



Universität Paderborn

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

**Entwurf und Implementierung einer Import/Export
Funktionalität für die Entwicklungsumgebung Fujaba**

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

von

Philipp Hoven
Am Fichtenhang 26
34431 Marsberg

Mike Liebrecht
Greve Straße 5
33106 Paderborn

vorgelegt bei
Prof. Dr. Wilhelm Schäfer
und
Prof. Dr. Gerd Szwillus

Paderborn, August 2002

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

Datum, Ort

Unterschrift - Philipp Hoven

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

Datum, Ort

Unterschrift - Mike Liebrecht

Inhaltsverzeichnis

1	Einleitung	1
1.1	Zielsetzung	3
1.2	Gliederung	3
2	Lade- und Speicheralgorithmus	5
2.1	Fujaba´s Abstrakter Syntax Graph	5
2.2	Ist-Zustand Persistenz	6
2.3	Probleme bei Teildiagrammen	9
2.4	Konzepte zum Laden und Speichern von Teildiagrammen	12
2.4.1	Filterkonzept	13
2.4.2	Mergekonzept	16
2.4.3	Gesamtüberlick	17
2.5	Technische Realisierung	17
2.5.1	Ladealgorithmus	18
2.5.2	Speicheralgorithmus	22
3	Datenaustausch	27
3.1	Probleme und Lösungen	27

Inhaltsverzeichnis

3.2	Datenformate	30
3.2.1	XML	30
3.2.2	XML Metadata Interchange	32
3.2.3	GXL	33
3.2.4	<i>Fujaba P</i> Projektdatei	36
3.2.5	FujabaXML	38
3.3	Document Object Model	41
3.4	XML Datentransformation	42
3.5	Technische Realisierung	43
3.5.1	Benutzerschnittstelle	44
3.5.2	Export	46
3.5.3	Import	53
3.5.4	Weitere Transformationen	58
3.5.5	Bewertung der Transformationen	58
4	Beispielsitzung	61
5	Zusammenfassung und Ausblick	67

1 Einleitung

(Philipp Hoven, Mike Liebrecht)

In der Entwicklung von komplexen Softwaresystemen ist der Einbezug von objektorientierten Modellierungssprachen wie die Unified Modeling Language [UML] in den Unified Process von zentraler Bedeutung, da sie eine einheitliche Notation bietet. Auf Basis dieser Notation ist ein standardisierter Austausch von Informationen zwischen den verschiedenen, am Entwicklungsprozess beteiligten, Personen möglich.

Die Unified Modeling Language wurde erstmals 1997 von Grady Booch, Ivar Jacobson und James Rumbaugh vorgestellt und durch die Object Management Group [OMG] als Sprache zur Beschreibung von objektorientierten Modellen standardisiert.

UML ermöglicht die objektorientierte Modellierung von Softwaresystemen anhand definierter Diagrammartentypen. Insgesamt umfasst die Unified Modeling Language: Use-Case-Diagramme, Klassendiagramme, Interaktionsdiagramme, Zustandsdiagramme, Aktivitätsdiagramme, Packagediagramme und Implementierungsdiagramme. Mit diesen Diagrammen können die verschiedenen Aspekte eines Systems modelliert werden. So wird zum Beispiel die statische Struktur eines Softwaresystems mittels Klassendiagrammen dargestellt, während Aktivitätsdiagramme genutzt werden um Abläufe innerhalb des Systems zu modellieren.

In diesem Kontext wurde im Rahmen der Diplomarbeit [FNT98] an der Universität Paderborn mit der Realisierung der Entwicklungsumgebung Fujaba begonnen. Fujaba unterstützt unter anderem Klassen- und Aktivitätsdiagramme sowie Story-Diagramme. Story-Diagramme sind Bestandteil des Story-Driven-Modeling [SWZ99] kurz SDM und spezifizieren das Laufzeitverhalten von Objektstrukturen. Somit können Story-Diagramme ergänzend zu den UML-Diagrammartentypen eingesetzt werden. In Fujaba werden Story-Diagramme genutzt um das Laufzeitverhalten von Methoden zu modellieren, die anhand von Klassendiagrammen spezifiziert wurden.

Damit bietet Fujaba ein Framework zur Erstellung von UML/SDM-Diagrammen und deren Semantik. Die Entwicklungsumgebung bietet somit Möglichkeiten zur Spezifikation der statischen und dynamischen Struktur von Softwaresystemen und

1 Einleitung

der Generierung des entsprechenden Quellcodes. Da der Quellcode durch Aktivitätsdiagramme und Story-Diagramme spezifiziert wird, geht die Codegenerierung über die Ebene von leeren Methodenrumpfen hinaus. Somit wird eine allgemeine Anforderung an Modellierungswerkzeuge, die Unterstützung des Forward-Engineering, von Fujaba erfüllt.

Eine weitere Anforderung, die an Modellierungswerkzeuge gestellt wird, ist die Fähigkeit des Re-Engineering. Also, die Generierung eines UML-Modells aus vorhandenem Java Quelltext. Hierbei unterstützt Fujaba den Anwender durch einlesen des Java Quelltextes und automatischem Generieren des UML-Modells. Fujaba erstellt dabei aus dem Quelltext äquivalente Klassen- und Aktivitätsdiagramme.

Die dritte wichtige Anforderung ist die Fähigkeit des Werkzeugs das erstellte Modell zu speichern. Dabei sollte sowohl das Speichern des gesamten UML-Modells als auch der Export einzelner (Teil)-Diagramme in den verschiedensten Austauschformaten unterstützt werden. Und hier liegt eine entscheidende Schwachstelle der Entwicklungsumgebung Fujaba. Die aktuelle Version kann das erstellte Modell nur als ein zusammengehöriges Projekt in einem eigenen proprietären Format speichern. Diese Art der Speicherung beinhaltet folgende Nachteile:

- Keine Möglichkeit einzelne, im Rahmen der Spezifikation erstellte, Diagramme zu speichern und wieder zu verwenden.
- Aufgrund des proprietären Formats der Projektdatei ist es nicht möglich Spezifikationen in andere UML-Werkzeuge wie Together-J oder Rational Rose zu übernehmen.
- Keine Copy&Paste-Funktionalität.

Da im Rahmen des Entwicklungsprozesses eine Vielzahl von Entwicklern, mit den unterschiedlichsten UML-Werkzeugen, an der Spezifikation eines Softwaresystems beteiligt sind, stellen vor allem die fehlende Unterstützung von Austauschformaten und die fehlende Fähigkeit einzelne Diagramme zu exportieren ein entscheidendes KO-Kriterium für den Einsatz von Fujaba dar. Fujaba kann aufgrund dieser beiden Mängel keinen Datenaustausch mit anderen UML-Werkzeugen durchführen und ist somit nicht in den Entwicklungsprozess integrierbar.

Des weiteren führen die oben angeführten Nachteile zu einem Mangel an Anwendungskomfort. Aufgrund der fehlenden Fähigkeit einzelne (Teil)-Diagramme zu speichern besitzt Fujaba zum einen keine Copy&Paste-Funktionalität und zum anderen generell keine Möglichkeit Teile der erstellten Spezifikationen wieder zu verwenden. Daher müssen immer wiederkehrende nützliche Teilspezifikationen auch immer wieder neu "per Hand" modelliert werden.

Um diese Limitierungen aufzuheben wird Fujaba im Rahmen dieser Studienarbeit um einen flexiblen Lade- und Speicheralgorithmus für (Teil)-Diagramme, sowie einem Algorithmus zur Transformation des Fujaba-Datenformats in die Graph Exchange Language [GXL] oder XML Metadata Interchange [XMI] erweitert.

1.1 Zielsetzung

Aufgrund ihres gemeinsamen Metamodells, werden alle in Fujaba verwendeten Diagrammarten als ein Projekt in einem eigenen proprietären Format abgespeichert. Dies führt zu den in der Einleitung dargestellten Einschränkungen von Fujaba.

Daher wird im Rahmen dieser Studienarbeit ein Konzept für das Laden und Speichern von (Teil)-Diagrammen und der Transformation des fujabaeigenen Datenformats in GXL und XMI erarbeitet. Dabei lässt sich die Arbeit in zwei von einander getrennte Teilbereiche untergliedern. Zum Einen wird ein Konzept erarbeitet, das auf das vorhandene Persistenzkonzept von Fujaba aufsetzt und das Laden und Speichern von (Teil)-Diagrammen ermöglicht. Schwerpunkt dieses Teilbereiches ist die Identifikation, Abgrenzung und Konsistenzerhaltung der zu speichernden Diagramme.

Zum Anderen wird ein Konzept zur Transformation des fujabaeigenen Datenformats in standardisierte Austauschformate erarbeitet und implementiert. Dabei ist zu beachten, das der Transformationsprozess unabhängig von den zu unterstützenden Formaten zu entwerfen ist und sowohl die Transformation des Fujaba-Formats in das gewünschte Zielformat, als auch die Rückrichtung leisten muss. Schwerpunkt dieses Teilbereiches ist die Transformation des fujabaeigenen Datenformats in ein "neutrales" XML-Format um darauf aufbauend weitergehende Transformationen unter Verwendung von XSLT-Stylesheets zu ermöglichen. Zusätzlich werden im Rahmen dieser Studienarbeit die Austauschformate GXL und XMI implementiert.

1.2 Gliederung

Das nachfolgende Kapitel "Lade- und Speicheralgorithmus" führt zuerst in Fujaba's abstrakten Syntaxgraphen und das darauf aufsetzende Persistenzkonzept ein. Anschließend werden die Schwächen des vorhandenen Persistenzkonzeptes im Hinblick auf das Laden und Speichern von (Teil)-Diagrammen aufgezeigt und die daraus resultierenden Konzepte zum Laden und Speichern von (Teil)-Diagrammen vorgestellt. Zum Abschluss des Kapitels wird die technische Umsetzung des Konzeptes

1 Einleitung

erläutert. Dabei wird detailliert auf die Funktionalität der implementierten Java Klassen, sowie der Schnittstellen zum Transformationskonzept eingegangen.

Das Kapitel “Datenaustausch” stellt das Transformationskonzept vor. Zuerst werden die aus dem fujabaeigenen Datenformat resultierenden Probleme erläutert um dann das Konzept zu deren Lösung vorzustellen. Anschließend führt das Kapitel in die vom Transformationsprozess verwendeten Zwischen- und Austauschformate sowie den XML-Mechanismen zu deren Transformation ein. Darauf aufbauend wird dann zum Abschluss des Kapitels die technische Realisierung des Datenaustausch erläutert. Dabei werden detailliert die einzelnen Transformationsschritte beschrieben sowie die implementierten Java Klassen und XSLT-Stylesheets vorgestellt.

Im darauf folgenden Kapitel “Beispielsitzung” wird der Export eines UML-Klassendiagrammes aus Sicht des Anwenders erläutert und anhand von Screenshots dargestellt. Das letzte Kapitel fasst alle vorgestellten Konzepte zusammen und gibt einen Ausblick auf zukünftige mögliche Erweiterungen.

2 Lade- und Speicheralgorithmus

(Mike Liebrecht)

In Kapitel 1.1 wurde dargestellt, dass sich das Gesamtkonzept zur Realisierung einer Import- und Exportfunktionalität für Fujaba in zwei Teilbereiche aufteilen lässt. Dieses Kapitel behandelt nun den ersten Teilbereich, die Konzepte zum Laden und Speichern von Diagrammen.

Die beiden folgenden Abschnitte 2.1 und 2.2 erläutern die notwendigen Grundlagen. So führt Abschnitt 2.1 in Fujaba's Metamodell ein. Dabei wird der abstrakte Syntaxgraph detailliert erläutert. Dieser spezifiziert die interne Datenstruktur eines Projektes in Fujaba. Anschließend beschreibt Abschnitt 2.2 das auf den abstrakten Syntaxgraphen aufbauende Persistenzkonzept zur Speicherung eines Projektes.

Nachdem die ersten beiden Kapitel in den Ist-Zustand von Fujaba eingeführt haben, wird in Abschnitt 2.3 der aktuelle Zustand dem Ziel, der Import- und Exportfunktion für Diagramme, gegenübergestellt. Dabei wird deutlich, dass das vorhandene Persistenzkonzept zur Speicherung von (Teil)-Diagrammen nicht ausreicht. Abschnitt 2.4 stellt dann die Konzepte vor, die auf Basis des vorhandenen Persistenzkonzeptes, das Laden und Speichern von Diagrammen gewährleisten sollen.

Abschließend wird im Abschnitt 2.5 die technische Umsetzung der Konzepte erläutert. Dabei wird detailliert auf die implementierten Java Klassen und ihre Funktion eingegangen.

2.1 Fujaba's Abstrakter Syntax Graph

Dieser Abschnitt gibt einen Überblick über die internen Datenstrukturen von Fujaba. Dabei liegt der Focus der Betrachtung auf der Datenstruktur die für das Speichern und Laden von Modellen relevant ist, der abstrakte Syntaxgraph. Einen detaillierteren Einblick in das Metamodell von Fujaba gibt [FNT98].

Der abstrakte Syntaxgraph spezifiziert die logische Repräsentation eines in Fujaba erstellten Modells. Die Abbildung 2.1 zeigt den abstrakten Syntaxgraphen für UML-Klassendiagramme.

Der zentrale Knoten jedes abstrakten Syntaxgraphen, nachfolgend kurz ASG, ist ein Objekt der Klasse *UMLProject*. Dieses Objekt kapselt zum Einen alle relevanten Informationen eines Projektes, wie den Projektnamen, zum Anderen speichert es über die Assoziation *diags* alle im Rahmen dieses Projektes erstellten Diagramme. Ein UML-Diagramm wird im ASG durch eine Instanz der Klasse *UMLDiagram* repräsentiert. Die einzelnen Diagrammarten, wie Klassendiagramme oder Aktivitätsdiagramme, werden durch Spezialisierungen der Klasse *UMLDiagram* beschrieben. So repräsentiert eine Instanz der Klasse *UMLClassDiagram* ein Klassendiagramm, während ein Objekt des Typs *UMLActivityDiagram* die Eigenschaften eines Aktivitätsdiagramms kapselt.

Die Containerklasse *UMLDiagramItem* repräsentiert alle UML-Objekte die in einem UML-Diagramm verwendet werden können. Auch hier werden wieder die verschiedenen Typen der UML-Objekte durch Spezialisierungen der Klasse *UMLDiagramItem* beschrieben. So werden zum Beispiel die UML-Objekte *Klasse* und *Assoziation* durch die von *UMLDiagramItem* ererbenden Klassen *UMLClass* und *UMLAssoc* dargestellt. Die Grafik 2.1 zeigt noch weitere Beispiele.

UML-Objekte wie Klassen, Methoden und Attribute benötigen zum beschreiben ihrer Eigenschaften UML-Typen. Diese Typen lassen sich in zwei Kategorien gliedern. Zum Einen die Java Grundtypen wie boolean, Integer und float und zum Anderen werden auch Java Klassen selbst als Typen genutzt. Im ASG wird diese Gliederung durch die Klassen *UMLType*, *UMLBaseType* und *UMLClass* umgesetzt. *UMLType* dient als Containerklasse und kapselt die gemeinsamen Eigenschaften der beiden Spezialisierungen *UMLBaseType* und *UMLClass*. *UMLBaseType* repräsentiert den Grundtyp und eine Instanz der Klasse *UMLClass* die dementsprechende Java-Klasse.

Der abstrakte Syntaxgraph für Aktivitätsdiagramme ist weitgehend äquivalent aufgebaut. Die Unterschiede zum ASG des Klassendiagrammes sind auf Ebene der UML-Objekte zu finden. Es werden in der UML zur Beschreibung eines Aktivitätsdiagrammes andere Objekte mit anderen Eigenschaften genutzt und dies schlägt sich im ASG für Aktivitätsdiagramme nieder.

2.2 Ist-Zustand Persistenz

Um das Speichern und spätere Weiterbearbeiten eines Projektes in Fujaba zu ermöglichen, wurde im Rahmen der Diplomarbeit [FNT98] ein Persistenzkonzept

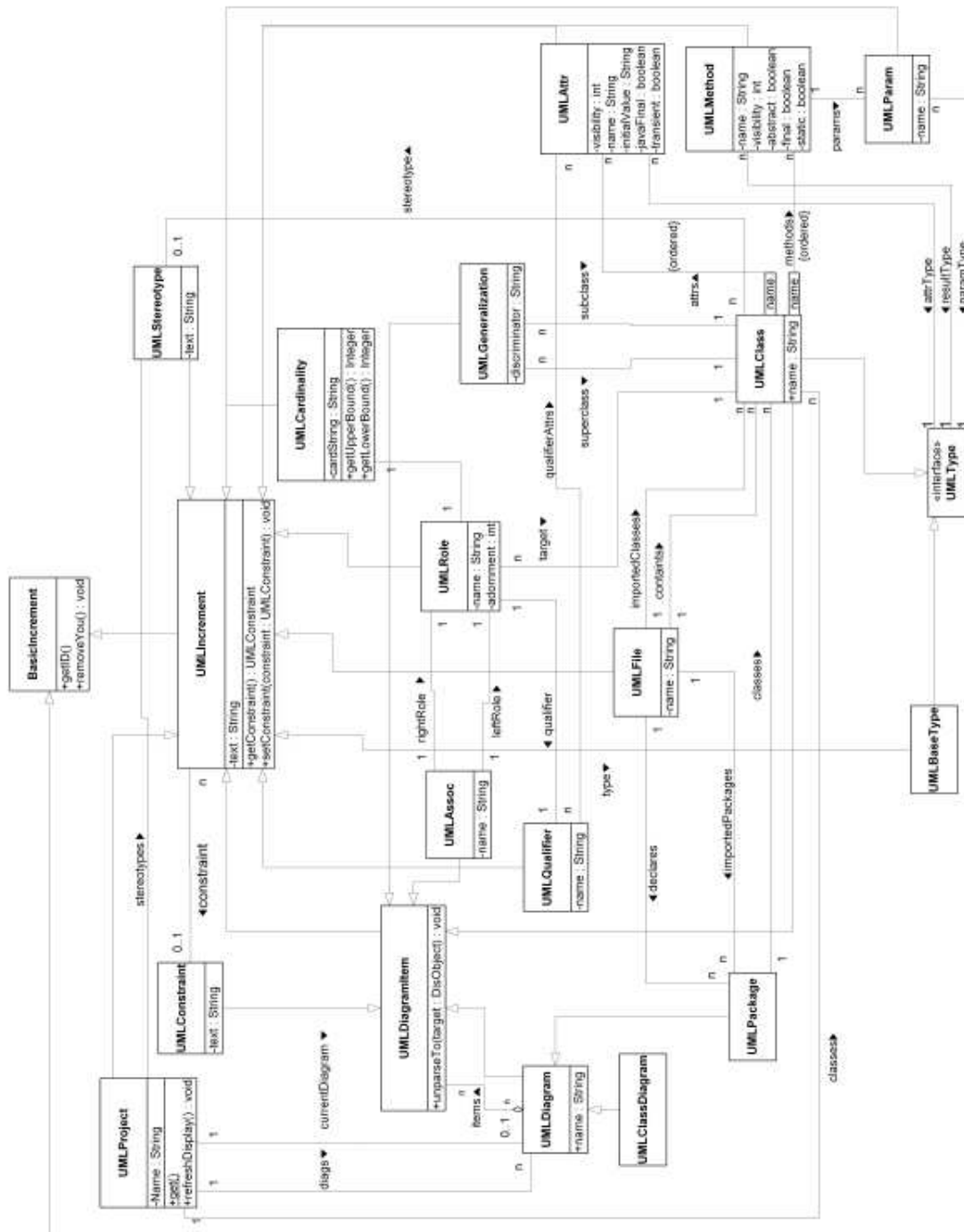


Abbildung 2.1: Abstrakter Syntaxgraph (entnommen aus [FNT98])

entwickelt. Aufgabe dieses Konzeptes ist die Abbildung des abstrakten Syntaxgraphen, als logische Repräsentation des erstellten UML-Modells, in eine ASCII-Datei. Sowie die Rückrichtung: Das Generieren des ASG aus der Projektdatei.

Aufgrund der hohen Zahl an der Entwicklung und Pflege von Fujaba beteiligter Personen und den immer wieder auftretenden Problemen, durch flüchtig implementierten Sourcecode, wurde auf ein statisches Speicherkonzept verzichtet. Da ansonsten jedes UML-Objekt Methoden zur Speicherung seiner Eigenschaften bereitstellen müsste. Somit hätte jede Änderung am UML-Objekt auch eine Modifikation der Methode zum Speichern des Objektes nach sich gezogen.

Ziel war daher die Entwicklung eines generischen Algorithmus, basierend auf dem Java Reflection Mechanismus, der auf alle Objekte des abstrakten Syntaxgraphen angewandt werden konnte.

