Universität Koblenz-Landau
Abteilung Koblenz
Fachbereich Informatik
Institut für Softwaretechnik

Entwurf und Implementation eines erweiterten Hyperspace-Modells zur Trennung der Belange

STUDIENARBEIT
im Diplom-Studiengang Informatik

Sascha Berkessel

1. März 2010

Betreuer:
Prof. Dr. Jürgen Ebert
Dipl.-Inform. Hannes Schwarz

Erklärung

Ich erkläre, dass ich die vorliegende Arbeit selbstständig verfasst, keine anderen als die angegebenen Hilfsmittel benutzt und alle Stellen, die anderen Werken dem Sinn oder Wortlaut nach entnommen sind, unter Angabe der Quelle kenntlich gemacht habe.

Koblenz, 1. März 2010

Sascha Berkessel

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Seit Jahrzehnten wird versucht, im Softwareentwicklungsprozess eine saubere, d.h. möglichst vollständige Trennung der Belange zu erreichen. In den meisten Fällen ist diese Trennung jedoch unvollständig.

Daniel Lohmann hat, basierend auf einer Idee von Harold Ossher und Peri Tarr, einen Ansatz geschaffen, das Problem der unvollständigen Trennung der Belange zu lösen. Diese Studienarbeit versucht, Lohmanns Idee für ein existierendes Software-Modellierungstool, den *Enterprise Architect*, in der Praxis anwendbar zu machen.

Nachfolgend werden die Motivation hinter sowie das Ziel und der Aufbau dieser Studienarbeit näher erläutert.

1.1 Motivation

Wie in allen ingenieurwissenschaftlichen Disziplinen, spielt auch in der Softwareentwicklung das Prinzip der *Trennung der Belange* (Separation of Concerns) eine zentrale Rolle: Die begrenzte Aufnahmefähigkeit des Menschen zwingt uns zu einer Zerlegung komplexer Zusammenhänge in voneinander unabhängige Bestandteile. Nur durch das Verständnis der einzelnen Bestandteile und des Zusammenwirkens zwischen diesen wird es dem Menschen ermöglicht, den jeweiligen gesamten Kontext zu verstehen. Man spricht hier auch von *„divide et impera“* - *„teile und herrsche“*.

Eine Vielzahl verschiedener Ansätze versucht, diesem Prinzip im Softwareentwicklungsprozess Rechnung zu tragen. In den meisten Fällen gelingt dies jedoch nur unzureichend. Auch der Ansatz der *Aspektorientierten Programmierung* kann das Prinzip der Trennung der Belange nicht vollständig umsetzen.

Mit dem Hyperspace-Modell stellten Harold Ossher und Peri Tarr am Ende der 90er Jahre erstmals ihr *Hyperspace-Konzept* einschließlich des APIs *Hyper/J* vor ([OT99], [OT00]), das eine nahezu vollständige und gleichzeitige Trennung der Belange ermöglicht. In der Arbeitsgruppe Softwaretechnik der Universität Koblenz-Landau wurde dieses Konzept im Jahr 2002 von Daniel Lohmann im Rahmen seiner Diplomarbeit *Mehrdimensionales Trennen der Belange im Softwareentwurf* ([Loh02]) untersucht. Daraus resultierte ein *erweitertes Hyperspace-Modell*, welches die Kritikpunkte am ursprünglichen Modell zu verbessern versucht.

1.2 Ziele der Studienarbeit

Im Rahmen dieser Studienarbeit soll das Software-Modellierungstool *Enterprise Architect*[1] von *Sparx Systems* so erweitert werden, dass die Trennung der Belange entsprechend dem erweiterten Hyperspace-Modell von Lohmann ermöglicht wird.

Daneben soll auch die Bestimmung so genannter *Slices* ermöglicht werden. Bei einem Slice handelt es sich um die Menge aller Belange und syntaktischen Einheiten, welche mit einem oder mehreren Belangen direkt oder transitiv verknüpft sind. Die Idee der Bestimmung solcher Slices ist *graphenorientiert*: Die transitive Hülle der ausgewählten Belange und der mit diesen verknüpften Belange wird bestimmt. Dies kann auf bestimmte Kantenarten, welche die Arten der Verknüpfung von Belangen widerspiegeln, eingeschränkt werden.

