal sinnvoll bestimmte Nachfolger durch negative Kriterien auszuschließen zu können.

# Anhang A

# Beispiel Sourcecode

Von Fujaba generierter Sourcecode für die Methode `findRoomForTest(..)`.

```
/**
 * UMLMethod: findRoomForTest (pStart: Integer, pEnd: Integer,
 *                             course: Course, studentName: String): Void
 */
public void findRoomForTest (int pStart, int pEnd, Course course,
                             String studentName)
{
   Occupation o1 = null;
   Occupation o2 = null;
   Occupation o3 = null;
   Occupation o4 = null;
   Occupation o5 = null;
   Occupation o6 = null;
   Room r1 = null;
   Room r2 = null;
   Student s1 = null;
   TreeSet s2 = null;
   Student s3 = null;
   TreeSet s4 = null;
   Student s5 = null;

   boolean fujaba__Success = false;
   Object fujaba__TmpObject = null;

   try
   {
      fujaba__Success = false;
      // bind r2: Room
      Iterator fujaba__IterThisRoomR2 = this.iteratorOfRoom ();
      while (!fujaba__Success && fujaba__IterThisRoomR2.hasNext ())
```

```
{
   try
   {
      r2 = (Room) fujaba__IterThisRoomR2.next ();

      // check To-Many-Link 'occupation' between r2 and o4
      Iterator fujaba__IterR2Occupation04 =
         r2.iteratorOfOccupation ();
      while (fujaba__IterR2Occupation04.hasNext ())
      {
         o4 = (Occupation) fujaba__IterR2Occupation04.next ();

         JavaSDM.ensure (!( (o4.getStart () >= pStart) &&
                            (o4.getStart () < pEnd)) );
      }  // while

      // check To-Many-Link 'occupation' between r2 and o5
      Iterator fujaba__IterR2Occupation05 =
         r2.iteratorOfOccupation ();
      while (fujaba__IterR2Occupation05.hasNext ())
      {
         o5 = (Occupation) fujaba__IterR2Occupation05.next ();

         JavaSDM.ensure (!( (o5.getEnd () > pStart) &&
                            (o5.getEnd () <= pEnd)) );
      }  // while

      // check To-Many-Link 'occupation' between r2 and o6
      Iterator fujaba__IterR2Occupation06 =
         r2.iteratorOfOccupation ();
      while (fujaba__IterR2Occupation06.hasNext ())
      {
         o6 = (Occupation) fujaba__IterR2Occupation06.next ();

         JavaSDM.ensure (!( (o6.getStart () <= pStart) &&
                            (o6.getEnd () >= pEnd)) );
      }  // while

      // bind r1: Room
      Iterator fujaba__IterThisRoomR1 = this.iteratorOfRoom ();
      while (!fujaba__Success && fujaba__IterThisRoomR1.hasNext ())
      {
         try
         {
            r1 = (Room) fujaba__IterThisRoomR1.next ();

            // check isomorphic binding
            JavaSDM.ensure (!(r2.equals (r1)));
```

```
            // check To-Many-Link 'occupation' between r1 and o1
            Iterator fujaba__IterR1OccupationO1 =
               r1.iteratorOfOccupation ();
            while (fujaba__IterR1OccupationO1.hasNext ())
            {
               o1 = (Occupation) fujaba__IterR1OccupationO1.next ();

               JavaSDM.ensure (!( (o1.getEnd () > pStart) &&
                                  (o1.getEnd () <= pEnd)) );
            }  // while

            // check To-Many-Link 'occupation' between r1 and o2
            Iterator fujaba__IterR1OccupationO2 =
               r1.iteratorOfOccupation ();
            while (fujaba__IterR1OccupationO2.hasNext ())
            {
               o2 = (Occupation) fujaba__IterR1OccupationO2.next ();

               JavaSDM.ensure (!( (o2.getStart () <= pStart) &&
                                  (o2.getEnd () >= pEnd)) );
            }  // while

            // check To-Many-Link 'occupation' between r1 and o3
            Iterator fujaba__IterR1OccupationO3 =
               r1.iteratorOfOccupation ();
            while (fujaba__IterR1OccupationO3.hasNext ())
            {
               o3 = (Occupation) fujaba__IterR1OccupationO3.next ();

               JavaSDM.ensure (!( (o3.getStart () >= pStart) &&
                                  (o3.getStart () < pEnd)) );
            }  // while

            fujaba__Success = true;
         }
         catch (JavaSDMException fujaba__InternalException)
         {
         }  // try catch

      } // while fujaba__IterThisRoomR1

   }
   catch (JavaSDMException fujaba__InternalException)
   {
   }  // try catch

} // while fujaba__IterThisRoomR2

}
```