Implementiert wurde dieser Algorithmus in der Klasse *BasicIncrement*. Da die Klasse *BasicIncrement* die Oberklasse aller im abstrakten Syntaxgraphen verwendeten UML-Objekte ist (siehe dazu auch Grafik 2.1), ist damit auch sichergestellt das der Speicheralgorithmus auf alle Objekte des ASG angewandt werden kann. Zusätzlich zum Lade- und Speicheralgorithmus kapselt *BasicIncrement* auch den Mechanismus zur Identifikation jedes UML-Objektes. Die Methode *getID()* liefert für jedes Objekt des ASG eine eindeutige ID.

Zentraler Kern des Speicheralgorithmus ist die Methode *writeClassToStringBuffer(data: StringBuffer, allIncrements:DList)*. Sie überprüft zunächst ob das aktuelle Objekt schon gespeichert wurde. Ist dies der Fall, wird die Methode verlassen und das nächste Objekt des ASG behandelt. Wurde das aktuelle Objekt noch nicht gespeichert, so werden durch Aufruf der Methode *writeAttributes(data: StringBuffer, setOfNeighbours: OrderedSet)* die Attribute des Objekts gespeichert. Dabei werden die Attribute mittels Java Reflection Mechanismus in den StringBuffer *data* geschrieben.

Der Reflection Mechanismus des Java Development Kit bietet die Möglichkeit, über die Methode *getDeclaredFields()* alle Attribute des aktuellen Objektes zu erfragen. Dadurch wird die Implementierung einer Speichermethode für jede Klasse die ein Objekt des abstrakten Syntaxgraphen beschreibt und vor allem die Pflege dieser Methode eingespart.

Nachdem die Attribute rausgeschrieben wurden, ruft sich *writeClassToStringBuffer(data: StringBuffer, allIncrements:DList)* selbst auf alle direkten "Nachbar-Objekte" des aktuellen Objektes auf. Nachbar-Objekte sind in diesem Zusammenhang, alle referenzierten UML-Objekte, keine Datentypen.

Die Grafik 2.2 zeigt wie der Speicheralgorithmus den abstrakten Syntaxgraphen

eines Beispielmodells traversiert.

Ausgangspunkt ist das einzige *UMLProject*-Objekt des Modells. Anhand der Methode *saveProject(File outputFile)* wird der Speichervorgang gestartet. *saveProject(File outputFile)* instanziiert den *StringBuffer*, in dem der Kernalgorithmus alle Daten zwischenspeichert, schreibt den Header der Projektdatei in den *StringBuffer* und führt die Methode *writeClassToStringBuffer(data: StringBuffer, allIncrements:DList)* aus. Da das *UMLProject*-Objekt noch nicht gespeichert wurde, wird das Objekt mit seinen Attributen in den *StringBuffer* geschrieben und der Kernalgorithmus wird auf dem ersten direkten "Nachbarn", einem Objekt der Klasse *UMLClassDiagram*, ausgeführt.

Die Grafik 2.2 zeigt, wie der Algorithmus rekursiv den abstrakten Syntaxgraphen durchläuft. Angefangen beim *UMLProject*-Objekt über das erste erreichbare Diagramm, den UML-Objekten des Diagramms, bis auf die unterste Ebene dem *UMLType*-Objekt. Sind alle Objekte des linken Zweigs (*UMLClassDiagram*) gespeichert, kehrt der rekursive Algorithmus bis auf die Ebene des *UMLProject*-Objektes zurück um dann das zweite Diagramm (*UMLActivityDiagram*) zu speichern. Ist auch dieser Teil des Graphen gespeichert, terminiert der Algorithmus da alle Objekte des abstrakten Syntaxgraph erreicht wurden. Abschließend schreibt die Methode *saveProject(File outputFile)* den *StringBuffer* in die Datei.

Der bis hierhin erläuterte Aufbau des Speicheralgorithmus zeigt deutlich, dass der Algorithmus nur auf die Speicherung einer 1:1 Abbildung des abstrakten Syntaxgraphen ausgerichtet ist. Weitere Aspekte, wie die Speicherung von Teilen des Modells, Daten-Overhead und daraus resultierende Projektdateigröße oder eine Konsistenzprüfung sind nicht berücksichtigt. Jedes Attribut von jedem Objekt wird gespeichert. In wie weit die Daten des Attributes für das UML-Modell relevante Informationen enthalten wird nicht betrachtet. Hauptgesichtspunkt, nachdem der Algorithmus implementiert wurde, ist die Minimierung der Fehleranfälligkeit des Algorithmus gegenüber Sourcecodemodifikationen durch die Entwickler.

Der Ladealgorithmus des Persistenzkonzeptes ist vom Konzept und der Implementierung äquivalent zu diesem Ansatz und wird daher hier nicht näher erläutert. Weitergehende Informationen zum Ladealgorithmus und dem Aufbau der Projektdatei können [FNT98] entnommen werden.

2.3 Probleme bei Teildiagrammen

Das Ziel dieses Teils der Studienarbeit ist die Konzeption und Umsetzung eines Lade- und Speicheralgorithmus auf Basis des im Abschnitt 2.2 eingeführten Persistenzkon-

2 Lade- und Speicheralgorithmus

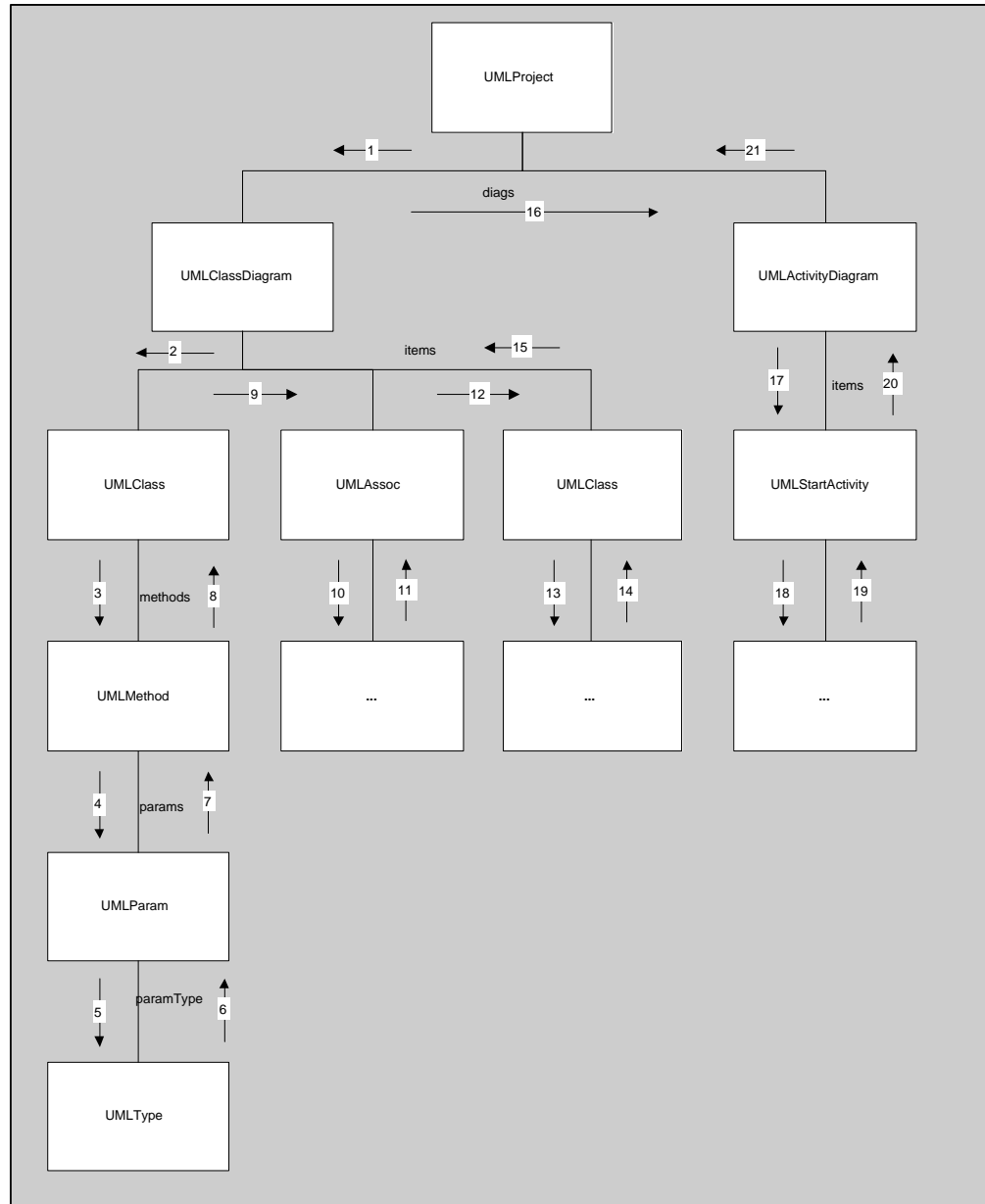


Abbildung 2.2: Ablauf des Speichervorgangs

zeptes. Das Persistenzkonzept ist aber nur auf die Speicherung eines Abbilds des gesamten abstrakten Syntaxgraphen ausgelegt. Eine Erweiterbarkeit des Persistenzkonzeptes um Diagramme oder Teile von Diagrammen speichern zu können, wurde nicht berücksichtigt.

Stellt man nun das Persistenzkonzept den Anforderungen eines Algorithmus zum Laden und Speichern von (Teil)-Diagrammen gegenüber, so werden die folgenden Probleme offensichtlich:

- Wie wird ein Diagramm im ASG identifiziert?
- Wie wird erkannt welche Objekte Bestandteil des zu speichernden Diagrammes sind?
- Ist ein Ausschnitt des ASG in sich konsistent?
- Wie werden Zusammenhänge zwischen den einzelnen Diagrammen erkannt und aufgelöst?
- Wie wird verhindert das zum Beispiel Teile eines Klassendiagrammes, in ein Aktivitätsdiagramm geladen werden?
- Beinhaltet der abstrakte Syntaxgraph UML-Objekte die jeweils nur einmal im Modell vorhanden sein dürfen ?

Fasst man die oben angeführten einzelnen Problematiken zusammen, so sind drei Problemfelder bei der Konzeption und Umsetzung des Lade- und Speicheralgorithmus für Diagramme zu berücksichtigen:

- Identifikation und Abgrenzung des zu speichernden Diagramms. Dabei sind auch Verbindungen zwischen verschiedenen Diagrammen zu erkennen und dementsprechend zu behandeln.
- Konsistenzprüfung des zu speichernden Diagramms. Das Diagramm wird durch einen Ausschnitt des ASG repräsentiert. Dieser Ausschnitt ist aber nicht automatisch in sich konsistent.
- Es sind unzulässige Kombinationen von verschiedenen Diagrammartem beim Ladevorgang zu verhindern.

2.4 Konzepte zum Laden und Speichern von Teildiagrammen

In Abschnitt 1.1 wurden die Vorgaben für die Umsetzung des Lade- und Speicheralgorithmus dargestellt. Abschnitt 2.3 stellte die drei Problemfelder die im Rahmen der Konzeption des Lade- und Speicheralgorithmus gelöst werden müssen vor. Beide, die Vorgaben und die Problemfelder, haben entscheidenden Einfluss auf die zu erarbeitenden Konzepte und werden zur Verdeutlichung noch einmal nachfolgend aufgeführt:

- Es ist ein generischer Lade- und Speicheralgorithmus zu implementieren, der auf dem vorhandenen Algorithmus zum Speichern und Laden von Fujaba-Projekten aufbaut.
- Als Austauschformat soll die Graph Exchange Language genutzt werden. Der Algorithmus ist jedoch formatunabhängig zu implementieren.
- Identifikation und Abgrenzung des zu speichernden Diagrammes. Dabei sind auch Verbindungen zwischen verschiedenen Diagrammen zu erkennen und dementsprechend zu behandeln.
- Konsistenzprüfung des zu speichernden Diagrammes. Das Diagramm wird durch einen Ausschnitt des ASG repräsentiert. Dieser Ausschnitt ist aber nicht automatisch in sich konsistent.
- Es sind unzulässige Kombinationen von verschiedenen Diagrammartent beim Ladevorgang zu verhindern.

Die erste Vorgabe legt den Ansatzpunkt des zu entwickelnden Algorithmus fest, die Klasse *BasicIncrement* welche das Persistenzkonzept kapselt.

Die zweite Anforderung führt zu einer klaren Trennung zwischen Lade-/Speicheralgorithmus und Überführung der Daten in die Graph Exchange Language. Das Ergebnis des Speicheralgorithmus wird im FPR-Format an den Transformationsprozess übergeben, um anschließend anhand der Transformationsschritte aus Kapitel 3, in das gewünschte Format transformiert zu werden. Der Ladealgorithmus arbeitet auf einer Zugriffsschicht, die vom Transformationskonzept zur Verfügung gestellt wird.

Die Identifikation und Abgrenzung des zu speichernden Diagrammes wird durch Anwendung eines Filterkonzeptes erreicht. Dieses Filterkonzept definiert für die verschiedenen in Fujaba unterstützten Diagrammartent, welche UML-Objekte Bestandteil des aktuell zu speichernden Diagramms sind, und welche nicht. Dabei

ist zu beachten das die Grenze zwischen den einzelnen Diagrammartentypen verwischt. Das heißt es reicht nicht einen Filter für Aktivitätsdiagramme und einen getrennten Filter für Klassendiagramme zu nutzen. Vielmehr muss es möglich sein die Filter kombiniert einzusetzen. Nachfolgendes Szenario verdeutlicht das Problem:

Fujaba bietet die Möglichkeit, die Implementierung einer Methode anhand eines Story-Diagramms zu spezifizieren. Dies führt dazu, dass das zu exportierende Klassendiagramm nun um ein zusätzliches Diagramm, dem Story-Diagramm für die Implementierung der Methode, ergänzt wird.

Dies ist nun eine Situation, wo verschiedene Filter kooperativ eingesetzt werden müssen, um den Gesamtkontext des Modells zu erhalten. Nutzt der Algorithmus lediglich den Klassendiagramm-Filter, so wird das Story-Diagramm nicht mit exportiert und die Implementierung der Methode geht verloren. Das heißt, der Speicheralgorithmus muss erkennen, das für Komponenten meines Diagrammes, Verbindungen zu anderen Diagrammen bestehen und diese dann gegebenenfalls unter Anwendung anderer Filter mit speichern.

Der vierte Punkt die Konsistenzprüfung wird durch das Filter- und das Mergekonzept umgesetzt. Das Filterkonzept gewährleistet die Konsistenz des zu speichernden Diagramms nach dem Speichervorgang. Das Mergekonzept ist Bestandteil des Ladevorgangs und garantiert die Konsistenz des ASG nach hinzufügen des Diagramms.

Der letzte oben angeführte Punkt führt zu einer Typprüfung des Diagramms zu Beginn des Ladevorgangs. Die Typprüfung wird im Abschnitt 2.5 "Technische Realisierung" vorgestellt.

Die nachfolgenden Abschnitte 2.4.1 und 2.4.2 führen in die einzelnen Konzepte ein. Abschließend werden dann die Einzelkonzepte in Abschnitt 2.4.3 zusammengefasst.

2.4.1 Filterkonzept

Das Filterkonzept lässt sich in drei Teilbereiche untergliedern. Die einzelnen Filter, der Algorithmus der diese Filter anwendet und das Verfahren zur Protokollierung und Bereinigung unzulässiger Referenzen. Für jeden Diagrammtyp wird ein Filter, in XML codiert, genutzt. Dieser Filter spezifiziert welche UML-Objekttypen nicht Bestandteil meines Diagramms sind und erlaubt dadurch dem Algorithmus diese Objekte zu filtern. Der nachfolgende Auszug, zeigt den in XML codierten Filter für Klassendiagramme:

2 Lade- und Speicheralgorithmus

```
1<!DOCTYPE GXLExcluded [  
2 <!ELEMENT GXLExcluded (General, UMLDiagram*)>  
3  
4 <!ELEMENT General (Class*)>  
5  
6 <!ELEMENT UMLDiagram (Class*, MergePoints*)>  
7 <!ELEMENT Class (#PCDATA)>  
8 <!ELEMENT MergePoints (#PCDATA)>  
9  
10 <!ATTLIST Diagram          class          CDATA          #REQUIRED>  
11]>  
12  
13  
14<GXLExcluded>  
15  
16 <!-- ***** General excluded ***** -->  
17  
18 <General>  
19 <Class>de.uni_paderborn.fujaba.dis.DisObject</Class>  
20 <Class>de.uni_paderborn.fujaba.fsa.FSAObject</Class>  
21 <Class>de.uni_paderborn.fujaba.dis.DisCanvas</Class>  
22 <Class>de.uni_paderborn.fujaba.dis.DisText</Class>  
23 <Class>de.uni_paderborn.fujaba.dis.DisColumn</Class>  
24 <Class>de.uni_paderborn.fujaba.dis.DisFrame</Class>  
25 <Class>de.uni_paderborn.fujaba.dis.DisResizable</Class>  
26 <Class>de.uni_paderborn.fujaba.dis.DisLine</Class>  
27 <Class>de.uni_paderborn.fujaba.dis.DisGrab</Class>  
28 <Class>de.uni_paderborn.fujaba.basic.PointIncrement</Class>  
29 </General>  
30  
31  
32 <!-- *****UMLDiagrams Excluded Classes ***** -->  
33  
34 <UMLDiagram class = "de.uni_paderborn.fujaba.uml.UMLClassDiagram" >  
35 <Class>de.uni_paderborn.fujaba.uml.UMLTransition</Class>  
36 <Class>de.uni_paderborn.fujaba.uml.UMLActivity</Class>  
37 <Class>de.uni_paderborn.fujaba.uml.UMLActivityDiagram</Class>  
38 <Class>de.uni_paderborn.fujaba.uml.UMLTransitionGuard</Class>  
39 <Class>de.uni_paderborn.fujaba.uml.UMLStopActivity</Class>  
40 <Class>de.uni_paderborn.fujaba.uml.UMLStatement</Class>  
41 <Class>de.uni_paderborn.fujaba.uml.UMLStartActivity</Class>  
42 <Class>de.uni_paderborn.fujaba.uml.UMLNopActivity</Class>  
43 <Class>de.uni_paderborn.fujaba.uml.UMLStatementActivity</Class>  
44 <Class>
```

```
45     de.uni_paderborn.fujaba.uml.UMLActivityDiagramPriorityComparator
46     </Class>
47     <Class>de.uni_paderborn.fujaba.uml.UMLComplexState</Class>
48     <Class>de.uni_paderborn.fujaba.uml.UMLConnection</Class>
49     <Class>de.uni_paderborn.fujaba.uml.UMLSequenceDiagram</Class>
50     <Class>de.uni_paderborn.fujaba.uml.UMLCollabStat</Class>
51     <Class>de.uni_paderborn.fujaba.uml.UMLStoryActivity</Class>
52     <Class>de.uni_paderborn.fujaba.uml.UMLStoryPattern</Class>
53     <Class>de.uni_paderborn.fujaba.uml.UMLObject</Class>
54     <Class>de.uni_paderborn.fujaba.uml.UMLLink</Class>
55 </UMLDiagram>
56...
```

Auszug aus dem in XML codierten Filter für Klassendiagramme

Die XML-Datei ist in drei Bereiche gegliedert. Zuerst die DTD, die die zulässigen Konstrukte festlegt. Anschließend folgt der “General Excluded” Abschnitt. Er beschreibt die Objekte die diagrammübergreifend gefiltert werden. Der letzte Teilbereich “UMLDiagrams Excluded Classes” beschreibt die für Klassendiagramme spezifischen Filterobjekte.

Der Algorithmus lädt auf Basis des zu speichernden Diagramms den dementsprechenden Filter. In diesem Fall den Filter für Klassendiagramme. Anschließend wird der abstrakte Syntaxgraph durchlaufen. Jedes Objekt des ASG wird dabei nach folgendem Muster geprüft:

Zuerst wird das aktuelle Objekt gegen den Filter geprüft. Ist die Prüfung erfolgreich, ist sichergestellt, dass das Objekt einen zulässigen Objekttyp besitzt. Daraufhin prüft der Algorithmus in einem zweiten Schritt ob das Objekt Bestandteil des zu speichernden Diagramms ist und ob es Referenzen auf andere Diagramme kapselt. Werden Referenzen auf andere Diagramme gefunden so werden diese zwischengespeichert um dann in einem weiteren Durchlauf des Algorithmus diese Diagramme unter Anwendung der dementsprechenden Filter zu speichern.

Nachdem alle Objekte des ASG auf diese Weise verarbeitet wurden, enthält die Exportdatei alle Objekte des zu speichernden Diagrammes und gegebenenfalls, auf Wunsch des Benutzers, die referenzierten Diagramme.