Bevor dies implementiert werden kann, muss untersucht werden, ob die Funktionalität mit den im Enterprise Architect gegebenen Möglichkeiten realisiert werden kann. Ist dies nicht der Fall, so müssen Modelle aus dem Enterprise Architect zunächst mit Hilfe von *JGraLab*[2] in Graphen überführt werden, um anschließend die Slices unter Anwendung der Graphenanfragesprache *Graph Repository Query Language (GReQL)*[3] ([KW99]) zu bestimmen.

Die Funktionalitäten sollen in einem gemeinsamen Software-Baustein realisiert werden, der sich konform zu den für den Enterprise Architect vorgegebenen Schnittstellen verhalten muss und keine Konflikte im Programmablauf verursachen darf.

1.3 Aufbau der Studienarbeit

Da zum Verständnis der Studienarbeit das Wissen über die Hintergründe essentiell ist, wird das Kapitel *2 Theoretische Grundlagen* zunächst einen Überblick über die Begrifflichkeiten sowie die verschiedenen Konzepte zur Trennung der Belange geben. Hierzu wird zuerst ein Überblick über den Begriff *Belang* (2.1.1) und das Prinzip der *Trennung der Belange* (2.1.2) gegeben. Im Anschluss werden in 2.2 das Paradigma der *Aspektorientierten Programmierung* sowie in 2.3 das *Hyperspace-Modell* von Ossher und Tarr vorgestellt. Danach wird in 2.4 das *erweiterte Hyperspace-Modell* von Lohmann, welches den theoretischen Kern dieser Studienarbeit darstellt, erläutert. Abschließend folgt eine Diskussion des Begriffs *Slice* (2.5).

Das Kapitel 3 *Praktische Grundlagen* befasst sich in 3.1 mit dem Modellierungstool *Enterprise Architect* von Sparx Systems und dem in der Arbeitsgruppe Softwaretechnik entwickelten API *JGraLab* (3.2). Neben der Funktionalität von Enterprise Architect werden hier vor allem die angebotenen Schnittstellen und das API für Java beleuchtet, auf deren Basis die Erweiterung des Tools implementiert werden wird. JGraLab wird ebenfalls detailliert dargestellt, da dieses API für Java Möglichkeiten zur Erzeugung von Graphen und deren Überführung in für den Enterprise Architect

[1] Internet: `http://www.sparxsystems.com/products/ea/index.html`
[2] Internet: `http://jgralab.uni-koblenz.de/`
[3] Internet: `http://www.uni-koblenz-landau.de/koblenz/fb4/institute/IST/AGEbert/MainResearch/Graphentechnologie/GReQL`

geeignete Dateien bietet, welche für die Erstellung von Slices von Bedeutung sind. In diesem Zusammenhang wird in 3.3 auch das von Elmar Brauch im Rahmen seiner Studienarbeit ([Bra07]) entwickelte Tool zur Überführung von UML-Modellen zwischen dem Enterprise Architect und JGraLab vorgestellt. Die Vorstellung der *Graph Repository Query Language (GReQL)* in 3.4 schließt das Kapitel über die praktischen Grundlagen der Studienarbeit ab.

Im Anschluss leitet das Kapitel 4 *Anforderungen und Entwurf* das Hauptthema dieser Studienarbeit, den Entwurf und die Entwicklung der Erweiterung des Enterprise Architect, ein. Hier werden in 4.1 zunächst die möglichen Einsatzszenarien und Anwendungsfälle vorgestellt, ehe 4.2 die Anforderungen an die zu entwickelnde Software definiert. Anschließend wird in 4.3 erläutert, wie das erweiterte Hyperspace-Modell in Enterprise Architect umgesetzt ist. Die für die Funktionalität der zu entwickelnden Software nötigen Änderungen am Überführungstool werden in 4.4 vorgestellt, ehe die Erläuterungen zur Architektur der zu entwickelnden Software in 4.5 das Kapitel abschließen.