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

if (fujaba__Success)
{
   try
   {
      fujaba__Success = false;
      // check isomorphic binding
      JavaSDM.ensure (!(r1.equals (r2)));

      // check object is really bound
      JavaSDM.ensure (r1 != null);

      // check object is really bound
      JavaSDM.ensure (course != null);

      // check object is really bound
      JavaSDM.ensure (r2 != null);

      // check To-One-Link 'uni' between course and this
      JavaSDM.ensure (course.getUni () != null &&
                      course.getUni ().equals (this));

      // check To-One-Link 'uni' between r1 and this
      JavaSDM.ensure (r1.getUni () != null &&
                      r1.getUni ().equals (this));

      // check To-One-Link 'uni' between r2 and this
      JavaSDM.ensure (r2.getUni () != null &&
                      r2.getUni ().equals (this));

      // multiLink bind s1 : Student
      s1 = (Student)course.getFirstOfStudent ();
      JavaSDM.ensure (s1 != null);

      // check isomorphic binding
      JavaSDM.ensure ( (s4 == null || !s4.contains (s1)));

      // multiLink bind s5 : Student
      s5 = (Student)course.getLastOfStudent ();
      JavaSDM.ensure (s5 != null);

      // check isomorphic binding
      JavaSDM.ensure ( (s4 == null || !s4.contains (s5)) &&
                      !(s1.equals (s5)));
```

```
// multilink bind s3 : Student
Iterator fujaba__IterS3 = course.iteratorOfStudent (s1);
boolean fujaba__UpperBound_S3 = false;
while (!fujaba__Success && fujaba__IterS3.hasNext () &&
       !fujaba__UpperBound_S3)
{
   try
   {
      s3 = (Student) fujaba__IterS3.next ();

      if (s3.equals(s5))
      {
         fujaba__UpperBound_S3 = true;
         throw new JavaSDMException ();
      }
      // precondition check
      JavaSDM.ensure ((s3.getName () != null) &&
                      (s3.getName ().compareTo (studentName) == 0));
      // check isomorphic binding
      JavaSDM.ensure ((s4 == null || !s4.contains (s3)) &&
                     !(s1.equals (s3)) && !(s5.equals (s3)));

      // multilink set bind s2 : Student
      s2 = new TreeSet (new StandardComparator());
      Iterator fujaba__IterS2 = course.iteratorOfStudent (s1);

      while  (fujaba__IterS2.hasNext ())
      {
         try
         {
            fujaba__TmpObject = fujaba__IterS2.next();
            if (fujaba__TmpObject.equals(s3))
            {
               break;
            }  // fi

            s2.add (fujaba__TmpObject);
         }
         catch (JavaSDMException fujaba__InternalException)
         {
         }  // try catch

      }
      // check isomorphic binding
      s2.remove (s1);
      s2.remove (s3);
      s2.remove (s5);
      // check multilink s2 to s3
      JavaSDM.ensure (s2 == null ||
```

```
   course.isBeforeOfStudent ((Student) s2.last(), s3));