Zusätzlich zur Identifikation und Abgrenzung der Diagramme, setzt das Filterkonzept noch einen Aspekt der Konsistenzprüfung um. Dabei werden während des Speichervorgangs unzulässige Referenzen innerhalb der gespeicherten Objekte herausgefiltert. Unzulässige Referenzen sind in diesem Zusammenhang Referenzen auf

Objekte die nicht Bestandteil meines Diagramms sind und daher auch nicht gespeichert werden.

Während des Speichervorgangs protokolliert das Filterkonzept mit, welche Objekte des ASG nicht gespeichert wurden und überprüft anschließend den StringBuffer nach Referenzen auf diese Objekte. Da diese referenzierten Objekte nicht gespeichert werden und im Umkehrschluss auch nicht wieder geladen werden, müssen die Referenzen herausgefiltert werden um Inkonsistenzen, im aus dem Ladevorgang resultierenden ASG, auszuschließen.

2.4.2 Mergekonzept

Das Mergekonzept ist Bestandteil des Ladealgorithmus und setzt einen weiteren Aspekt der Konsistenzerhaltung um. Das Mergekonzept hat die Funktion ausgezeichnete UML-Objekte des ASG und des zu ladenden Diagramms zu identifizieren und zu verarbeiten. Diese ausgezeichneten UML-Objekte, nachfolgend MergePoints genannt, haben die Eigenschaft das sie generell aus Konsistenzgründen nur einmal im UML-Modell enthalten sein dürfen.

Um ein Beispiel für einen solchen MergePoint zu nennen: Die Java-Grundtypen boolean, Integer oder float werden im abstrakten Syntaxgraphen durch Objekte der Klasse *UMLBaseType* repräsentiert. Diese Grundtypen und somit die dementsprechenden *UMLBaseType*-Objekte dürfen aufgrund der Spezifikation nur einmal im UML-Modell enthalten sein und damit auch nur einmal im ASG auftreten.

Das Persistenzkonzept speichert diese MergePoints beim Schreiben einer Projektdatei mit ab. Dies ist auch notwendig, da ansonsten die Konsistenz des Projektes verloren ginge. Beim Speichern eines Diagramms ist es aus den selben Gründen wie beim Projekt notwendig das die MergePoints mit abgespeichert werden. Was allerdings beim Laden eines Projektes unproblematisch ist, verursacht beim Laden des Diagramms Inkonsistenzen. Erzeugt das Persistenzkonzept aus der Projektdatei einen neuen abstrakten Syntaxgraphen, so ist automatisch gewährleistet, das die MergePoints jeweils nur einmal im UML-Modell auftreten. Im Gegensatz dazu wird beim Laden eines Diagramms kein neuer ASG erzeugt. Somit kollidieren die MergePoints des vorhandenen ASG mit den MergePoints in der zu ladenden GXL-Datei.

Aufgabe des Mergekonzeptes ist es diese Kollisionen zu erkennen und geeignet aufzulösen. Dabei unterscheidet das Konzept folgende zwei Szenarien:

- Es existiert schon ein äquivalenter MergePoint im ASG. Dann bildet das Mer-

gekonzepnt den zu ladenden auf den vorhandenen MergePoint ab.

- Es existiert kein äquivalenter MergePoint im ASG. Dann wird das Objekt normal erzeugt und in den ASG eingebunden.

2.4.3 Gesamtüberlick

Dieser Abschnitt fasst noch einmal die in den Abschnitten 2.3.1 und 2.3.2 einzeln vorgestellten Konzepte zusammen und stellt sie im Gesamtkontext vor.

Der Speicheralgorithmus für Diagramme basiert auf dem Speicherverfahren des Persistenzkonzeptes. Zusätzlich nutzt der Algorithmus das Filterkonzept um Inkonsistenzen beim Speichern des Diagrammes auszuschließen und um die einzelnen Komponenten des Diagrammes zu identifizieren.

Der Ladealgorithmus basiert äquivalent zum Speicherverfahren auf dem Ladealgorithmus des Persistenzkonzeptes. Zu Beginn des Algorithmus wird anhand einer Typprüfung verifiziert ob das zu ladende (Teil)-Diagramm und das Zieldiagramm vom gleichen Typ sind. Anschließend gewährleistet das Mergekonzept das der vorhandene ASG nach dem Ladevorgang konsistent ist.

Im nachfolgenden Abschnitt wird die technische Umsetzung des Lade- sowie des Speicheralgorithmus erläutert. Dabei stellt das Kapitel detailliert den Ablauf der einzelnen Algorithmen, sowie die jeweils implementierten Java Klassen vor.

2.5 Technische Realisierung

Dieses Kapitel erläutert die Umsetzung der in Abschnitt 2.4 vorgestellten Konzepte. Dabei wird detailliert der Ablauf der Lade- und Speicheralgorithmen sowie die implementierten Java Klassen und deren Funktion dargestellt.

Der Abschnitt 2.5.1 stellt zunächst die Klassen und den Ablauf des Ladealgorithmus vor. Darauf aufbauend wird die zentrale Implementierung des Algorithmus, die Klasse *GXLImport*, erläutert. Abschließend werden die Klassen, die das Mergekonzept umsetzen beschrieben.

Der Abschnitt 2.5.2 ist äquivalent aufgebaut. Zuerst werden die Klassen und der Ablauf des Speicheralgorithmus erläutert. Anschließend wird vertiefend der Ablauf des Filterkonzeptes dargestellt, um dann die wichtigste Klasse des Speicheralgorithmus,

die Klasse *GXLFilter* vorzustellen. Abschließend werden die Dateien zum Laden der Filter beschrieben.

2.5.1 Ladealgorithmus

Der Ladealgorithmus ist in den Klassen *GXLImport*, *Creator*, *AbstractMerger* und *UMLClassDiagramMerger* implementiert. Die Hauptfunktionalität des Ladealgorithmus kapselt die Klasse *GXLImport*. In den drei weiteren Klassen ist das Mergekonzept implementiert.

Die Grafik 2.3 zeigt den Ablauf des Algorithmus beim Laden eines UML-Klassendiagramms.

Nachdem der Benutzer die zu ladende GXL-Datei ausgewählt hat, wird im ersten Schritt die Datei eingelesen und ein DOM-Baum aus den GXL-Konstrukten erzeugt. Die DOM-Baum Repräsentation bietet die Möglichkeit des schnellen und direkten Zugriffs, sowie die gezielte Suche nach "GXL-Informationen". Im nächsten Schritt nimmt der Algorithmus eine Typprüfung des Diagramms vor um anschließend auf Basis der gewonnenen Informationen den entsprechenden "Merger" zu laden.

Für jeden Diagrammtyp nutzt der Ladealgorithmus einen entsprechenden "Merger". Die Aufgabe des Merger's ist es, die MergePoints im DOM-Baum zu identifizieren und geeignet zu verarbeiten.

Nachdem der entsprechende Merger instanziiert wurde, werden die UML-Objekte des Diagrammes aus dem DOM-Baum eingelesen, erzeugt und anhand des Mergers gegen die vorhandenen MergePoints validiert. Um die jeweiligen UML-Objekte zu erzeugen und ihre Attribute zu setzen wird der Reflection Mechanismus des Java Development Kit genutzt. Dabei werden die Objekte anhand *Class.forName(String class)* instanziiert und die Werte der Attribute per *invoke(Object obj, Object[] args)* gesetzt.

Abschließend werden die so erzeugten UML-Objekte im abstrakten Syntaxgraph unter dem aktuell selektierten UML-Diagramm eingehängt und das geladene (Teil-)Diagramm ist Bestandteil des aktuellen Projektes.

Zentrale Implementierung des Ladealgorithmus ist die Java-Klasse *GXLImport*. Sie kapselt die Typprüfung der Diagramme und beinhaltet auch die Algorithmen zur Instanziierung der UML-Objekte und setzen der Attributwerte (siehe dazu auch Grafik 2.3). Zusätzlich ist auch der Parse-Algorithmus des im Rahmen der Projektgruppe "Entwurfsunterstützung von verteilten Multimedia Anwendungen mit Hilfe von Design Pattern" entwickelten Pattern-Managers in dieser Klasse implementiert.

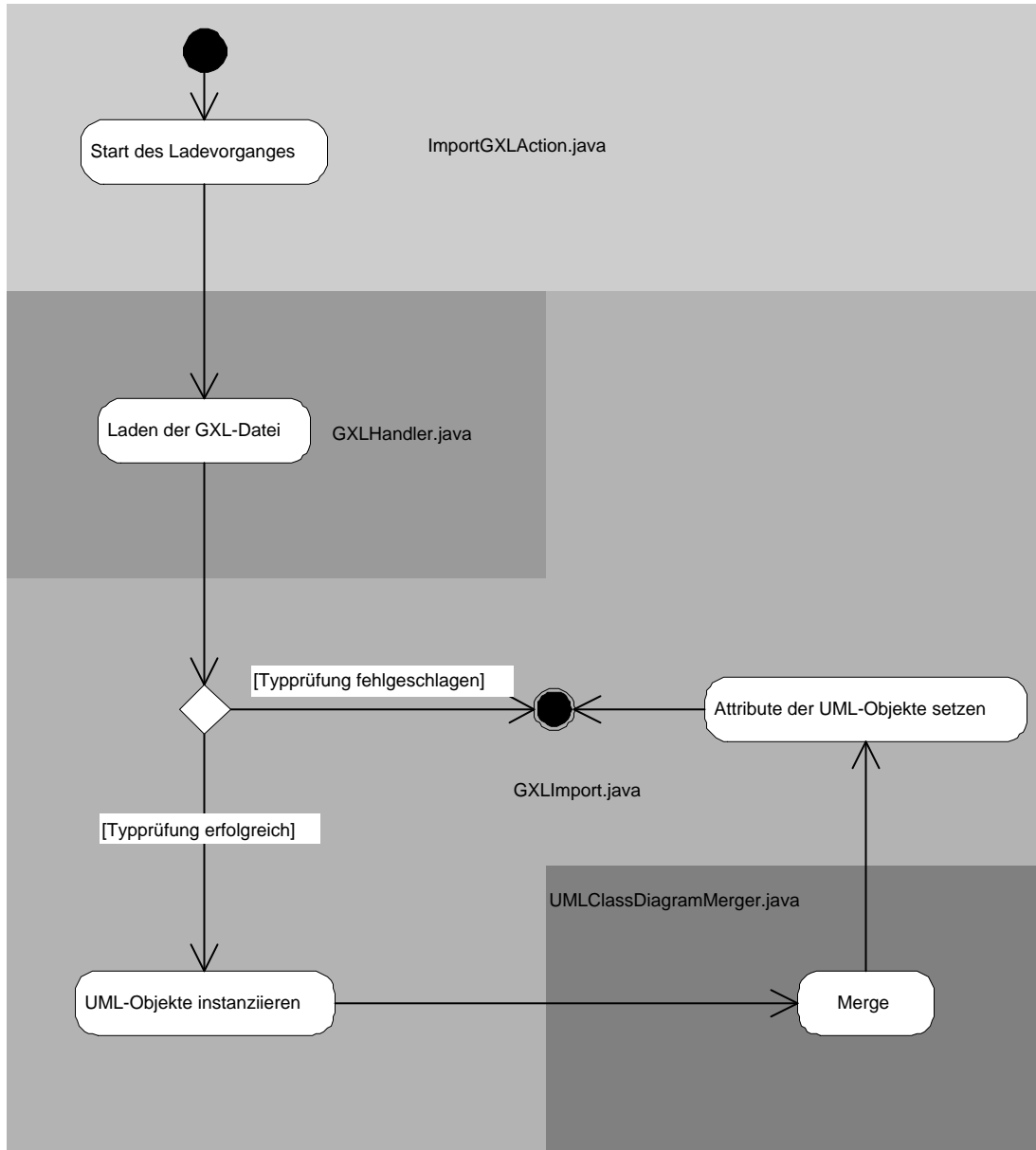


Abbildung 2.3: Ladevorgang eines UML-Klassendiagramms

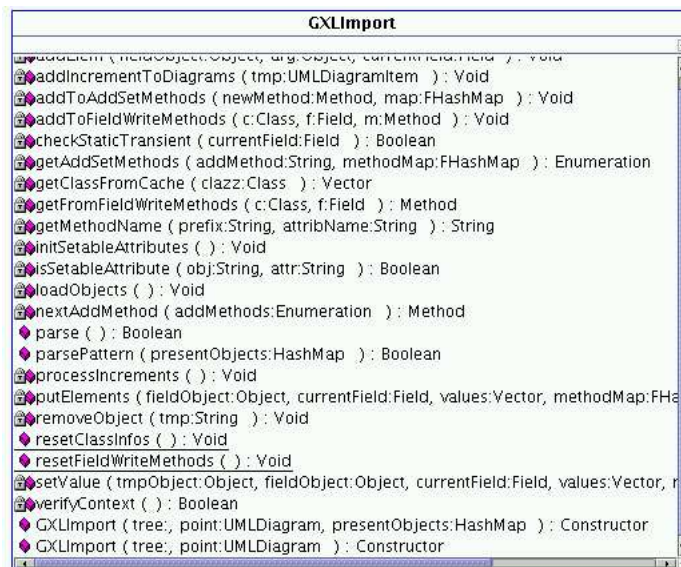


Abbildung 2.4: Methoden der Klasse *GXLImport*

Die Grafik 2.4 zeigt die wichtigsten Methoden der Klasse *GXLImport*.

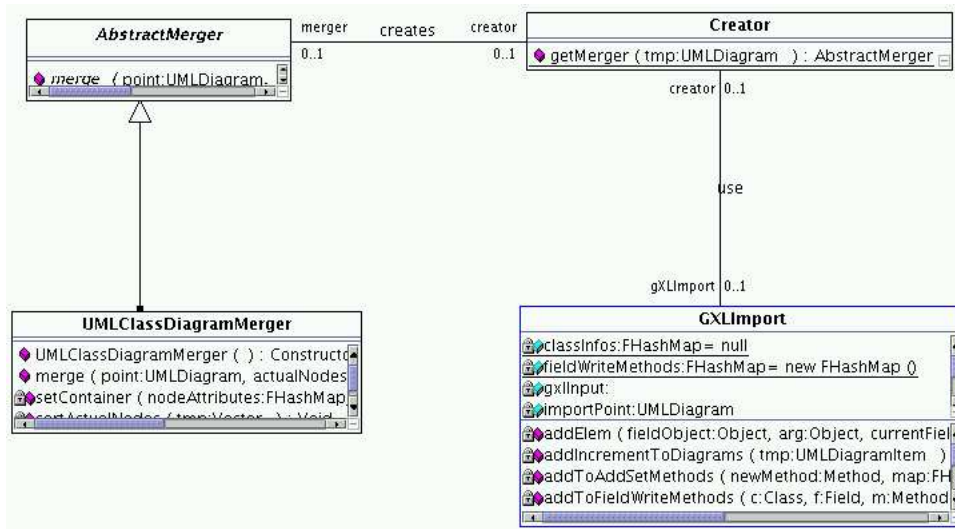
GXLImport stellt zwei Konstruktoren zur Verfügung: *public GXLImport (FXML-Tree tree, UMLDiagram point)* wird vom Ladealgorithmus genutzt und benötigt die Parameter *tree*, der DOM-Baum des zu ladenden (Teil)-Diagrammes, und *point*, das UML-Diagramm in dem das (Teil)-Diagramm eingefügt werden soll.

Der zweite Konstruktor *public GXLImport (FXMLTree tree, UMLDiagram point, HashMap presentObjects)* wird ausschliesslich vom Pattern-Manager verwendet und benötigt den zusätzlichen Parameter *presentObjects*, welcher die mit dem Pattern zu mergenden Objekte des aktuellen Diagramms kapselt.

Der Konstruktor stösst die in der Methode *verifyContext()* implementierte Typprüfung an und instanziiert dann auf Basis der übergebenen Parameter, den benötigten Merger. Wird die Prüfung erfolgreich abgeschlossen, so werden alle notwendigen UML-Objekte auf Basis des Dom-Baum erzeugt und an den Merger übergeben.

Nach Abschluss des Merge-Vorgangs wird der Parse-Vorgang gestartet. Dieser ist implementiert in der Methode *parse()* und setzt die Attributwerte aller neu erzeugten UML-Objekte. Der Algorithmus ist weitgehend identisch zu dem in der Klasse *BasicIncrement* implementierten Mechanismus des Persistenzkonzeptes.

Der Parse-Algorithmus unterscheidet zwischen Containerattributen und “Einzelwert”-Attributen und führt die dementsprechenden Zugriffsmethoden

Abbildung 2.5: Statische Verknüpfung *GXLImport*, *UMLClassDiagramMerger*

anhand `invoke(Object obj, Object[] args)` aus der Klasse *Method* aus. Nach erfolgreichem setzen des Attributes wird die Zugriffsmethode, zur Laufzeitoptimierung, in einer zusätzlichen Datenstruktur gecacht.

Nach dem alle Eigenschaften der neuen UML-Objekte gesetzt sind, werden diese dem UML-Diagramm hinzugefügt und der Ladealgorithmus terminiert.

Das Mergekonzept für Klassendiagramme ist in den drei Klassen *Creator*, *AbstractMerger* und *UMLClassDiagramMerger* implementiert. Die Grafik 2.5 zeigt die statische Struktur der Implementierung.

Während des Ladevorgangs instanziiert die Klasse *GXLImport* ein Objekt der Klasse *Creator*. Einzige Funktion der Klasse *Creator* ist die Identifikation des geeigneten Mergers für das zu ladende (Teil)-Diagramm. Die Klasse *GXLImport* arbeitet auf einer Variablen des Typs *AbstractMerger*, der abstrakten Oberklasse aller Merger. Sie spezifiziert die Methode `public abstract void merge(UMLDiagram point, Vector actualNodes, HashMap objects)` die jeder Merger zur Verfügung stellen muss. Jede Spezialisierung der Klasse *AbstractMerger* implementiert diese Methode.

Die Grafik 2.5 zeigt den speziellen Merger *UMLClassDiagramMerger*, der genutzt wird um die MergePoints der Klassendiagramme zu verarbeiten.

2.5.2 Speicheralgorithmus

Der Speicheralgorithmus ist in den Klassen *GXLFilter* und *GXLExcludedLoader* implementiert. Zentrale Implementierung des Speicheralgorithmus ist die Klasse *GXLFilter*, die das Filterkonzept umsetzt. In der Klasse *GXLExcludedLoader* ist der Algorithmus zum Laden der einzelnen Filter implementiert.

Den Ablauf des Speicheralgorithmus zeigt die Grafik 2.6.

Nach dem der Benutzer das zu exportierende Diagramm ausgewählt und einen Dateinamen festgelegt hat, wird auf Basis dieser Informationen der Filter geladen. Dieser Mechanismus ist in den Klassen *GXLExcludedLoader* und *GXLFilter* implementiert und wurde im Abschnitt 2.4.1 vorgestellt. Anschließend wird der eigentliche Speichervorgang anhand der überlagerten Methode *writeClassToStringBuffer()* in der Klasse *BasicIncrement* angestoßen. Um auf Basis dieses bereits vorhandenen Algorithmus einzelne Diagramme speichern zu können, musste die Methode *writeClassToStringBuffer(StringBuffer data, FTreeSet allIncrements)* überlagert werden. Dies ist notwendig um die zu exportierenden Daten in den StringBuffer, unter Anwendung der Filter, schreiben zu können.

Beginnend beim UMLDiagram-Objekt des Diagramms werden alle erreichbaren Objekte, die der Filter akzeptiert, mittels Java Reflection Mechanismus in den StringBuffer geschrieben. Alle erreichten Objekte die herausgefiltert wurden, werden in einer zusätzlichen Datenstruktur gespeichert. Nach dem Schreibvorgang, werden alle Referenzen auf die herausgefilterten Objekte gelöscht und der bereinigte StringBuffer der Klasse *FileSaver* übergeben. Diese Klasse führt die in Kapitel 3 erläuterten Transformationen durch, um die Daten in das gewünschte Exportformat zu überführen.

Die Grafik 2.6 zeigt nicht die Erweiterungen in der Klasse *BasicIncrement*, da sie lediglich ein Objekt der Klasse *GXLFilter* dem Speicheralgorithmus zu Verfügung stellt. Die Anwendung der Filter ist in der Klasse *GXLFilter* implementiert. Die Grafik 2.7 zeigt die wichtigsten Methoden der Klasse *GXLFilter*.

Dem Konstruktor wird als Parameter das selektierte Diagramm, oder die selektierten UML-Objekte des zu speichernden Teildiagrammes übergeben. Auf Basis der Parameter bestimmt der Konstruktor den Diagrammtyp und lädt über die Methode *loadView(String type)* den benötigten Filter. Bevor die Eigenschaften eines UML-Objektes in den StringBuffer geschrieben werden, wird anhand der Methode *public boolean validateObject (BasicIncrement obj) throws IOException* das Objekt gegen den Filter geprüft.