Das folgende Kapitel 5 *Hynalysis im Einsatz* zeigt anhand eines Beispiels die Anwendung und Funktionsweise der neu entwickelten Software auf. In 5.1 wird zunächst das Beispielszenario beschrieben. Im Anschluss werden in 5.2 die für die Nutzung der neu entwickelten Komponenten nötigen Einstellungen im Enterprise Architect vorgestellt. Schließlich zeigt 5.3 die Nutzung der Software im Kontext des Anwendungsbeispiels.

Im Kapitel 6 *Ergebnisse* wird in 6.1 zunächst der Funktionsumfang der entwickelten Software vorgestellt und dann untersucht, inwieweit die zu Beginn gestellten Anforderungen erfüllt werden konnten. Basierend darauf erfolgt eine Bewertung der entwickelten Software. Anschließend werden in 6.2 Anregungen für zukünftige Weiterentwicklungen der Software gegeben und ein Ausblick auf die sich ergebenden Möglichkeiten geboten. 6.3 schließt die Studienarbeit mit einer Zusammenfassung ab.

2 Theoretische Grundlagen

Um eine dem Thema dieser Studienarbeit gerecht werdende Software zu entwickeln und zu implementieren, ist es unerlässlich, hinreichende Kenntnisse über die theoretischen Grundlagen der multidimensionalen Trennung der Belange im allgemeinen und das erweiterte Hyperspace-Modell im speziellen zu erwerben. Dieses Kapitel soll die elementaren Inhalte erläutern.

Zunächst werden die Begrifflichkeiten *Belang* sowie *Trennung der Belange* vorgestellt. Darauf folgen Erläuterungen zur *Aspektorientierten Programmierung*, zum *Hyperspace-Modell* von Ossher und Tarr und schließlich zum *erweiterten Hyperspace-Modell* von Lohmann.

2.1 Begrifflichkeiten

Wird in der Softwaretechnik vom Prinzip der Trennung der Belange gesprochen, so denken die Beteiligten gewöhnlich an David L. Parnas und Edsger W. Dijkstra, welche in diesem Zusammenhang maßgeblichen Einfluss hatten. Zwar besteht oft ein intuitives Verständnis der Begrifflichkeiten, doch bedarf es einer konkreten Eingrenzung, um das Prinzip selbst und die zu dessen Verwirklichung entwickelten Modelle zu durchdringen.

2.1.1 Der Belangbegriff

Was genau ist gemeint, wenn wir von *Concern*, also *Belang* sprechen? Die Beantwortung dieser Frage ist nicht so trivial, wie es auf den ersten Blick scheint. Tatsächlich ist der Begriff des Belangs nur unscharf abgegrenzt.

Zur Näherung bietet sich ein Blick in ein Wörterbuch an. So gibt *Oxford Advanced Learner's Dictionary* folgende Erläuterung zum Begriff *concern*: *„a thing that is important or interesting to sb“* ([Cro99], S. 228), also *„eine Sache, die für jemanden wichtig oder interessant ist.“*

Lohmann verweist in [Loh02] (S. 5) auf den Artikel *Aspects, Concerns, Subjects, Views, ...* von Rich Hilliard. Laut diesem drückt ein Belang *„ein spezifisches Interesse an einem Thema, bezogen auf ein bestimmtes System oder eine andere Angelegenheit“* aus: *„A concern expresses a specific interest in some topic pertaining to a particular system of interest (or other subject matter).“* ([Hil99], S.1)

Beide Quellen, Wörterbuch und Artikel, sind in ihrer Aussage prinzipiell ähnlich, wobei Hilliard den Belangbegriff im Hinblick auf die Softwareentwicklung etwas näher konkretisiert. Daher sei zusammengefasst: Belange spiegeln in der Softwareent-

wicklung die Intentionen von Stakeholdern, also deren individuelle Interessen und Anforderungen, wider.

Belange können während des gesamten Softwarelebenszyklus auftreten. Sie werden in unterschiedlichen Arten oder Gestalten dargestellt.