// multilink set bind s4 : Student
s4 = new TreeSet (new StandardComparator());
Iterator fujaba__IterS4 = course.iteratorOfStudent (s3);

while  (fujaba__IterS4.hasNext ())
{
   try
   {
      fujaba__TmpObject = fujaba__IterS4.next();
      if (fujaba__TmpObject.equals(s5))
      {
         break;
      }  // fi

      s4.add (fujaba__TmpObject);
   }
   catch (JavaSDMException fujaba__InternalException)
   {
   }  // try catch

}
// check isomorphic binding
s4.removeAll (s2);
s4.remove (s1);
s4.remove (s3);
s4.remove (s5);
// check multilink s4 to s5
JavaSDM.ensure (s4 == null ||
   course.isBeforeOfStudent ((Student) s4.last(), s5));

// create link
s1.setRoom (r1);

// create link
s3.setRoom (r1);

// create link
s5.setRoom (r2);

// create link
Iterator fujaba__IterR1RoomS2 = s2.iterator ();
while  (fujaba__IterR1RoomS2.hasNext () )
{
   Student fujaba__TmpObjectStudent =
      (Student) fujaba__IterR1RoomS2.next();
   fujaba__TmpObjectStudent.setRoom (r1);
}  // while
```

```
            // create link
            Iterator fujaba__IterR2RoomS4 = s4.iterator ();
            while  (fujaba__IterR2RoomS4.hasNext () )
            {
               Student fujaba__TmpObjectStudent =
                  (Student) fujaba__IterR2RoomS4.next();
               fujaba__TmpObjectStudent.setRoom (r2);
            }  // while

            fujaba__Success = true;
         }
         catch (JavaSDMException fujaba__InternalException)
         {
         }  // try catch

      }  // while fujaba__IterS3

   }
   catch (JavaSDMException fujaba__InternalException)
   {
      fujaba__Success = false;
   }  // try catch

}

return;
}
```

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Literaturverzeichnis


[1] Dorothea Blostein, Andy Schürr. *Computing with Graphs and Graph Rewriting*. Technical Report AIB 97-8, Fachgruppe Informatik, RWTH Aachen, Germany

[2] Fujaba Homepage. `http://www.fujaba.de`, August 2001

[3] Progres Homepage. `http://www-i3.informatik.rwth-aachen.de /research/projects/progres`, August 2001

[4] Andy Schürr. *Progress for Beginners*. Fachgruppe Informatik, RWTH Aachen 1997, Germany

[5] Albert Zündorf. *Folienskript zur Vorlesung Graphentechnik SS00*. Universität-Gesamthochschule Paderborn, 2000

[6] Albert Zündorf. *Eine Entwicklungsumgebung für PROgrammierte GRaphersEtzungsSysteme - Spezifikation, Implementierung und Verwendung*. Dissertation (in German), RWTH Aachen 1995

[7] T.H. Cormen, C.E. Leiserson, R.L. Rivest. *Introduction to Algorithms*. MIT Press, 1998

[8] Martin Fowler. *UML konzentriert - eine strukturierte Einführung in die Standard-Objektmodellierungssprache*. Addison-Wesley-Longman Verlag, 2000 2. aktualisierte Auflage

[9] Bruce Eckel. *Thinking in Java*. Prentice Hall PTR, 1998

[10] T. Fischer, J.Niere, L. Torunski. *Konzeption und Realisierung einer integrierten Entwicklungsumgebung fur UML, Java und Story-Driven Modeling*. Diplomarbeit, Universität Paderborn Deutschland, Juli 1998

[11] Nicolai Josuttis. *Die C++ Standardbibliothek : Eine detaillierte Einführung in die vollständige ANSI/ISO-Schnittstelle*. Addison-Wesley-Longman Verlag, 1996 1. Auflage

[12] David Flanagan. *Java in a Nutshell - A Desktop Quick Reference*. O'Reilly & Associates, November 1999 Third Edition

[13] Steven S. Skiena. *The algorithm design manual*. Springer Verlag, 1998.