Nach dem alle Objekte in den StringBuffer geschrieben wurden, werden Referenzen

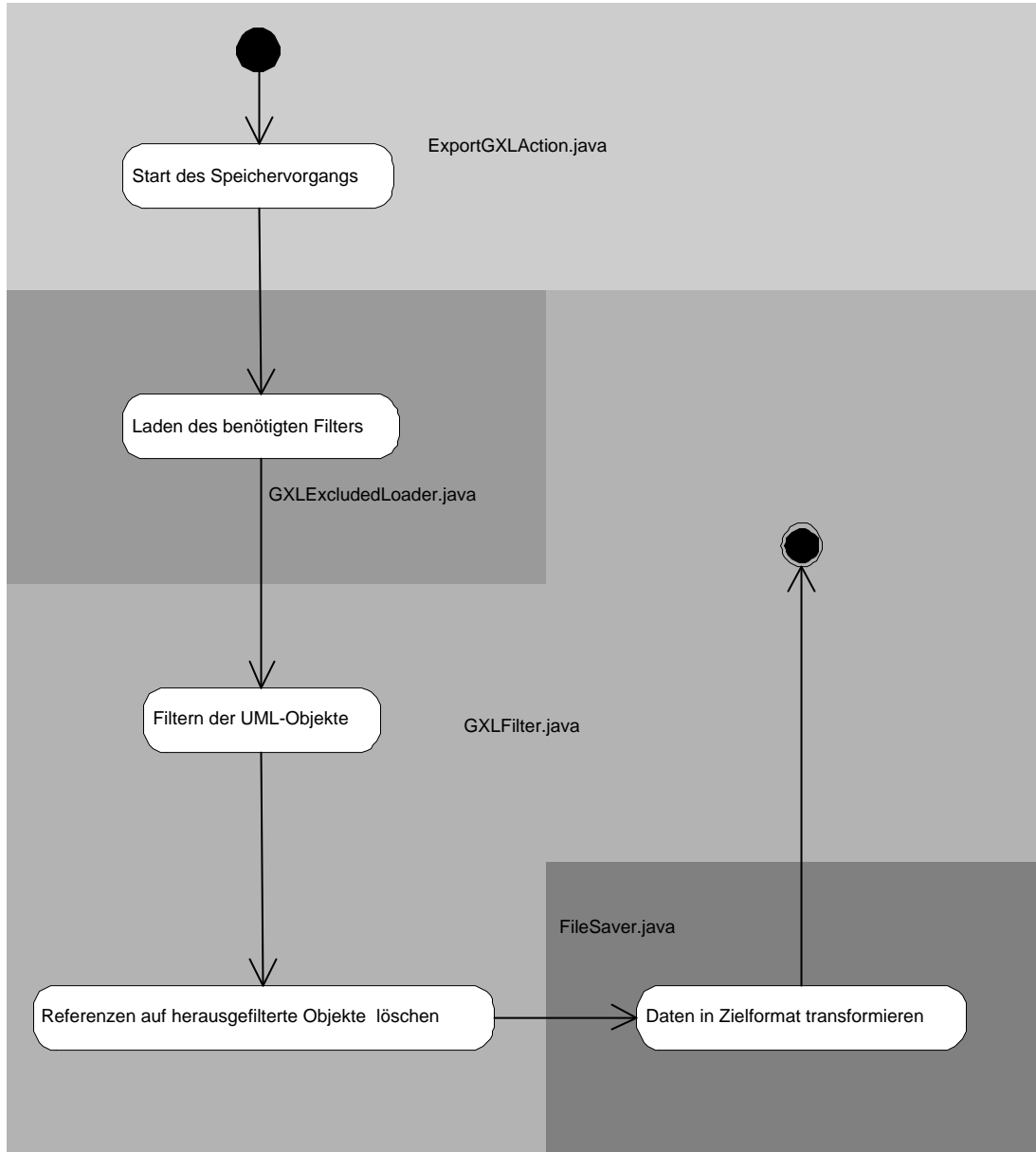
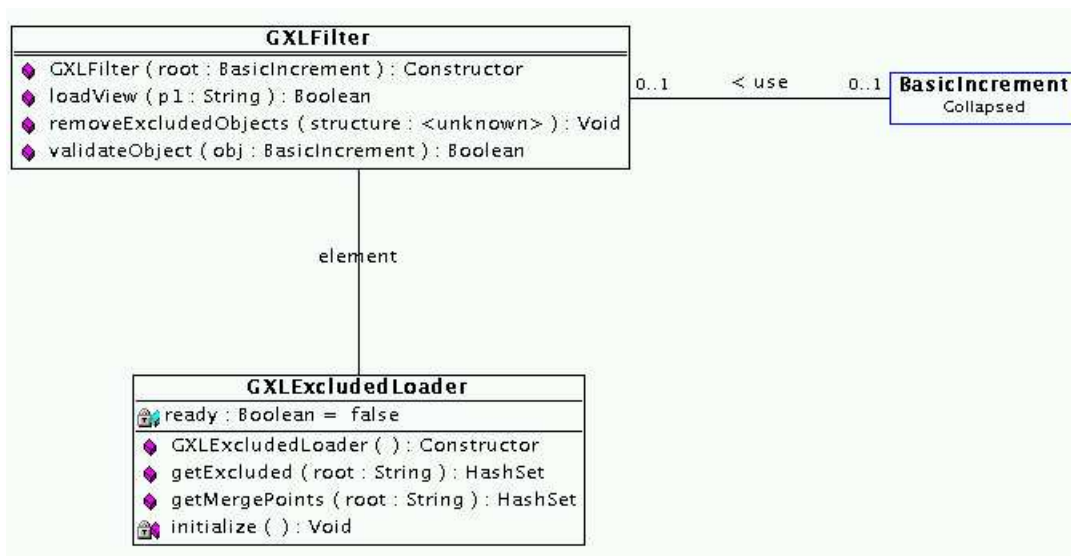


Abbildung 2.6: Ablauf Speichervorgang

Abbildung 2.7: Die Klasse *GXLFilter*

auf herausgefilterte Objekte durch die Methode *removeExcludedObjects(StringBuffer buffer)* entfernt und der bereinigte *StringBuffer* zur weiteren Transformation an die Klasse *FileSaver* übergeben. Die weiteren Schritte bis zur fertigen Exportdatei werden im Kapitel “Datenaustausch” erläutert.

Die Grafik 2.8 zeigt in einem Ausschnitt des Ladealgorithmus, die Funktion des Filterkonzeptes.

Die Klasse *GXLFilter* identifiziert beim Start des Speicheralgorithmus den Diagrammtyp des zu exportierenden Diagramms und lädt den benötigten Filter. Diese Aufgabe wird an ein Objekt der Klasse *GXLExcludedLoader* übertragen, welches den in XML codierten Filter lädt. Anschließend wird der eigentliche Speichervorgang eingeleitet. Ausgehend vom *UMLDiagram*-Objekt wird jedes erreichbare Objekt, was vom Filter akzeptiert wird, gespeichert. Zusätzlich werden nun noch bestimmte *UML*-Objekte meines Diagrammes auf Referenzen zu anderen Diagrammen geprüft. Ein Beispiel: Der Benutzer möchte ein Klassendiagramm exportieren. In diesem Klassendiagramm existieren Methoden, die anhand weiterer *Story*-Diagramme, näher spezifiziert sind.

Der Speicheralgorithmus startet nun mit dem *UMLDiagram*-Objekt des Klassendiagramms. Im Verlauf des Speichervorgangs erreicht der Algorithmus die Methoden des Klassendiagramms, repräsentiert durch *UMLMethod*-Objekte. Der Algorithmus prüft nun, ob diese Objekte Referenzen auf andere Diagramme enthalten. In unserem Beispiel sind dies die Methoden, die durch *Story*-Diagramme näher spezifiziert sind. Diese Referenzen werden in einer zusätzlichen Datenstruktur zwischengespeichert. Nachdem das Klassendiagramm gespeichert wurde prüft der Algorithmus ob

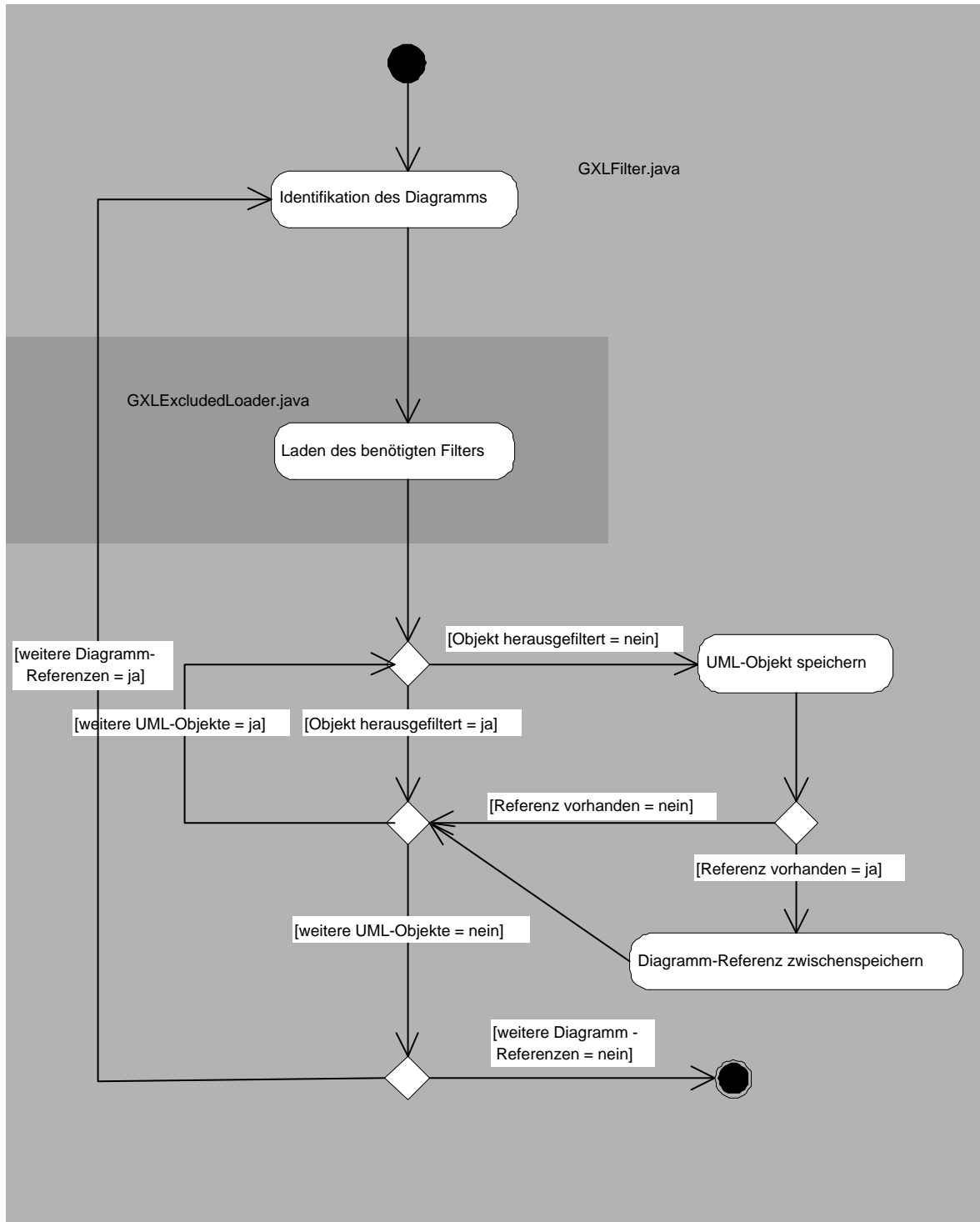


Abbildung 2.8: Funktionsweise Filterkonzept

2 Lade- und Speicheralgorithmus

die Datenstruktur weitere Referenzen enthält. Ist dies der Fall so werden anhand der Referenzen die dementsprechenden Diagramme ebenfalls gespeichert. Siehe dazu auch die Grafik 2.8. Die resultierende Exportdatei enthält somit das Klassendiagramm, sowie die Story-Diagramme zur Spezifikation der Methoden.

3 Datenaustausch

(Philipp Hoven)

Gegenstand dieses Kapitels ist der Austausch der Daten, also der Im- und Export der Daten aus Fujaba heraus. Sollen Daten exportiert werden, so umfasst der Austausch alle Aktionen, die nach dem Herauslösen der Daten aus dem abstrakten Syntaxgraphen Fujaba´s vorgenommen werden. Entsprechend sind beim Datenimport alle Aktionen, die vor der Integration der Daten in den abstrakten Syntaxgraphen stattfinden, Teil des Austausches.

Zunächst geht der Abschnitt *Probleme und Lösungen* auf die gegenwärtige Situation beim Datenexport ein. Er beschreibt die Probleme, die bei der Einführung eines neuen Im- und Exportmechanismus zu bewältigen sind. Abschließend wird ein Konzept zur Lösung der Probleme vorgestellt. Die verschiedenen Formate, die beim Datenaustausch Anwendung finden, werden im Abschnitt *Datenformate* behandelt. Die Umformung zwischen diesen Formaten ist Gegenstand des Abschnitts 3.4 *XML Datentransformation*. Im Abschnitt *Technische Realisierung* wird die Umsetzung des zuvor beschriebenen Konzeptes und die Implementierung erläutert.

3.1 Probleme und Lösungen

Sollen Daten exportiert werden, so liegen diese nach den in Kapitel 2 *Lade- und Speicheralgorithmus* beschriebenen Schritten im fujabaeigenen FPR-Format vor. Da als Ausgabeformate GXL beziehungsweise XMI angestrebt werden, müssen die Daten transformiert werden. Ebenso muss für die Gegenrichtung die Transformation von GXL beziehungsweise XMI in ein für den neuen Lademechanismus geeignetes Format, bewältigt werden.

An den zu erstellenden Transformationsprozess lassen sich somit folgende Anforderungen stellen:

3 Datenaustausch

- Der Lade- und Speichermechanismus sollte möglichst unabhängig vom Rest des Datenaustausches bestehen.
- Weitere Formate sollten sich in den Prozess einbinden lassen.
- Die Umsetzung sollte spätere Änderungen zulassen.
- Die angestrebten Austauschformate GXL und XMI sind XML-Dialekte. Das Konzept sollte dies berücksichtigen.

Durch die Trennung von Lade- und Speichermechanismus und Transformation der Daten zu oder von einem bestimmten Format, entfällt die Notwendigkeit alle Formate direkt durch den Lade- und Speichermechanismus zu unterstützen. Um diese Trennung zu realisieren, wurde das Format FujabaXML eingeführt (für Details siehe Abschnitt 3.2.5).

Der Lade- und Speichermechanismus benötigt somit nur zwei Transformationen (je eine für Im- und Export), die unabhängig vom Austauschformat immer gleich sind.

Da FujabaXML eng an FPR angelehnt ist, gestaltet sich die Transformation nach FujabaXML recht einfach. In der Gegenrichtung musste der Lademechanismus angepasst werden, um FujabaXML einlesen zu können. Da das ursprüngliche Format FPR keinen Import von Teildiagrammen unterstützt und neu gestaltet werden musste, stellt diese Anpassung keinen Mehraufwand dar.

Damit Fujaba ein neues Format unterstützt, reicht es somit aus, für dieses Format eine Transformation von und nach FujabaXML zu definieren. Diese Transformation muss nicht direkt umgesetzt werden, sondern kann indirekt über ein bereits unterstütztes Format geschehen. Im Fall von XMI wurde eine indirekte Transformation über GXL gewählt, da bereits Stylesheets zur Transformation von XMI nach GXL existieren. Details zu dieser Transformation sind im Kapitel 3.4.3 *Weitere Transformationen* aufgeführt.

Sollten sich später einzelne Formate oder der Lade- und Speichermechanismus ändern, so genügt es für die neue Komponente eine Transformation von und nach FujabaXML zu definieren.

Da die angestrebten Austauschformate GXL und XMI beide XML-Dialekte darstellen, bietet sich die Verwendung von bestehenden Transformationskonzepten für XML wie XSLT (siehe Abschnitt 3.3) an. Solange die Daten in einem XML-Format vorliegen, können zur Transformation Stylesheets eingesetzt werden.

Die Grafik 3.1 stellt eine Übersicht des Transformationsprozesses dar. Dabei lassen sich innerhalb des Transformationsprozesses Im- und Export in jeweils zwei Phasen unterteilen.

3.2 Datenformate

Dieser Abschnitt behandelt die verschiedenen Formate, die bei der Umsetzung der zuvor vorgestellten Konzepte angewandt wurden.

Zunächst wird die Extensible Markup Language, welche die Grundlage für andere Formate wie FujabaXML, GXL oder XMI bildet, vorgestellt und einige ihrer Konzepte erläutert. Anschließend wird der Aufbau der Datenformate der Graph Exchange Language und XMI näher beschrieben. Danach wird das aktuelle Standardformat von Fujaba - FPR - vorgestellt und die daran durchgeführten Änderungen erläutert. Abschließend wird das neu eingeführte Format FujabaXML behandelt. Es dient als Zwischenformat, um den Transformationsprozess transparenter und flexibler zu gestalten.

3.2.1 XML

Die Extensible Markup Language [XML] ist eine Auszeichnungssprache zur Strukturierung von typisierten Daten, welche von SGML abgeleitet wurde. Im Februar 1998 wurde XML in der Version 1.0 vom W3C als Empfehlung verabschiedet. XML ermöglicht den anwendungsübergreifenden Austausch von Daten.

Es existieren zahlreiche spezialisierte XML-Formate aus vielen unterschiedlichen Anwendungsgebieten, welche die Sprache an ihre jeweiligen Anforderungen angepasst haben.

Ein XML-Dokument enthält im wesentlichen ausgezeichnete Daten. Dazu werden Element und Attribute verwendet, die in einer baumartigen Struktur angeordnet werden. In XML gibt es zwei verschiedene Arten der Korrektheit: Wohlgeformt (well-formed) und gültig (valid). Die Bedingung der Wohlgeformtheit ist erfüllt, wenn das Dokument syntaktisch korrekt ist.

Ein gültiges XML-Dokument benötigt eine vorgegebene Struktur, gegen die es validiert werden kann. Diese Struktur liegt in Form einer *Document Type Definition*, kurz DTD, oder einer *XML Schema Definition*, kurz XSD, vor.

DTDs beschreiben ausschließlich die Baumstruktur mit Hilfe von Elementen und deren Attribute. Dabei nutzten sie eine eigene Syntax, die keine Unterstützung von verschiedenen Datentypen bietet. Lediglich für Attribute werden einige Typen angeboten:

- **NMTOKEN** - Dem Attribut muss ein Name zugeordnet werden, der den

Namenskonventionen von XML folgt.

- **CDATA** - Dem Attribut lässt sich jede beliebige Zeichenkette zuordnen.
- **ID** - Über IDs lassen sich Elemente eindeutig identifiziert. Dabei darf jede ID nur einmal auftreten.
- **IDREF** - ID-Referenz, die auf Instanzen von Elementen verweist.
- **ENTITY** - Der Attributwert muss der Name eines Entitys sein.
- $(name_1|..|name_n)$ - Dieser Aufzählungstyp besteht aus einer Namensliste mit durch Balken getrennten Werten.

Liste der Attributtypen einer DTD

Zusätzlich können für Attribute Vorgabewerte deklariert werden. Ist dem Vorgabewert ein *#FIXED* vorangestellt, so kann der Attributwert nicht verändert werden. Durch das Anhängen von *#REQUIRED* wird die Angabe eines Attributes erzwungen. Mit *#IMPLIED* wird ein optionales Attribut gekennzeichnet.

Für die Erweiterung existiert bei DTDs nur das rudimentäre Konzept der *Entities*, die ausschließlich einfache Zeichenersetzung anbieten. Solche Erweiterungen sind beispielsweise in der DTD für GXL vorhanden, um selbstdefinierte Elemente in die Graphen zu integrieren.

Da die Syntax einer DTD aus nur wenigen verschiedenen Elementen besteht, sind sie übersichtlicher als eine gleichwertige XSD. Wenn die einfachen Konstrukte einer DTD genügen und beispielsweise keine Datentypen benötigt werden, so ist eine DTD ausreichend.

Im Gegensatz dazu können mit XSD sowohl eigene Datentypen definiert werden als auch eigene Erweiterungen eingebracht werden. XSDs nutzen zur Definition XML-Syntax. Sie stellen also selbst gültige XML-Dokumente dar. XSDs sind somit weit mächtiger als DTDs, jedoch sind ihr Aufbau weitaus komplexer. Sie unterstützen eine breite Palette an Datentypen, auch Boolesche-, Integer- beziehungsweise Dezimalwerte und bieten darüber hinaus auch die Möglichkeit weitere Datentypen zu definieren.

Auch wenn die zu definierende Struktur eher klein ist, so kann die XSD durch ihre Vielzahl an Konzepten und komplexe Syntax unübersichtlich und aufwändig werden.

Weitere Details zu DTD und XML Schemata sind in [HoLi01] aufgeführt.

3.2.2 XML Metadata Interchange

Die von der OMG verabschiedete Spezifikation XML Metadata Interchange [XMI] definiert ein XML-basiertes Austauschformat für anhand der UML modellierten Metadaten. XMI setzt sich aus den drei Standards XML, UML und Meta Object Facility zusammen. Dabei stellt [MOF] einen Standard der OMG für Metamodellierung und Metadaten Repositories dar.

Die Ziele bei der Konzeption von XMI waren unter anderem die Schaffung eines standardisierten Industrieformates für den Austausch von (Meta)-Modellen sowie die Schaffung eines Formats das in eine Vielzahl verschiedener Modelle übertragen werden kann. Das Erreichen dieser Ziele erlaubt den Austausch von UML-Spezifikationen zwischen verschiedenen UML-Tools.

Ein XMI-Dokument verfügt über einen Header, der Informationen über den Ursprung der Daten enthält. Das Tag `<XMI.content>` schließt alle Objekte, die Bestandteil des Diagramms sind, ein. Diese sind durch `<UML:Namespace.ownedElement>`-Tags gekapselt. Klassen (`<UML:Class>`) beinhalten ihre Attribute und Methoden in `<UML:Classifier.feature>`-Tags. Dabei werden die Methoden - hier Operationen genannt - aufgeteilt in die eigentliche Operation (Zeile 26), die unter anderem den Namen und die Sichtbarkeit enthält, und die Signatur (Zeile 30). Letztere wird innerhalb des Tags `<UML:BehavioralFeature.parameter>` aufgelistet. Dabei gilt der Rückgabewert als Parameter (Zeile 31) vom Typ *return*.

```

1 <!DOCTYPE XMI SYSTEM "http://www.fujaba.de/xmi11-uml13.dtd">
2 <XMI xmlns:UML="http://www.omg.org/uml/1.3"
      xmlns:xlink="http://www.w3.org/1999/xlink">
3 <XMI.header>
4 <XMI.documentation>
5 <XMI.exporter>Fujaba</XMI.exporter>
6 <XMI.exporterVersion>3.0</XMI.exporterVersion>
7 </XMI.documentation>
8 <XMI.metamodel xmi.name="UML" xmi.version="1.3"/>
9 </XMI.header>
10 <XMI.content>
11 <UML:Model name="Main" xmi.id="id17">
12 <UML:Namespace.ownedElement>
13 <UML:Package name="RootPackage"
      namespace="id17" xmi.id="id4"/>
14 </UML:Namespace.ownedElement>
15 <UML:Namespace.ownedElement>
16 <UML:DataType name="Void" namespace="id17" xmi.id="id39"/>

```

```

17     <UML:DataType name="Boolean" namespace="id17" xmi.id="id30"/>
18         ...
19     </UML:Namespace.ownedElement>
20     <UML:Namespace.ownedElement>
21         <UML:Class isAbstract="false" name="Main"
                namespace="id4" visibility="private"
                xmi.id="id3">
22             <UML:Classifier.feature>
23                 <UML:Attribute changeable="none" name="ready"
                        owner="id3" type="id30"
                        visibility="public" xmi.id="id10"/>
24             </UML:Classifier.feature>
25             <UML:Classifier.feature>
26                 <UML:Operation name="showResult" owner="id3"
                        visibility="public" xmi.id="id14">
27                     <UML:Operation.method>
28                         <UML:Method owner="id3" specification="id14"/>
29                     </UML:Operation.method>
30                     <UML:BehavioralFeature.parameter>
31                         <UML:Parameter kind="return" type="id39"/>
32                     </UML:BehavioralFeature.parameter>
33                 </UML:Operation>
34         ...
35     </UML:Classifier.feature>
36 </UML:Class>
37 </UML:Namespace.ownedElement>
38 </UML:Model>
39 </XMI.content>
40 </XMI>

```

Auszug aus einem XMI-Dokument

3.2.3 GXL

GXL wurde im Januar 2001 als Format für den Austausch zwischen Software-Entwicklungsumgebungen auf dem Dagstuhl-Seminar “Interoperability or Reengineering Tools” vorgestellt. Die wesentliche Arbeit bei der Entwicklung leisteten Richard C. Holt (University of Waterloo), Andy Schürr (Bundeswehr Universität München) und Andreas Winter (Universität Koblenz-Landau). GXL hat viele Vorgängerformate. Die wichtigsten davon sind GRAPh eXchange format (GraX), Tuple Attribute Language (TA),PROGRES (Bw Universität, München), Relation

3 Datenaustausch

Partition Algebra (RPA, Philips Research, Eindhoven) und Rigi Standard Format (RSF, University of Victoria). Da GXL viele Gemeinsamkeiten von diesen Formaten übernommen hat, läßt es sich als Verallgemeinerung derselben betrachten.

Als XML-Format ist GXL durch eine DTD definiert. Der nachfolgende Auszug zeigt den Aufbau dieser DTD in Version 1.0.

```
1 <!ENTITY % val "bool | int | string | set">
2 <!ELEMENT gxl (graph*) >
3 <!ELEMENT graph (attr* , (node | edge)* )>
4 <!ATTLIST graph
5   id          ID          #REQUIRED
6 >
7 <!ELEMENT node (attr*, graph*) >
8 <!ATTLIST node
9   id          ID          #REQUIRED
10 >
11 <!ELEMENT edge (attr*, graph*) >
12 <!ATTLIST edge
13   id          ID          #IMPLIED
14   from        IDREF      #REQUIRED
15   to          IDREF      #REQUIRED
16 >
17 <!ELEMENT attr (attr*, (%val;)) >
18 <!ATTLIST attr
19   name        NMTOKEN    #REQUIRED
20   kind        NMTOKEN    #IMPLIED
21 >
22 <!ELEMENT set  (%val;)* >
23 <!ELEMENT bool (#PCDATA) >
24 <!ELEMENT int  (#PCDATA) >
25 <!ELEMENT string (#PCDATA) >
```

Auszug aus der DTD für GXL 1.0

Wurzelement eines jeden GXL-Dokumentes ist das Tag *gxl*. Jedes Dokument kann beliebig viele Graphen - durch *graph* gekennzeichnet - beinhalten.

Graphen bestehen aus einer Menge von Knoten (*node*) und Kanten (*edge*), die in beliebiger Reihenfolge auftreten dürfen. Jedes dieser drei Elemente verfügt über ein Attribut *id*, welches innerhalb des Dokumentes eindeutig ist.

Kanten verfügen über die Attribute *from* und *to* vom Typ *IDREF*, welche die IDs von Start- und Zielknoten referenzieren. Die Konsistenz von Ids und deren Referenzen wird bereits auf Ebene von XML sichergestellt.

Konkrete Werte werden in Attribut-Elementen (*attr*) angegeben. Dieses Element verfügt über die Attribute *name* und *kind*, wobei ersteres zwingend und letzteres optional ist. Der eigentliche Wert wird in einem Unterelement angegeben, für das GXL einige atomare Typen wie *bool*, *int*, *string* zur Verfügung stellt. Diese können beliebige Werte annehmen. Zur besseren Strukturierung gibt es auch die Möglichkeit mehrere *attr*-Elemente in einem *Set* zu kapseln.

Darüber hinaus ist in der vollständigen DTD für jedes Element eine Erweiterung in Form von Entities vorgesehen. Knoten und Kanten können selbst wiederum einen Graphen beinhalten. Diese Möglichkeit wird für hierarchische Graphen benötigt.

Werden die Daten im GXL-Format abgespeichert, so werden die zuvor beschriebenen Tags verwendet. Im Gegensatz zum Format FujabaXML werden Attribute, die Referenzen auf andere Objekte enthalten, nicht im Knoten selbst gespeichert. Diese Attribute werden in Kanten ausgelagert. Handelt es sich um einfache Referenzen wie beispielsweise in Zeile 13 bis 15 wird das Attribut im *type*-Tag aufgeführt. Ist das Attribut jedoch in einer Datenstruktur gekapselt (Zeile 25 bis 30), so erhält die Kante ein Attribut namens *key*, welches den konkreten Wert des Attributes enthält.

```

1 <!DOCTYPE gxl SYSTEM "http://www.fujaba.de/gxl.dtd">
2 <gxl xmlns:xlink="http://www.w3.org/1999/xlink">
3   <graph id="FujabaExport">
4     <node id="id14">
5       <type xlink:href="de.uni_paderborn.fujaba.uml.UMLClass"
6         xlink:type="simple"/>
7       <attr name="de.uni_paderborn.fujaba.uml.UMLClass::name">
8         <string>AbstractProduct</string>
9       </attr>
10      <attr name="de.uni_paderborn.fujaba.uml.UMLClass::umlAbstract">
11        <string>true</string>
12      </attr>
13    </node>
14    <edge from="id14" to="id7">
15      <type xlink:href=
16        "de.uni_paderborn.fujaba.uml.UMLClass::roles"/>
17    </edge>
18    <node id="id16">
19      <type xlink:href="de.uni_paderborn.fujaba.uml.UMLClass"
20        xlink:type="simple"/>

```

```
18 <attr name="de.uni_paderborn.fujaba.uml.UMLClass::name">
19 <string>ConcreteProduct</string>
20 </attr>
21 <attr name=
    "de.uni_paderborn.fujaba.uml.UMLClass::umlAbstract">
22 <string>>false</string>
23 </attr>
24 </node>
25 <edge from="id16" to="id52">
26 <type xlink:href=
    "de.uni_paderborn.fujaba.uml.UMLClass::methods"/>
27 <attr name="key">
28 <string>Product()</string>
29 </attr>
30 </edge>
31 </graph>
32 <gxl>
```

Auszug aus Daten im Format GXL

3.2.4 Fujaba *PRO*jektdatei

Dies ist das Format, das von Fujaba gegenwärtig zur Speicherung von Projekten verwendet wird. Dabei werden die Attribute der einzelnen Objekte blockweise aufgeführt. Die Objekte werden über den bereits bestehenden Speichermechanismus in das FPR-Format übertragen. Dadurch ist das Auslesen der Objekte unabhängig von der Implementierung der Im-/Export-Funktionalität.

Das Datenformat ist abgewandelt worden. Einen wichtigen Unterschied stellen die fehlende Hashtabelle und Header (Zeilen 1 - 9) dar. Ersterer wird im Originalformat benötigt, da zuerst alle Objekte erzeugt und erst anschließend mit Attributen und Referenzen versehen werden. Der Header enthält die Kenndaten des abzuspeichernden Projektes. Näheres siehe [FNT98] ab Seite 77. Da der Lademechanismus ebenfalls neu gestaltet wurde und da die Daten nur einen Ausschnitt aus einem Projekt darstellen, wurde auf Header und Hashtabelle verzichtet.

In den Zeilen elf, fünfzehn und siebzehn beginnt jeweils ein neues Objekt gekennzeichnet durch *, gefolgt von Id und Klassenzugehörigkeit. In den jeweils nachfolgenden Zeilen werden die einzelnen Attribute der Objekte aufgelistet. Diese Zeilen beinhalten den Namen des Attributes sowie den Wert. Gegebenenfalls ist auch die Datenstruktur aufgeführt, in der das Attribut gekapselt ist (Zeile 14).

```

1 # Fujaba-Project-File (do not alter this file!!!)
2 # Filename: main.fpr.gz
3 # Date      : Mon Jun 17 11:16:30 CEST 2002
4 -;FileVersion;3
5 # HashTable of this File
6 -;HashTableLength;76
7 +;10;de.uni_paderborn.fujaba.uml.UMLAttr
8 +;15;de.uni_paderborn.fujaba.uml.UMLMethod
9 +;3;de.uni_paderborn.fujaba.uml.UMLClass
.
.
10 # Object references
11 *;3;de.uni_paderborn.fujaba.uml.UMLClass
12 ~;de.uni_paderborn.fujaba.uml.UMLClass::name;Main
13 ~;de.uni_paderborn.fujaba.uml.UMLClass::attrs;ready;10
.
.
14 ~;de.uni_paderborn.fujaba.uml.UMLClass::methods;setReady(Boolean);15
.
.
15 *;10;de.uni_paderborn.fujaba.uml.UMLAttr
16 ~;de.uni_paderborn.fujaba.uml.UMLAttr::name;ready
.
.
17 *;15;de.uni_paderborn.fujaba.uml.UMLMethod
18 ~;de.uni_paderborn.fujaba.uml.UMLMethod::name;setReady
19 ~;de.uni_paderborn.fujaba.uml.UMLMethod::resultType;39
.
.

```

Auszug aus einem Export im ursprünglichen FPR-Format

In Fujaba verfügen alle Objekte des abstrakten Syntaxgraphen über eine eindeutige ID. Fujaba beginnt bei der Vergabe von IDs bei 0. Der numerische Wert wird in Form eines Strings als nächste ID vergeben. Die Vergabe erfolgt zentral in *BasicIncrement* durch den Aufruf von *getUniqueID()*.

Die IDs sollen auch innerhalb der XML-Dokumente als solche dienen. XML bietet dazu den Attributtyp ID und garantiert dadurch die Eindeutigkeit der IDs. Gemäß der Spezifikation von XML dürfen IDs jedoch nicht mit einer Ziffer beginnen. Um dieser Einschränkung gerecht zu werden, erhalten alle IDs in Fujaba den Präfix *id*. Dies geschieht wie bisher in *BasicIncrement*. Als Initialwert für IDs wird nun *id0*

verwendet. Beim Inkrementieren des numerischen Wertes bleiben die ersten beiden Zeichen unberücksichtigt.

Der nachfolgende Auszug zeigt die Daten aus dem oben aufgeführten Beispiel nachdem die beschriebenen Modifikationen durchgeführt wurden:

```
11 *;id3;de.uni_paderborn.fujaba.uml.UMLClass
12 ~;de.uni_paderborn.fujaba.uml.UMLClass::name;Main
13 ~;de.uni_paderborn.fujaba.uml.UMLClass::attrs;ready;id10
    .
    .
14 ~;de.uni_paderborn.fujaba.uml.UMLClass::methods;setReady(Boolean)
    ;id15
    .
    .
15 *;id10;de.uni_paderborn.fujaba.uml.UMLAttr
16 ~;de.uni_paderborn.fujaba.uml.UMLAttr::name;ready
    .
    .
17 *;id15;de.uni_paderborn.fujaba.uml.UMLMethod
18 ~;de.uni_paderborn.fujaba.uml.UMLMethod::name;setReady
19 ~;de.uni_paderborn.fujaba.uml.UMLMethod::resultType;id39
    .
    .
```

Auszug aus einem Export im modifizierten FPR-Format

3.2.5 FujabaXML

Da das Zielformat der Daten GXL ist, bietet es sich an, die Daten möglichst früh in ein XML-Format zu konvertieren, um so die Möglichkeiten der Extensible Markup Language zu nutzen. *FujabaXML* ist daher als Ausgangspunkt für die Transformation in Richtung XML-basierter Formate gedacht.

Eine direkte Umformung von FPR nach GXL ist möglich, aber sehr unübersichtlich und schwierig auf andere Formate zu erweitern. Vor allem aber ist eine solche Implementierung äußerst unflexibel gegenüber Änderungen von Ein- oder Ausgabeformat. Da GXL ein recht neuer Standard ist, können Änderungen oder Erweiterungen auf dieser Seite nicht ausgeschlossen werden. Der Mechanismus zur Transformation müsste in so einem Fall komplett neu implementiert werden. Um FujabaProject

von GXL und anderen XML-basierten Formaten zu entkoppeln, wurde die XML-Untersprache FujabaXML eingeführt.

Bei deren Entwurf stand die möglichst einfache und direkte Umsetzung vom jetzigen Fujaba-Format FPR in einen XML-Dialekt im Vordergrund. Die Umwandlung in FujabaXML erfolgt unmittelbar im Anschluss an die Erzeugung des FPR-Formates. So können alle weiteren Verarbeitungsschritte ausschließlich auf XML-Ebene stattfinden. Dieses Format dient ausschließlich als Bindeglied zwischen dem Lade- und Speicheralgorithmus Fujaba's und anderen XML-basierten Formaten. FujabaXML ist nicht als Ausgangs- oder Endformat konzipiert und das Laden und Speichern dieses Formates wird nicht durch die Implementierung unterstützt.

Als XML-Dialekt ist FujabaXML durch eine DTD definiert:

```

1 <!ELEMENT fujaba_export (comment?, diagram+)>
2 <!ELEMENT diagram (comment?, object+)>
3 <!ELEMENT object (class,(attribute)*)>
4 <!ATTLIST object
5   id      ID                #REQUIRED
6 >
7 <!ELEMENT attribute (value|(reference,value) )>
8 <!ATTLIST attribute
9   name   NMTOKEN            #IMPLIED
10 >
11 <!ELEMENT reference (#PCDATA)>
12 <!ELEMENT value (#PCDATA)>
13 <!ELEMENT class (#PCDATA)>
14 <!ELEMENT comment (#PCDATA)>

```

Document Type Definition für FujabaXML

Das Wurzelement *fujaba_export* umschließt alle zu exportierenden Objekte. Da die Objekte entsprechend ihrer Zugehörigkeit zu Diagrammen gruppiert sind, befinden sich unter der Wurzelebene *diagram*-Tags. Diese erhalten die Gruppierung der Objekte. Somit stammen alle Objekte unterhalb eines *diagram*-Tags aus dem selben Kontext innerhalb von Fujaba (z.B. Klassendiagramm). Die *comment*-Tags auf Wurzel- und Diagrammebene bieten eine Möglichkeit den exportierten Daten zusätzliche Informationen in Form von Kommentaren mitzugeben. Diese sind optional. Die ID-Nummern der Objekte werden hier als Attribute vom Typ ID repräsentiert, da dies ihrer Funktion entspricht. Außerdem kann so auch in diesem Format die Eindeutigkeit der IDs garantiert werden. Jedes gespeicherte Objekt

3 Datenaustausch

wird durch `<object>` und `</object>` geklammert (Zeilen 4 bis 27 und 28 bis 33). Die Klassenzugehörigkeit der Objekte ist im Unterelement `class` angegeben. Die `attribute`-Elemente kapseln den Namen und den Wert der Attribute des Objektes. Diese Werte sind bei konkreten Werten und einfachen Referenzen atomar. Bei Referenzen, die in Hashtablen vorgehalten werden, wird zusätzlich noch ein Schlüssel benötigt. Referenzen auf andere Objekte sind hier noch wie im FPR-Format als Attribute dargestellt. Die Umformung zu Kanten findet erst zu einem späteren Zeitpunkt statt.

```
1 <!DOCTYPE fujaba_export SYSTEM
                                "http://www.fujaba.de/fujaba.dtd">
2 <fujaba_export xmlns:xlink="http://www.w3.org/1999/xlink">
3   <diagram>
4     <object id="id3">
5       <class>de.uni_paderborn.fujaba.uml.UMLClass</class>
6       <attribute name=
9         "de.uni_paderborn.fujaba.uml.UMLClass::name">
7         <value>Main</value>
8       </attribute>
9       <value>>false</value>
10      </attribute>
11     <attribute name=
12       "de.uni_paderborn.fujaba.uml.UMLClass::attrs">
12       <reference>ready</reference>
13       <value>id10</value>
14     </attribute>
15     <attribute name=
16       "de.uni_paderborn.fujaba.uml.UMLClass::methods">
16       <reference>isReady()</reference>
17       <value>id22</value>
18     </attribute>
19     <attribute name=
20       "de.uni_paderborn.fujaba.uml.UMLClass::methods">
20       <reference>setReady(Boolean)</reference>
21       <value>id15</value>
22     </attribute>
23     <attribute name=
24       "de.uni_paderborn.fujaba.uml.UMLClass::methods">
24       <reference>showResult()</reference>
25       <value>id14</value>
26     </attribute>
27   </object>
28   <object id="id10">
```

```

29     <class>de.uni_paderborn.fujaba.uml.UMLAttr</class>
30     <attribute name=
           "de.uni_paderborn.fujaba.uml.UMLAttr::name">
31     <value>ready</value>
32     </attribute>
           ...
33 </object>
34 </diagram>
35 </fujaba_export>

```

Auszug aus einem FujabaXML-Dokument

3.3 Document Object Model

Beim *Document Object Model* [DOM] handelt es sich um eine API, die den Zugriff auf die Elemente von beliebigen XML-basierten Dokumenten beschreibt. Für eine vollständige Spezifikation siehe [DOM].

Durch DOM wird die Dokumentstruktur auf eine Objektstruktur abgebildet. In dieser Objektstruktur treten die einzelnen Elemente als Baumknoten auf. Dabei bilden nicht nur die eigentlichen Elemente Knoten, sondern auch alle anderen Bestandteile des XML-Dokumentes treten als Knoten im DOM-Baum auf.

In der Java API sind diese Baumknoten als Interfaces umgesetzt. Die nachfolgende Liste führt einige dieser Interfaces auf. Für eine vollständige Liste siehe [DOM] und [JAPI]. Ab Version 1.4 unterstützt Java XML durch interne Parser. In diesen sind alle Knotentypen als Interfaces implementiert. Als Super-Interface dient hier *Node*.

- **Attr** - Attribute von Elementen werden in diesem Interface gekapselt.
- **Document** - Dieses Interface repräsentiert das gesamte XML-Dokument und ist somit Wurzel des DOM-Baumes.
- **Node** - Alle anderen Interfaces erben von Node. Es stellt die Grundfunktionalität aller Interfaces im DOM zur Verfügung.
- **DocumentType** - Der Verweis eines XML-Dokumentes auf eine DTD oder ein Schema wird durch das DocumentType-Interface repräsentiert
- **Text** - Dieser Knoten stellt den Inhalt von Elementen oder Attributen dar.

- **Element** - Einzelne Elemente des XML-Dokumentes werden durch dieses Interface repräsentiert.
- **ProcessingInstruction** - Dieses Interface kapselt die Verarbeitungsanweisungen, wie z.B. die erste Zeile `<?xml version="1.0" ?>`.
- **Comment** - Repräsentiert die Kommentare einer XML-Dokumentes, d.h. alle Zeichen zwischen `<!--` und `-->`.
- **Entity** Alle Arten von Entities werden durch dieses Interface repräsentiert.

Auszug der Liste der Interfaces des DOM, entnommen aus [JAPI]

Da das gesamte XML-Dokument im Speicher vorgehalten wird, benötigt DOM wesentlich mehr Speicherplatz als beispielsweise SAX. Anders als bei SAX ist jedoch beliebiger Zugriff auf Knoten und Navigation zwischen Knoten möglich. Dadurch ergeben sich wesentlich mehr Möglichkeiten zur Manipulationen des XML-Dokumentes durch Hinzufügen, Löschen oder Verschieben von Knoten/Teilbäumen.

Es existieren mehrere verschiedene Implementierungen dieser Interfaces. Java stellt den DOM-Parser Crimson zur Verfügung. Eine weitere weitverbreitete Implementierung stellt der aus dem Apache-Projekt stammende Parser Xerces dar.

Es gibt jedoch Unterschiede im Verhalten der einzelnen Implementierungen. So bietet beispielsweise Crimson keine Möglichkeit den *DocumentType*-Knoten, der den Verweis auf die DTD enthält, nachträglich zu ändern.

3.4 XML Datentransformation

Die XML bietet außer der Möglichkeit zur Definition von Dokumentstrukturen auch ein Konzept, um diese Strukturen zu transformieren. Zu diesem Zweck wurden die *eXtensible Stylesheet Language: Transformation*, kurz [XSLT], entworfen. Die Grundlage für XSLT bilden spezielle XML-Dokumente, Stylesheets genannt. Diese bestehen grundsätzlich aus einer Menge von sogenannten Template-Regeln, die aus Suchmustern (*xsl:template match='/'*) und Anweisungen bestehen. Die Anweisungen legen fest, auf welche Art und Weise der Knoten verarbeitet wird. Um in der Baumstruktur des XML-Dokumentes navigieren zu können, wird XPath-Syntax verwendet. XPath nutzt dabei eine Syntax ähnlich den Sprachelementen, die von Dateisystemen bekannt sind. Eine ausführlichere Beschreibung der Syntax von XPath bietet [XPath].

Mit XSLT bietet XML ein Konzept, um verschiedene XML-Dialekte ineinander zu überführen. Genau diese Funktionalität wird bei der Transformation zwischen Formaten wie FujabaXML und GXL benötigt. Da die entsprechenden XSLT-Transformatoren in den XML-Parsern bereits umgesetzt sind, können sie in Fujaba eingebunden werden.

Bei der Transformation können verschiedene Quellen und Ziele der Daten auftreten. So können Speicher und Datei bei einer Transformation sowohl Quelle als auch Ziel sein. Um für diese verschiedenen Fälle keine aufwendige Fallunterscheidung durchführen zu müssen, kapselt die Klasse *XSLTransformer* diese Transformer. Diese stellt eine überladene Methode *transform()* zur Verfügung bei der Quelle und Ziel durch die Signatur bestimmt werden. Der Parameter *how* übergibt dabei das Stylesheet der aktuellen Transformation. Die XML-Dokumente werden dabei als *Document*-Node übergeben. Die vier Erscheinungsformen dabei sind:

- *Document transform(File from, File how)* - Eine Datei wird in ein Dokument überführt
- *Document transform(Document from, File how)* - Ein Dokument wird in ein anderes transformiert.
- *void transform(Document from, File to, File how)* - Ein Dokument wird in einer Datei gespeichert.
- *void transform(File from, File to, File how)* - Transformation von Datei zu Datei.

Die vierte Methode ist nur der Vollständigkeit wegen aufgeführt. Sie wird in der aktuellen Implementierung nicht verwendet.

3.5 Technische Realisierung

Die Umsetzung des zuvor beschriebenen Konzeptes beruht auf der Anbindung des Formates FujabaXML an den Lade- und Speicheralgorithmus, sowie der Transformation von FujabaXML zum Austauschformat beziehungsweise umgekehrt.

Die Transformation zwischen FujabaXML und dem Lade- und Speicheralgorithmus wird unabhängig vom Austauschformat durchgeführt und kann deshalb nicht vom Benutzer beeinflusst werden. Da die restlichen Transformationen jedoch vom jeweiligen Austauschformat abhängen, lassen sich diese durch den Benutzer konfigurieren.

Die Benutzerschnittstelle wird im nachfolgenden Abschnitt erläutert. Die Abschnitte Export und Import verdeutlichen die einzelnen Schritte der Transformationen, die bei Ex- und Import durchgeführt werden.

3.5.1 Benutzerschnittstelle

Der Dialog in Grafik 3.2 bildet die Benutzerschnittstelle der Im-/Export-Funktionalität. Er wird für Im- und Export gleichermaßen verwendet, da die Funktionalität in beiden Fällen die selbe ist: Der Benutzer spezifiziert eine Datei, die als Quelle oder Ziel der Daten dient. Des weiteren besteht der Dialog aus zwei Listen. Rechts sind alle dem Dialog bekannten Stylesheets aufgelistet; links alle auf die Daten anzuwendenden Stylesheets. Diese Liste wird vom Benutzer entsprechend konfiguriert.

Sollen Daten importiert werden, werden die Daten aus der Quelldatei ausgelesen, durch die ausgewählten Stylesheets manipuliert und an den neuen Lademechanismus weiter gereicht. Umgekehrt werden beim Export die durch den Speichermechanismus bereitgestellten Daten anhand der Stylesheets manipuliert und anschließend in der angegebenen Zieldatei abgespeichert.



Abbildung 3.2: Dialog zum Im- und Export von GXL-Dateien

Der Dialog entnimmt alle verfügbaren Stylesheets der Datei *stylesheet.xml*. Dieses XML-Dokument enthält für jedes Stylesheet ein Tag *stylesheet*, welcher in seinen Attributen den Dateinamen und eine aussagekräftige Bezeichnung kapselt (Zeilen 2 und 3). Da zur Verarbeitung mindestens ein Stylesheet benötigt wird, gibt es die Möglichkeit ein Stylesheet als Standardwert zu setzen. Dies geschieht mittels der optionalen Attribute *import* und *export*. Haben diese den Wert *true*, so sind die entsprechenden Stylesheet beim Aufruf des Dialogs bereits in der Liste der *applied stylesheets* aufgeführt.

```

1 <stylesheets>
2 <stylesheet name="Fujaba -> GXL" file="xml2gxl.xsl" export="true"/>
3 <stylesheet name="GXL -> Fujaba" file="gxl2xml.xsl" import="true"/>
4 </stylesheets>

```

Die Datei *stylesheet.xml* listet die bekannten Stylesheets auf.

Mittels der beiden Methoden *showImportDialog()* und *showExportDialog()* wird der Dialog aufgerufen und wie oben beschrieben mit Standardwerten initialisiert. Beim Import werden die neuen Einträge an erster Stelle eingefügt; beim Export werden diese der Liste angehängt. Die Stylesheets werden jedoch immer von oben nach unten abgearbeitet.

Da Dateien im XML-Format naturgemäß sehr viele Tags gleichen Namens enthalten, eignen sie sich gut zur Datenkompression. Um diesen Vorteil zu nutzen, wird standardmäßig die Endung *.gz* an den Dateinamen angehängt. Wird dieser nicht manuell vom Benutzer entfernt, so wird bei der Speicherung der Datei die Java-interne GZIP-Kompression angewandt.

In diesem Abschnitt werden die verwendeten Stylesheets näher beschrieben. Stylesheets liefern dem XML-Transformer Vorschriften, wie er auf welche Tags zu reagieren hat. Die Vorschriften werden durch sogenannte Templates beschrieben. Dabei wird in der ersten Zeile eines jeden Templates definiert, auf welche Tags das entsprechende Template anzuwenden ist. Die Festlegung der Tags ist nicht auf deren Namen beschränkt. Mittels der Syntax von [XPath] können komplexe Regeln definiert werden. Innerhalb der Templates wird die Struktur des Ausgabedokumentes festgelegt. Dazu können einfache Tags angegeben, gezielt die Werte von Attributen und anderen Elementen ausgelesen, oder andere Templates aufgerufen werden.

Da alle Stylesheets selbst gültige XML-Dokumente darstellen, müssen sie wie im nachfolgend aufgeführten Header mit einer XML-Deklaration beginnen (siehe erste Zeile). In den Zeilen vier und fünf werden die für *XSL: Transformation* benötigten Namespaces angegeben. Mit *xsl:output* können die Eigenschaften des Zieldokumentes beschrieben werden. Hier ist als Ausgabeformat ein XML-Dialekt gefordert (Zeile 7). Um das Zieldokument gegen eine DTD validieren zu können, wird diese mit *doctype-system="http://www.fujaba.de/gxl.dtd"* angegeben. Der String *http://www.fujaba.de/gxl.dtd* dient hierbei zur eindeutigen Identifizierung der DTD. Dieser wird später bei der Validierung vom XML-Parser aufgelöst. Mit *indent="yes"* wird der Transformator aufgefordert die Elemente des Ausgabedokumentes einzurücken.

```
1 <?xml version=""1.0""?>
```

```
2
3 <xsl:stylesheet version=''1.0''
4 xmlns:xsl=''http://www.w3.org/1999/XSL/Transform''
5     xmlns:xlink=''http://www.w3.org/1999/xlink''>
6
7 <xsl:output method=''xml''
8     doctype-system=''http://www.fujaba.de/gxl.dtd''
9     indent=''yes''/>
```

Header der zur Transformation verwendeten Stylesheets

3.5.2 Export

Beim Export müssen die Daten zunächst den Speichermechanismus durchlaufen und nach FujabaXML überführt werden. Anschließend findet die Transformation ins gewünschte Austauschformat statt. Die Grafik 3.3 zeigt die am Export beteiligten Klassen mit einem Schwerpunkt auf der Transformation.

Werden Daten aus einem Diagramm exportiert, so beginnt der gesamte Vorgang bei der Klasse *ExportGXLAction*. Diese gibt dem Benutzer durch den Dialog 3.2 die Möglichkeit die Transformationen von FujabaXML hin zum gewünschten Austauschformat festzulegen. Sind diese und die Zieldatei durch den Benutzer festgelegt worden, so wird ein neues FileSaver-Objekt angelegt. Dabei werden ein Iterator mit den abzuspeichernden Objekten, die Zieldatei und die anzuwendenden Stylesheets übergeben.

```
1 public FileSaver (Iterator iter, File dest, File[] stylesheets)
2 {
3     this.iter = iter;
4     this.dest = dest;
5     this.stylesheets = stylesheets;
6 }
```

Konstruktor der Klasse *FileSaver*

Der eigentliche Speichervorgang beginnt mit dem Aufruf der Methode *save()*. Dem modifizierten Speichermechanismus wird das abzuspeichernde Objekt und ein Filter übergeben. Dieser Filter definiert alle unerwünschten Objekte (näheres siehe Kapitel 2 *Lade-/Speicher-Algorithmus*). Als Resultat wird ein StringBuffer mit Daten

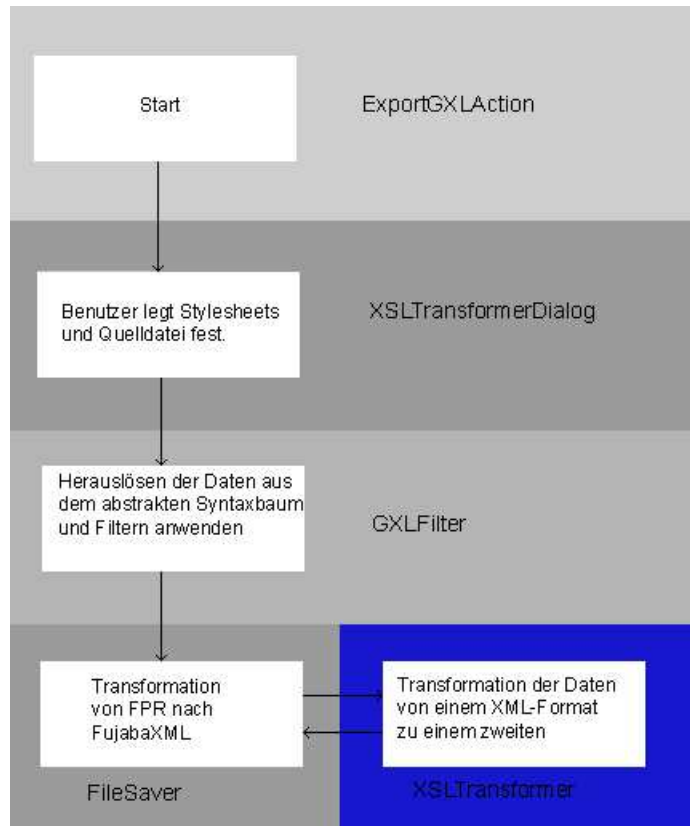


Abbildung 3.3: Übersicht über die am Export beteiligten Klassen

im FPR-Format zurückgegeben. Diese Daten (siehe Beispiel (1)) werden anhand der Methode *fpr2xml()* in das FujabaXML Format überführt. Dabei müssen einige Zeichen ersetzt werden. Die Methode *substitute()* dient der Vereinfachung der Zeichenersetzung. Es müssen dazu nur das Originalzeichen, das neue Zeichen sowie der Kontext angegeben werden.

```

1 *;id3;de.uni_paderborn.fujaba.uml.UMLClass
2 ~;de.uni_paderborn.fujaba.uml.UMLClass::name;Main
3 ~;de.uni_paderborn.fujaba.uml.UMLClass::attrs;ready;id10
4 *;id10;de.uni_paderborn.fujaba.uml.UMLAttr
5 ~;de.uni_paderborn.fujaba.uml.UMLAttr::name;ready
  
```

Beispiel (1) - Daten im Format FPR

Hier liegt das Beispiel im FPR Format vor. In dieser Form umfasst es lediglich fünf Zeilen. Zur weiteren Verarbeitung wird es nach FujabaXML transformiert.

3 Datenaustausch

Zur Zeit dient das Semikolon zur Trennung von Objektreferenz, Attributname und Attributwert innerhalb der FPR-Daten. Da das Semikolon jedoch Bestandteil von XML-Entities ist, muss es ersetzt werden. Als Ersatz dient hier Unicode *FFFA*. Da in Javakommentaren und somit im Quellcode alle Zeichen erlaubt sind, müssen alle Zeichen, die Bestandteil der XML-Syntax sind durch entsprechende Entities ersetzt werden. Dabei muss *ℓ* zuerst ersetzt werden, da es gleichzeitig Bestandteil aller Entities ist. Fujaba selbst ersetzt bereits im bestehenden Speichermechanismus den Zeilenumbruch und das Semikolon im Quellcode durch die Unicode-Zeichen 1 und 2. Diese Zeichen sind jedoch keine gültigen XML-Symbole. Deshalb werden sie durch *linebreak* und *semicolon* ersetzt. Alle diese Ersetzungen müssen beim Import der Daten rückgängig gemacht werden. Zu diesem Zweck wird die Methode *reReplace()* zur Verfügung gestellt.

Die so aufbereiteten Daten werden dann in FujabaXML transformiert. Zuerst wird ein XML-Header geschrieben. Dieser kennzeichnet die Daten als XML-Dokument, verweist auf die DTD und enthält Informationen über die Kodierung. Die FPR-Daten werden zeilenweise abgearbeitet. Beginnt die Zeile mit einem *, so wird ein neues *object*-Tag geöffnet - und das vorhergehende gegebenenfalls geschlossen. Attribute werden in *attribute*-Tags gekapselt. Die Werte und Referenzen sind in *value*- und *reference*-Tags gekennzeichnet. Einträge der Hashtabelle, Systeminformationen und Kommentare werden nicht verarbeitet und sind somit nicht Bestandteil der Zieldaten.

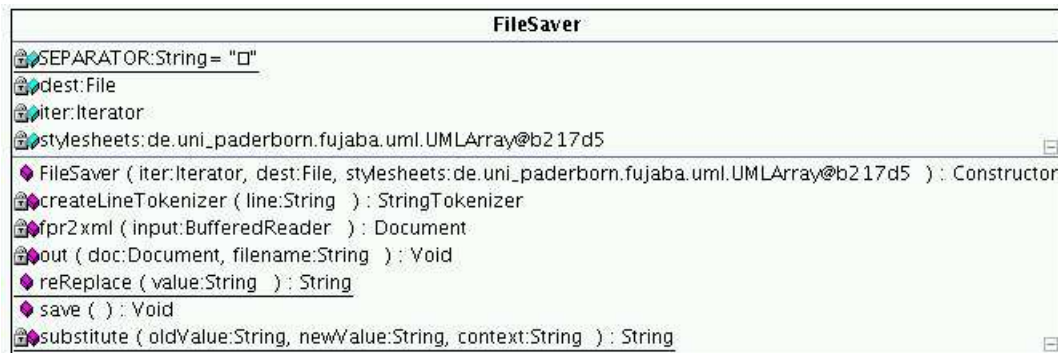


Abbildung 3.4: Die Klasse *FileSaver*

Die Daten liegen anschließend im FujabaXML Format vor (siehe Beispiel (2)) und können gemäß den Vorgaben des Benutzers durch ein oder mehrere Stylesheets bearbeitet werden. Dies geschieht durch mehrere Aufrufe der *transform*-Methode der Klasse *XSLTransformer*. Das jeweils letzte Stylesheet schreibt die Daten in die Zieldatei.

```
1 <object id="id3">
2 <class>de.uni_paderborn.fujaba.uml.UMLClass</class>
```

```

3   <attribute name=
           "de.uni_paderborn.fujaba.uml.UMLClass::name">
4     <value>Main</value>
5   </attribute>
6   <attribute name=
           "de.uni_paderborn.fujaba.uml.UMLClass::attrs">
7     <reference>ready</reference>
8     <value>id10</value>
9   </attribute>
10  </object>
11  <object id="id10">
12    <class>de.uni_paderborn.fujaba.uml.UMLAttr</class>
13    <attribute name=
           "de.uni_paderborn.fujaba.uml.UMLAttr::name">
14      <value>ready</value>
15    </attribute>
16  </object>

```

Beispiel (2) - Daten im Format FujabaXML

Durch die XML-Tags ist das Beispiel wesentlich größer geworden. Alle Informationen sind jedoch erhalten geblieben. Referenzen werden noch innerhalb der Objekte dargestellt.

Ist GXL das angestrebte Zielformat, so ist dabei einer der wichtigsten Punkte die Verschiebung der Attribute, die Referenzen enthalten, in die Kanten. Dadurch entsteht ein Graph mit den Objekten als Knoten und Referenzen als Kanten. Das Ausgabedokument enthält somit ausschließlich Tags gemäß der Document Type Definition von GXL.

Das nachfolgende Template verwaltet die Transformation der einzelnen Objekte in *nodes* des GXL-Dokumentes. Die Id des Objektes wird übernommen und bleibt so im *<node>*-Element erhalten. Die Verarbeitung des *class*-Tags und somit der Klassenzugehörigkeit wird an ein weiteres Template delegiert. Bei der Einbindung der Attribute wird zwischen solchen mit Referenzen und solchen mit konkreten Werten unterschieden. Erstere werden außerhalb der *node*-Tags verarbeitet, letztere innerhalb.

```

1  <xsl:template match="object">
2    <node>
3      <xsl:attribute name="id"><xsl:value-of select="@id"/>

```

3 Datenaustausch

```
    </xsl:attribute>
4   <xsl:apply-templates select="class"/>
5   <xsl:apply-templates select="attribute[not(starts-with(./value/
    text()),'id'))]" />
6   </node>
7   <xsl:apply-templates select="attribute[starts-with(./value/
    text()),'id')]" />
8 </xsl:template>
```

Template, das Objekte als ganzes bearbeitet

Das nachfolgende Template wird nur für die *class*-Tags aufgerufen. Es wird nur dann aktiv, wenn das *class*-Element direkt unterhalb eines *object*-Elementes auftritt. Die Klassenzugehörigkeit des Objektes wird innerhalb des *type*-Elementes gespeichert.

```
1 <xsl:template match="class[parent::object]">
2   <type>
3     <xsl:attribute name="xlink:href"><xsl:value-of select="."/>
        </xsl:attribute>
4   </type>
5 </xsl:template>
```

Template, das die Klassenzugehörigkeit transformiert

Die Elemente *value* und *reference* enthalten Strings, welche die Werte und Referenzen der Attribute wie *false*, *0* oder *id12345* kapseln. Um diese in einem GXL-Dokument handhaben zu können, werden sie in *string*-Tags eingeschlossen, abgespeichert.

```
1 <xsl:template match="value | reference">
2   <string>
3     <xsl:value-of select="."/>
4   </string>
5 </xsl:template>
```

Template zur Verarbeitung der Tags *value* und *reference*

Die nächsten beiden Templates haben die Aufgabe, Attribute zu verarbeiten, die nur über ein Unterelement verfügen, also direkt einen Wert oder eine Referenz enthalten.

Die Templates filtern Attribute, deren Werte in einer Datenstruktur geschachtelt sind, durch die Abfrage der Anzahl der untergeordneten Elemente (*count(*)*) in den Zeilen 1 und 17 heraus. Das erste der beiden Templates spricht nur auf Attribute an, deren Wert mit *id* beginnt. Diese Attribute enthalten Referenzen und werden als Kanten gespeichert. Alle die Attribute, welche diese Bedingung nicht erfüllen, werden in *attr*-Tags gekapselt (Zeilen 18 - 21) und verbleiben als Unterelemente innerhalb des Knotens.

```

1  <xsl:template match=
      "attribute[count(*)=1 and starts-with(./value/text(),'id')] ">
2  <edge>
3    <xsl:attribute name="from">
4      <xsl:value-of select="./@id"/>
5    </xsl:attribute>
6    <xsl:attribute name="to">
7      <xsl:value-of select="./value/text()"/>
8    </xsl:attribute>
9    <type>
10     <xsl:attribute name="xlink:href">
11       <xsl:value-of select="@name"/>
12     </xsl:attribute>
13   </type>
14 </edge>
15 </xsl:template>
16
17 <xsl:template match=
      "attribute[count(*)=1 and not(starts-with(./value/text()),
      'id'))] ">
18 <attr>
19   <xsl:attribute name="name">
20     <xsl:value-of select="@name"/>
21   </xsl:attribute>
22 <xsl:apply-templates/>
23 </attr>
24 </xsl:template>

```

Templates für Attribute mit einelementigen Werten

Im Gegensatz zu den beiden vorhergehenden Templates, verarbeitet dieses ausschließlich mehrelementige Attribute. Solche Attribute kapseln Referenzen auf andere Objekte in Datenstrukturen wie beispielsweise einer Hashtabelle. Dabei muss

3 Datenaustausch

neben der zu speichernden Id auch der entsprechende Schlüssel gespeichert werden. Deshalb erhält die Kante ein Attribut namens *key*, welches den Schlüssel beinhaltet.

```
1 <xsl:template match=
    "attribute[count(*)>1 and starts-with(./value/text(),'id')] ">
2 <edge>
3   <xsl:attribute name="from">
4     <xsl:value-of select="./@id"/>
5   </xsl:attribute>
6   <xsl:attribute name="to">
7     <xsl:value-of select="./value/text()"/>
8   </xsl:attribute>
9   <type>
10    <xsl:attribute name="xlink:href">
11      <xsl:value-of select="@name"/>
12    </xsl:attribute>
13  </type>
14  <attr>
15    <xsl:attribute name="name">key</xsl:attribute>
16    <xsl:apply-templates select="reference"/>
17  </attr>
18 </edge>
19 </xsl:template>
```

Template zur Verarbeitung mehrelementiger Attribute

Nach der Anwendung dieser Templates liegen die Daten im Format GXL vor. Das Beispiel hat nun folgende Struktur:

```
1 <node id="id3">
2   <type xlink:href="de.uni_paderborn.fujaba.uml.UMLClass"
    xlink:type="simple"/>
3   <attr name="de.uni_paderborn.fujaba.uml.UMLClass::name">
4     <string>Main</string>
5   </attr>
6 </node>
7 <edge from="id3" to="id10">
8   <type xlink:href=
    "de.uni_paderborn.fujaba.uml.UMLClass::attr"/>
9   <attr name="key">
10    <string>ready</string>
```

```

11 </attr>
12 </edge>
13 <node id="id10">
14   <type xlink:href="de.uni_paderborn.fujaba.uml.UMLAttr"
        xlink:type="simple"/>
15   <attr name="de.uni_paderborn.fujaba.uml.UMLAttr::name">
16     <string>ready</string>
17   </attr>
18 </node>

```

Beispiel (3) - Daten im Format GXL

Die Referenz zwischen den Objekten *id3* und *id10* ist hier in eine Kante ausgelagert worden. Da der Aufbau von GXL komplexer ist als der von FujabaXML, ist das Beispiel noch umfangreicher geworden.

3.5.3 Import

Beim Import müssen die Daten zuerst aus dem vorliegenden Format nach FujabaXML transformiert werden. Erst anschließend können die durch den Lademechanismus verarbeitet werden. Die Grafik 3.5 zeigt die am Import beteiligten Klassen mit einem Schwerpunkt auf der Transformation.

Der Benutzer kann anhand des Dialogs 3.2 die notwendigen Transformationen vom vorliegenden Format nach FujabaXML spezifizieren. Mit diesen Informationen wird ein neues *Validator*-Objekt erzeugt, welches die Möglichkeit bietet aus einer Datei einen DOM-Baum zu generieren.

Da dieser DOM-Baum auf Daten im Format FujabaXML basiert, werden die Daten zuvor durch die Klasse *XSLTransformer* nach FujabaXML überführt.

Wichtiger Bestandteil ist dabei die Wiedereingliederung der Kanten in die entsprechenden Knoten, da die DTD von FujabaXML keine Kanten zwischen den einzelnen Objekten vorsieht. Dies geschieht durch die nachfolgend aufgeführten Templates.

Diese Templates sorgen dafür, dass die Root-Tags des Ausgabedokumentes gesetzt werden. Anschließend wird die Verarbeitung der untergeordneten Elemente angestoßen.

```

1 <xsl:template match="gxl">

```

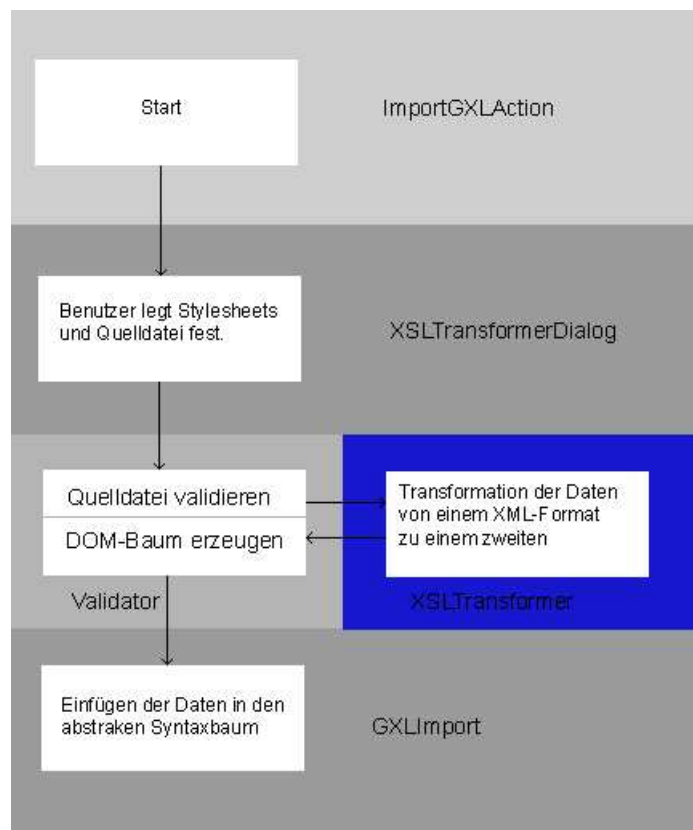


Abbildung 3.5: Übersicht über die am Import beteiligten Klassen

```

2 <fujaba_export>
3 <xsl:apply-templates select="graph"/>
4 </fujaba_export>
5 </xsl:template>
6
7 <xsl:template match="graph">
8 <diagram>
9 <xsl:apply-templates/>
10 </diagram>
11 </xsl:template>
  
```

Die Templates für die obersten Elemente der Hierarchie

Das nachfolgende Template erzeugt aus einem Knoten und den von ihm ausgehenden Kanten wieder ein *object*-Element für das Zielformat FujabaXML. Neben Id und Klassenzugehörigkeit, die aus dem Attribut *id* und dem Unterelement *type* gewonnen werden, müssen auch die Attribute wieder hergestellt werden. Dabei müssen

sowohl die als Unterelemente gespeicherten Attribute, als auch die in Kanten verschobenen Attribute berücksichtigt werden. Letztere können über ihr *from*-Attribut identifiziert werden, da dessen Wert der Id des aktuellen Knotens entspricht.

```

1 <xsl:template match="node">
2   <xsl:variable name="ownid">
3     <xsl:value-of select="@id"/>
4   </xsl:variable>
5   <object>
6     <xsl:attribute name="id">
7       <xsl:value-of select="@id"/>
8     </xsl:attribute>
9     <xsl:for-each select="type">
10      <class>
11        <xsl:value-of select="@xlink:href"/>
12      </class>
13    </xsl:for-each>
14    <xsl:apply-templates/>
15    <xsl:apply-templates select="../edge[@from=\$ownid]"/>
16  </object>
17 </xsl:template>

```

Template zur Erzeugung von *object*-Elementen

Alle Attribute, die innerhalb von *node*-Elementen auftreten, werden durch dieses Template verarbeitet. Da diese Attribute einelementig sind, werden lediglich Name und Wert benötigt, wobei letzterer im Unterelement *string* gespeichert ist.

```

1 <xsl:template match="attr[child::string and not(@name='comment')] ">
2   <attribute>
3     <xsl:attribute name="name">
4       <xsl:value-of select="@name"/>
5     </xsl:attribute>
6     <value>
7       <xsl:value-of select="child::string/text()"/>
8     </value>
9   </attribute>
10 </xsl:template>

```

Template zur Verarbeitung von internen Attributen

Sollen die in Kanten verschobenen Attribute verarbeitet werden, so findet dieses Template Anwendung. Dieses Template wird aus dem Template für *node*-Elemente heraus aufgerufen. Somit müssen die Attribute nicht aufwändig in den entsprechenden Knoten zurückverschoben werden. Bei der Generierung der untergeordneten Elemente *value* und *reference* muss zwischen ein- und mehrelementigen Attributen unterschieden werden. Dies geschieht durch Abfragen der Anzahl der Unterelemente von *attr* in den Zeilen 6 und 11.

```
1 <xsl:template match="edge">
2   <attribute>
3     <xsl:attribute name="name">
4       <xsl:value-of select="child::type/@xlink:href"/>
5     </xsl:attribute>
6     <xsl:if test="count(attr)=0">
7       <value>
8         <xsl:value-of select="@to"/>
9       </value>
10    </xsl:if>
11    <xsl:if test="count(attr)>0">
12      <reference>
13        <xsl:value-of select="child::attr[@name='key']/string"/>
14      </reference>
15      <value>
16        <xsl:value-of select="@to"/>
17      </value>
18    </xsl:if>
19  </attribute>
20 </xsl:template>
```

Template zur Erzeugung von Attributen aus Kanten

Nach dieser Transformation liegen die Daten im Format FujabaXML vor. Zur weiteren Verarbeitung wird eine Zwischenschicht genutzt, auf welche der Lademechanismus zugreift.

Da die *getDOMTree()*-Methode der Klasse Validator (siehe 3.6) überladen ist muss die Datei nicht angegeben werden. In diesem Fall wird die dem Konstruktor übergebene Quelldatei verwendet. Zusätzlich kann eine Datei mittels der Methode *valid()* testweise validiert werden ohne den gesamten Transformationsprozess anzustoßen.

Um den Lademechanismus von den XML-Strukturen zu entkoppeln, existiert eine Zwischenschicht, die alle zu importierenden Objekte des DOM-Baumes und deren

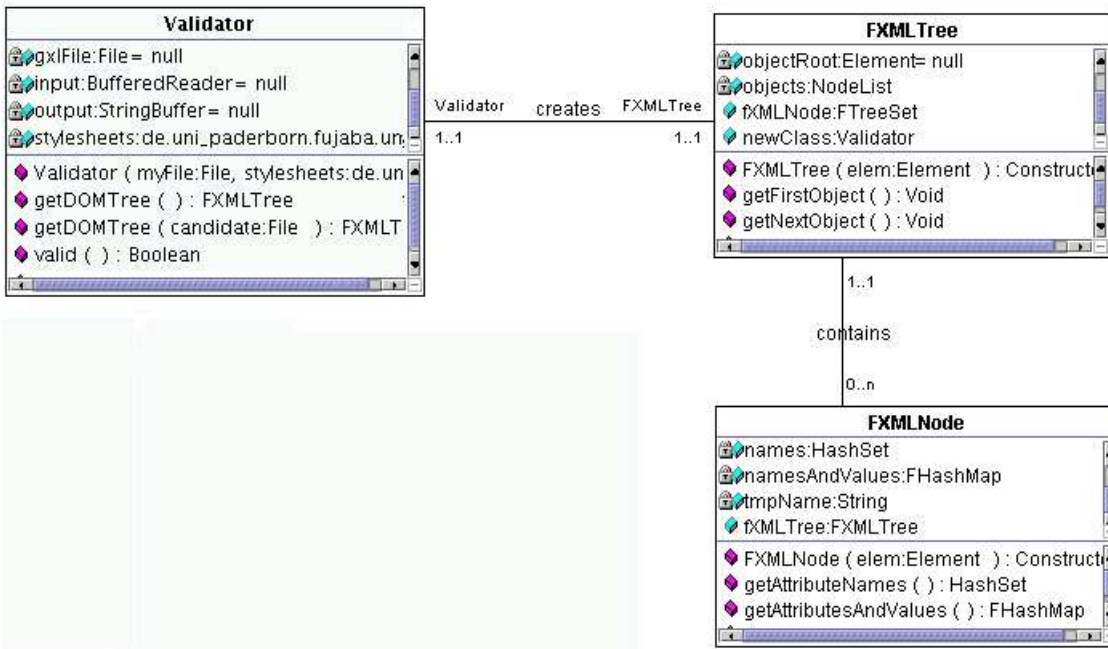


Abbildung 3.6: Statische Struktur der Klassen *Validator*, *FXMLTree* und *FXMLNode*

Attribute in Datenstrukturen vorhält. Die `getDOMTree()`-Methode der Klasse *Validator* (siehe 3.6) kapselt den auf FujabaXML basierenden DOM-Baum in einem *FXMLTree*-Objekt. Die Daten der Fujabaobjekten sind innerhalb des DOM-Baumes in einer *NodeList* gespeichert und werden erst bei Aufruf der Methoden `getFirstObject()` und `getNextObject()` in einem neu erzeugten *FXMLNode*-Objekt gekapselt.

FXMLNode stellt genau ein Objekt aus der Objektwelt Fujabas dar und kapselt alle relevanten Attribute. Die Attribute werden in einer *HashMap* vorgehalten und liegen somit in einer Javatypischen Struktur vor. Der Speichermechanismus kann diese mit den Methoden `getAttributeNames()` und `getAttributesAndValues()` abfragen und dann weiter verarbeiten. Die beim Export vorgenommenen Zeichenersetzungen werden hier wieder rückgängig gemacht.

Der ursprüngliche Lademechanismus wurde nicht dazu konzipiert Teilprojekte im FPR-Format einzulesen und in ein bestehendes Projekt einzubinden. Deshalb wurde der Lademechnismus komplett neu entwickelt (siehe Kapitel 2 *Lade-/Speicheralgorithmus*): Statt Daten im FPR-Format wird ein *FXMLTree* an ein *GXMLImport*-Objekt übergeben, welches die Daten in den abstrakten Syntaxgraphen einfügt.

3.5.4 Weitere Transformationen

Wie bereits im Abschnitt *Probleme und Lösungen* erwähnt, stellt die Erweiterbarkeit des Datenaustauschs um neue Formate ein wichtiger Punkt bei der Konzeption dar. XMI stellt einen verbreitetes Format zum Datenaustausch zwischen Entwicklungsumgebungen dar. Somit bringt die Unterstützung den Vorteil mit sich, Daten, die mit anderen Werkzeugen erstellt wurden, austauschen zu können.

Da bereits ein Stylesheet zur Transformation von XMI nach GXL existiert, wurde eine indirekte Anbindung des Formates XMI an Fujaba gewählt. Dieses Stylesheet stammt im Original von Daniel Volk [XIG] und liegt in der Version 1.0beta als *xig.xsl* vor. Die Gegenrichtung dieser Transformation übernimmt das Stylesheet *gxl2xmi.xsl*. Dabei wird das Format XMI durch die DTD zu UML 1.3 festgelegt.

3.5.5 Bewertung der Transformationen

Eine Transformation kann nur dann erfolgreich für den Austausch von Daten eingesetzt werden, wenn dabei keine wesentlichen Informationen verloren gehen. Der Verlust von Kommentaren oder Systeminformationen ist beispielsweise akzeptabel, da die eigentlichen Daten auch ohne diese verwertbar sind.

Bei der Transformation von FPR nach FujabaXML wird der Header der FPR-Daten nicht nach FujabaXML übertragen. Dies stellt jedoch kein Problem dar, da der neue Lademechanismus diese Informationen nicht benötigt. Wird ein (Teil)-Diagramm importiert, so werden keinerlei Informationen über das Ursprungsprojekt des Diagramms benötigt. In der Gegenrichtung tritt keinerlei Informationsverlust auf, da der Lademechanismus als Datenquelle einen DOM-Baum nutzt, der ausschließlich die Anordnung der FujabaXML-Daten verändert.

Bei der Transformation von FujabaXML nach GXL werden Referenzen als Kanten dargestellt. Die Information über die referenzierten Objekte bleibt jedoch in den Attributen *from* und *to* der Kanten erhalten. Alle anderen Informationen ändern bei der Transformation lediglich ihre Position im XML-Dokument, bleiben aber erhalten. Da FujabaXML ausschließlich Rohdaten enthält, tritt an dieser Stelle kein Verlust von Informationen auf.

Für die Umkehrung einer Transformation gelten natürlich die selben Anforderungen bezüglich Verlustlosigkeit. Um für eine Transformation eine Umkehrung zu definieren wird in der Regel ein zweites Stylesheet benötigt. Einzige Ausnahme hierbei ist die Transformation von FPR nach FujabaXML, für die es keine Umkehrung gibt, da der neue Lademechanismus keine Daten im FPR-Format benötigt, der Speicher-

mechanismus seine Daten jedoch als FPR ausgibt.

4 Beispielsitzung

(Philipp Hoven, Mike Liebrecht)

Anhand einer Beispielanwendung führt dieses Kapitel in die Benutzung des Lade- und Speicheralgorithmus ein. Als Beispiel dient ein Klassendiagramm, das die statische Struktur des Abstract Factory Pattern beschreibt. Im Rahmen eines Export- und Importdurchlaufs werden die wichtigsten Menüs und Dialoge vorgestellt.

Ausgangssituation ist das fertig erstellte Klassendiagramm, welches zu exportieren ist. Die Kontextmenüs zum Speichern des Diagramms, oder Teile des Diagramms, sind auf zwei Wege zu erreichen. Soll das komplette Diagramm gespeichert werden, so ist links im Projektbaum, das zu exportierende Diagramm zu selektieren. Siehe dazu Grafik 4.1.

Alternativ, zur Speicherung einzelner Teile des Diagramms, können auch direkt im Diagramm die zu exportierenden Objekte selektiert werden und über die rechte Maustaste das zugehörige Kontextmenü aktiviert werden. Die Grafik 4.2 zeigt das aktivierte Kontextmenü. Über das Untermenü "Import/Export" sind die Menüpunkte zum Im- und Export von (Teil)-Diagrammen zu erreichen. Da nun in einem ersten Schritt das Klassendiagramm zu exportieren ist, wird der Menüpunkt "Export GXL" selektiert und ausgelöst. Anschließend erscheint der Exportdialog auf dem Bildschirm. Der Benutzer wird nun aufgefordert den Dateinamen und das Format festzulegen. Siehe dazu Grafik 4.3.

In dieser Beispielanwendung soll das Klassendiagramm im GXL-Format in die Datei *demo.gxl* gespeichert werden. Die Angaben werden bestätigt und daraufhin das Klassendiagramm gespeichert. Damit ist der Exportvorgang abgeschlossen.

Um ein Diagramm in Fujaba zu importieren, muss der Benutzer zunächst ein neues Diagramm des zu importierenden Diagrammtyps anlegen, oder ein schon vorhandenes Diagramm des Typs selektieren. In diesem Fall wurde das neue Diagramm "demo2" angelegt. Über das Kontextmenü des Diagramms, wird der Menüpunkt "Import GXL" selektiert und der Importprozess ausgeführt. Siehe dazu Grafik 4.4

4 Beispielsitzung

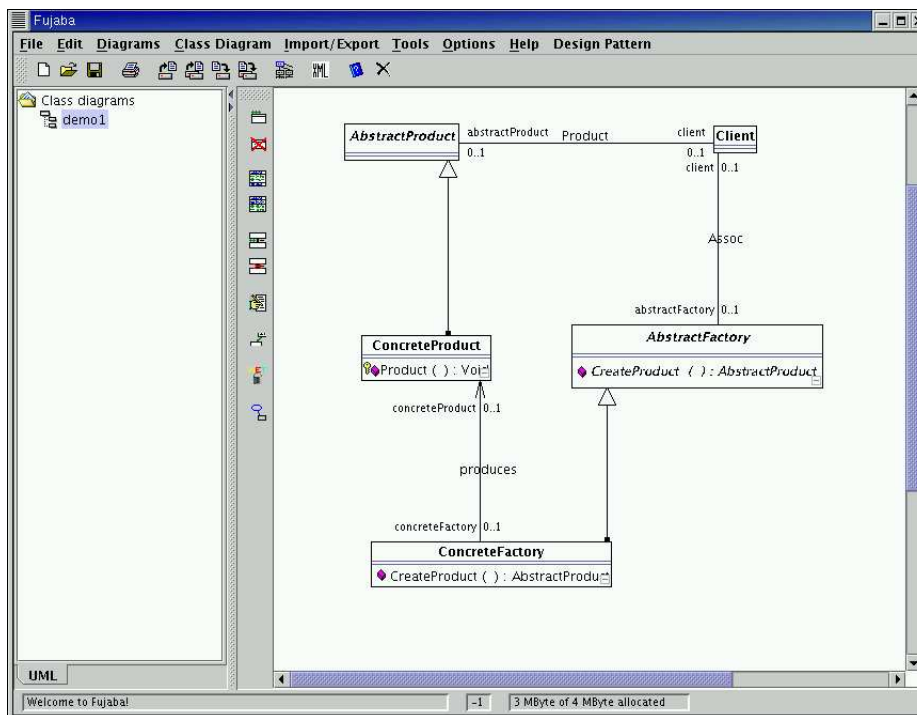


Abbildung 4.1: Das zu exportierende Diagramm

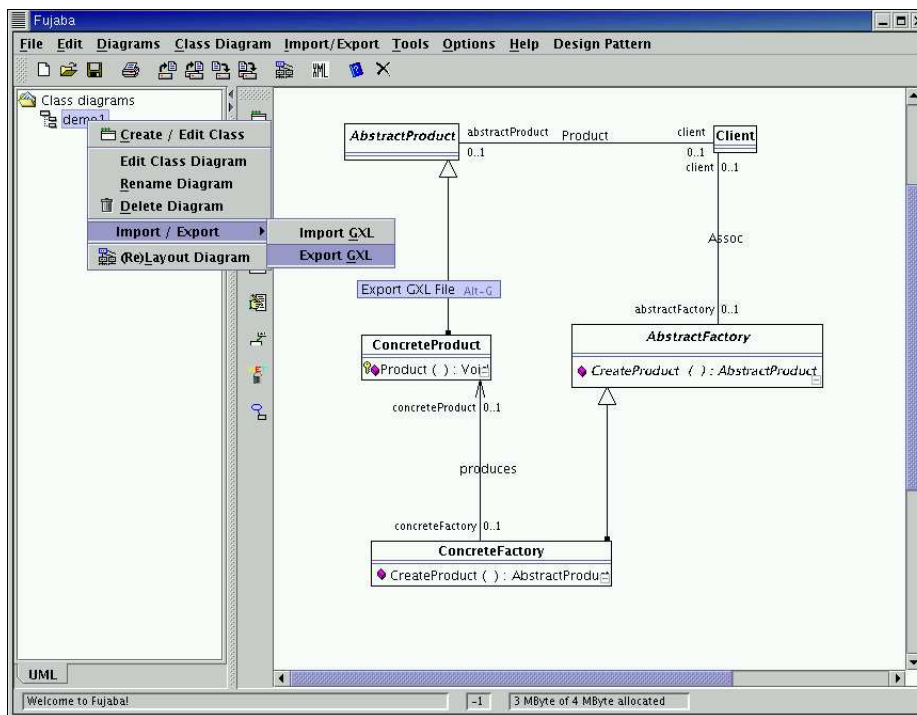


Abbildung 4.2: Menüpunkt Export

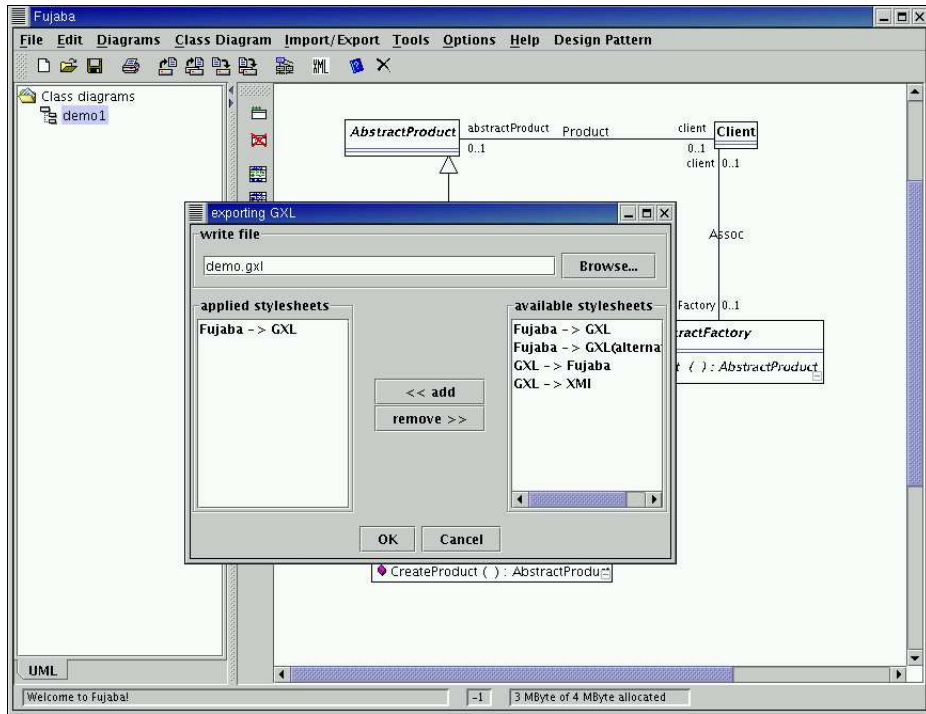


Abbildung 4.3: Der Exportdialog

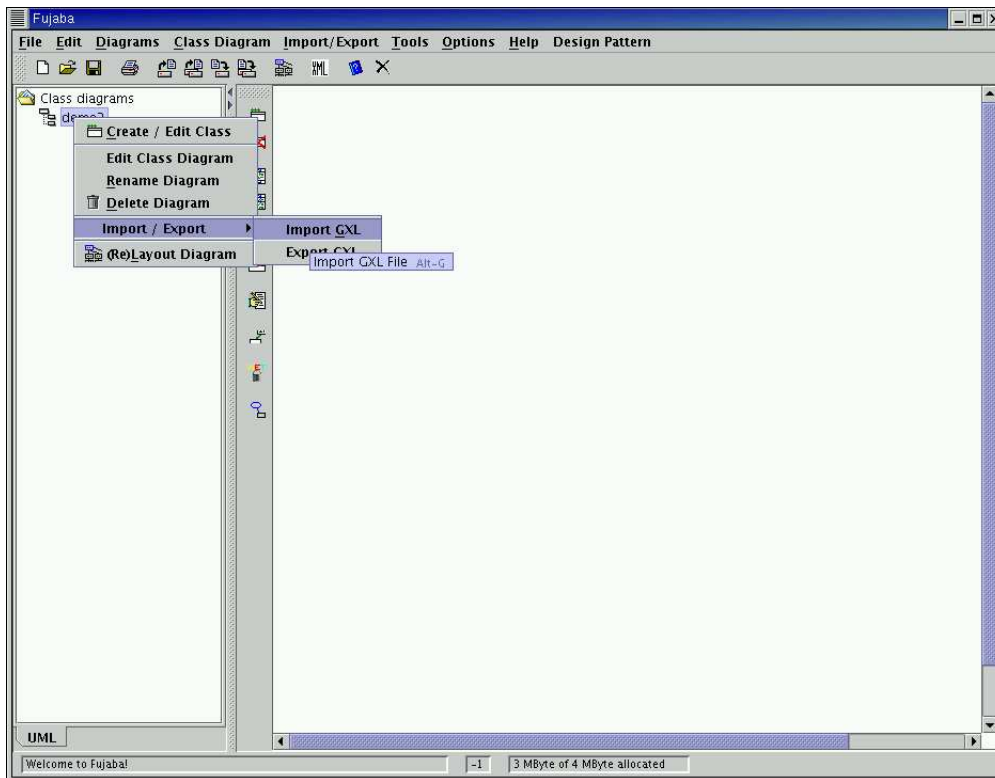


Abbildung 4.4: Menüpunkt Import

4 Beispielsitzung

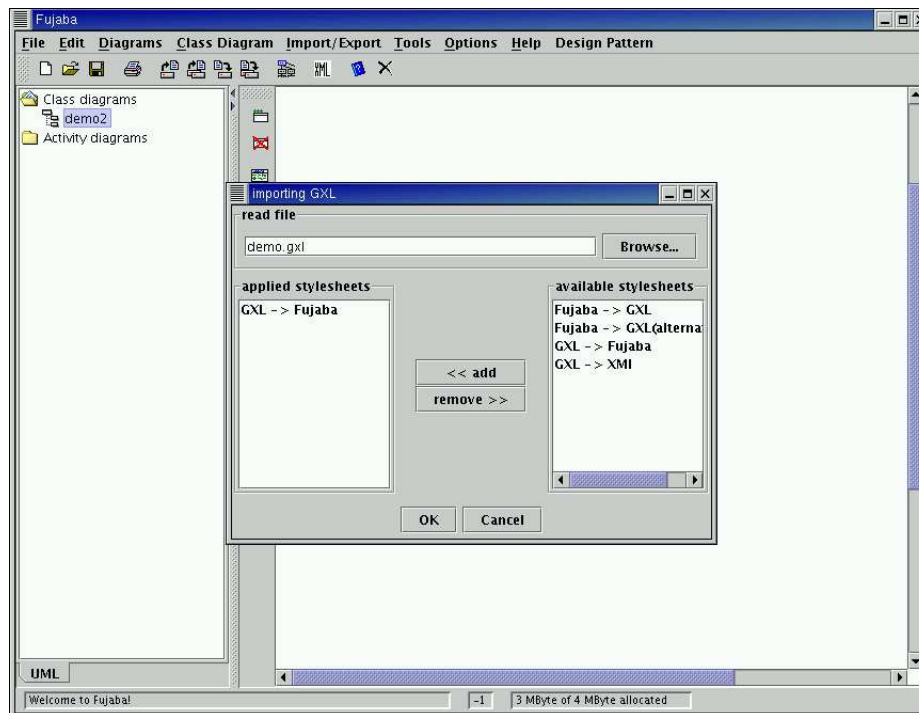


Abbildung 4.5: Der Importdialog

Es erscheint auch hier wieder ein Dialog, der den Benutzer dazu auffordert, die zu importierende Datei und das Format auszuwählen. Siehe dazu Grafik 4.5. Um das in der ersten Phase gespeicherte Klassendiagramm zu wählen, wird der Dateiname *demo.gxl* im Textfeld eingetragen, oder über den Button “Browse” und dem erscheinenden File-Dialog die Datei selektiert. Nachdem die Eingaben bestätigt wurden wird das Diagramm geladen. Die Grafik 4.6 zeigt nun das importierte Diagramm. Dieses kann nun ohne Einschränkung weiterverarbeitet werden. Damit ist der Importvorgang und auch die Beispielanwendung beendet.

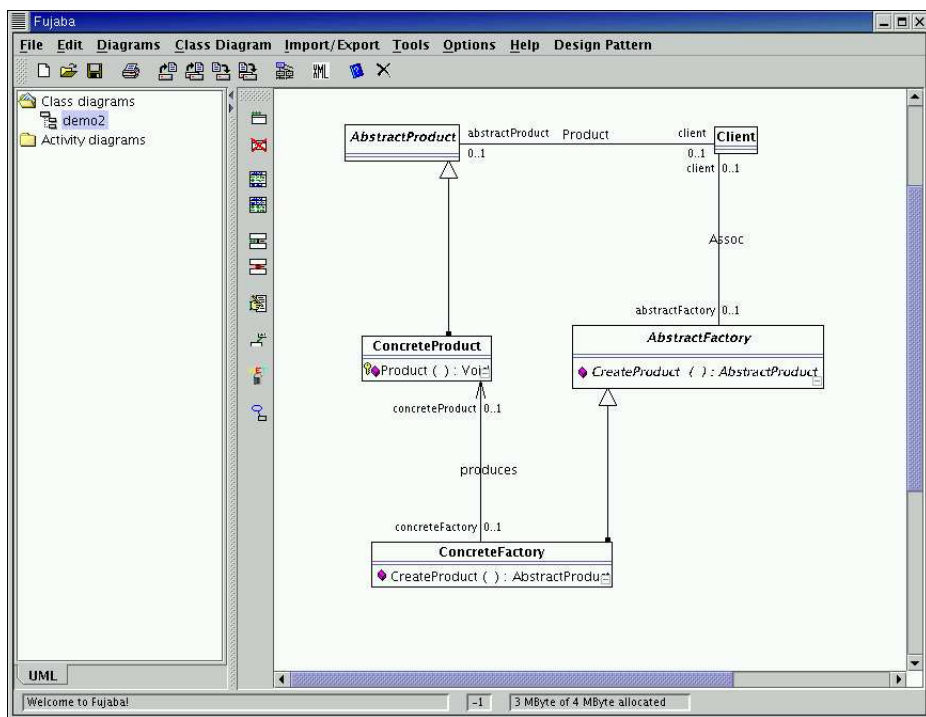


Abbildung 4.6: Das importierte Diagramm

4 Beispielsitzung

5 Zusammenfassung und Ausblick

(Philipp Hoven, Mike Liebrecht)

Durch die Umsetzung der oben aufgeführten Konzepte, ist Fujaba in der Lage Daten im Format GXL zu im- und exportieren. Dadurch ist ein Austausch von (Teil-)Diagrammen zwischen verschiedenen Projekten Fujabas möglich. Auch ist durch die unterstützten Formate der Datenaustausch mit anderen UML-Werkzeugen möglich, da man nicht mehr auf FPR als Ausgabeformat festgelegt ist.

Der neu konzipierte Speichermechanismus ist in der Lage einzelne (Teil-)Diagramme aus dem abstrakten Syntaxgraph herauszulösen und diese für eine weitere Verarbeitung zur Verfügung zu stellen. Der Lademechanismus ermöglicht es Fujaba bestehende Daten von außerhalb in den abstrakten Syntaxgraph einzufügen. Auf diese neuen Fähigkeiten aufbauend ist es nun möglich eine Copy&Paste-Funktionalität zu implementieren.

Der neue Lademechanismus beinhaltet in seiner jetzigen Form einen Merge-Algorithmus, der festlegt wie die zu importierenden Daten in das bestehende Projekt einzubinden sind. Dieser kann dahingehend erweitert werden, dass unterschiedliche Vorschriften für das Einfügen in die verschiedenen Diagrammart verwendet werden können. Im Rahmen der Projektgruppe “Entwurfsunterstützung von verteilten Multimedia Anwendungen mit Hilfe von Design Pattern” ist eine solche Erweiterung für Klassendiagramme umgesetzt worden.

Durch die Trennung von Lade- und Speichermechanismus und Datenformat besteht keine direkte Abhängigkeit mehr zwischen ihnen. Dadurch ist es möglich eine der Seiten zu verändern oder erweitern, ohne dass sich diese Veränderungen auf die andere Seite auswirken. Voraussetzung dabei ist lediglich, dass nach wie vor FujabaXML als Schnittstelle unterstützt wird.

Abbildungsverzeichnis

2.1	Abstrakter Syntaxgraph (entnommen aus [FNT98])	7
2.2	Ablauf des Speichervorgangs	10
2.3	Ladevorgang eines UML-Klassendiagramms	19
2.4	Methoden der Klasse <i>GXLImport</i>	20
2.5	Statische Verknüpfung <i>GXLImport</i> , <i>UMLClassDiagramMerger</i>	21
2.6	Ablauf Speichervorgang	23
2.7	Die Klasse <i>GXLFilter</i>	24
2.8	Funktionsweise Filterkonzept	25
3.1	Ablauf des Transformationsprozesses bei Im- und Export	29
3.2	Dialog zum Im- und Export von GXL-Dateien	44
3.3	Übersicht über die am Export beteiligten Klassen	47
3.4	Die Klasse <i>FileSaver</i>	48
3.5	Übersicht über die am Import beteiligten Klassen	54
3.6	Statische Struktur der Klassen <i>Validator</i> , <i>FXMLTree</i> und <i>FXMLNode</i>	57
4.1	Das zu exportierende Diagramm	62
4.2	Menüpunkt Export	62

Abbildungsverzeichnis

4.3	Der Exportdialog	63
4.4	Menüpunkt Import	63
4.5	Der Importdialog	64
4.6	Das importierte Diagramm	65

Literaturverzeichnis

[XIG] Daniel Volk, XIG - An XSLT-based XMI2GXL Translator,
<http://ist.unibw-muenchen.de/GXL/volk/index.html>, 17.06.2002

[MOF] Object Management Group, Meta-Object Facility,
<http://www.omg.org/technology/documents/formal/mof.htm>, 17.06.2002

[JAPI] Sun Microsystems Inc., Java 2 Platform, Standard Edition,
<http://java.sun.com/j2se/1.4/docs/api/>, 17.06.2002

[DOM] World Wide Web Consortium (W3C), Document Object Model (DOM)
Level 2 Core Specification, Version 1.0,
<http://www.w3.org/TR/DOM-Level-2-Core/>, 30.07.2002

[GXL] Ric Holt/Andy Schürr/Susan Elliott Sim/Andreas Winter, The GXL
Homepage, <http://www.gupro.de/GXL>, 17.06.2002

[HW] R. Holt/A. Winter, A Short Introduction to the GXL Software Exchange
Format(2000), <http://www.gupro.de/winter/Papers/holtwinter2000.pdf>,
17.06.2002

[HWS00] R. Holt/A. Winter/A. Schürr, GXL: Towards a Standard Exchange
Format(2000),
<http://ist.unibw-muenchen.de/People/schuerr/download/WCRE.pdf>, 17.06.2002

[Kay00] Michael Kay: XSLT - Programmer's Reference, Wrox Press, 2000

[WKR] A. Winter/B. Kullbach/V. Riediger, An Overview of the GXL Graph
Exchange Language(2001),
<http://www.uni-koblenz.de/ist/WSR2001/papers/WinterKullbachRiediger.pdf>,
17.06.2002

[Wi01] A. Winter, GXL: Graph Exchange Language(2001),
<http://www.gupro.de/winter/Papers/winter2001.pdf>, 17.06.2002

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

[HoLi01] Philipp Hoven und Mike Liebrecht, Definition von Dokumenttypen: DTDs und XMLSchema(2001), <http://www.upb.de/cs/ag-taentzer/Dtdxsd.pdf>, 17.06.2002

[SWZ99] A. Zündorf/A.Schürr/A.J. Winter, Story driven modeling, Technical Report Universität Paderborn, August 1999

[XML] W3C XML Working Group, Extensible Markup Language (XML) Version 1.0 (W3C Recommendation), <http://www.w3c.org/XML/>, 17.06.2002

[XSLT] World Wide Web Consortium (W3C), eXtensible Stylesheet Language: Transformation(XSLT) Version 1.0 (W3C Recommendation), <http://www.w3.org/TR/xslt>, 17.06.2002

[XPath] World Wide Web Consortium (W3C), XML Path Language, (W3C Recommendation), <http://www.w3.org/TR/xpath>, 17.06.2002

[Wi01a] Andreas Winter, Exchanging Graphs with GXL, Universität Koblenz-Landau, 2001