2.1.2 Trennung der Belange

Als *Trennung der Belange* bezeichnet man das unter Betrachtung bestimmter Belange erfolgende Zerlegen eines (Software-)Systems in *Module*, *„also in wohl definierte, funktional abgeschlossene und überschaubare Einheiten mit minimaler Kopplung und Redundanz“* ([Loh02], S. 1). Parnas und Dijkstra beschreiben dies als intentionale und lokale Beschreibung der Elemente eines Systems ([Loh02]).

In der Praxis werden zur Trennung der Belange verschiedene *Dekompositionstechniken* verwendet. Eine der bekanntesten ist die *objektorientierte* Dekomposition: Man zerlegt ein System in *Klassen* und ihre *Beziehungen* zueinander.

Wendet man zur Trennung der Belange eine bestimmte Dekompositionstechnik an, so erhält man eine bestimmte *Sicht* auf ein Problem, eine *Dimension*. Der Begriff Dimension wird in diesem Zusammenhang erstmals von Harold Ossher und Peri Tarr verwendet, z.B. in [OT00] (S. 2): *„We refer to a kind of concern, like class [...], as a dimension of concern.“* Anders ausgedrückt enthält eine Dimension alle Belange einer bestimmten Art. Geht man also im Beispiel der objektorientierten Dekomposition davon aus, dass jede Klasse einen Belang repräsentiert, so ist die gemeinsame Dimension aller von einer Klasse repräsentierten Belange die Dimension *Klassen*. Ossher und Tarr führen weiter aus, dass Trennung der Belange die Dekomposition von Software anhand einer oder mehrerer Dimensionen umfasst.

Ein Belang kann in mehreren Dimensionen liegen, ebenso kann eine Dimension mehrere Belange betreffen. Die Betrachtung eines Belangs anhand einzelner Dimensionen führt zu verschiedenen Ansichten, aus denen Aktivitäten, Ziele und Gestaltungsgrundlagen hervorgehen. In gängigen Dekompositionstechniken erfolgt die Zerlegung häufig entlang einer dominanten Dimension. Dadurch können sich Belange über mehrere aus der Dekomposition hervorgehende Elemente verteilen, obwohl sie bei Anwendung einer anderen Dekompositionstechnik völlig problemlos und sehr gut zu kapseln wären. Dies stellt eine Verletzung des Prinzips der Trennung der Belange dar. Ossher und Tarr bezeichnen dieses Problem in [OT99] und [OT00] als *Tyrannei der dominanten Dekomposition*.

Da sich je nach Kontext die Menge der betrachteten Belange und Dimensionen ändern kann, kann auch eine Dekomposition nicht anhand einer einzelnen Dimension erfolgen. Stattdessen ist vor dem Hintergrund der grundsätzlich mehrdimensional verlaufenden Softwareentwicklung eine *simultane multidimensionale Trennung der Belange* nötig. Dabei handelt es sich nach [OT00] (S. 3) um eine Trennung der Belange mit

- *beliebig vielen Dimensionen* von Belangen,
- *simultanem Trennen der Belange* entlang dieser Dimensionen,

- der Möglichkeit, neue Belange und Belangdimensionen *dynamisch* (d.h. wann immer sie im Softwarelebenslauf auftauchen) zu handhaben, und
- *überlappenden und interagierenden Belangen* - völlig unabhängige Belange sind eine Seltenheit.

2.2 Aspektorientierte Programmierung

Dieses Kapitel beruht vollständig auf dem dritten Kapitel aus [Loh02]. Es soll bewusst nur einen kurzen Überblick über die Bereiche bieten, welche im Hinblick auf die in den folgenden Kapiteln beschriebenen Konzepte von Bedeutung sind.

Neben der Tyrannei der dominanten Dekomposition und den damit verbundenen Verletzungen des Prinzips der Trennung der Belange treten bei der Objektorientierten Programmierung (OOP) zwei weitere, miteinander in Zusammenhang stehende Probleme auf: Vererbungsanomalien und Code Tangling.

Als *Vererbungsanomalien* bezeichnet man Szenarien, in denen bei Einfügen einer zusätzlichen simplen Eigenschaft in eine Klasse aus einer bestehenden Vererbungshierarchie eine Vielzahl an Methoden überschrieben werden muss. Von *Code Tangling* spricht man, wenn zum Einfügen oder Realisieren einer zusätzlichen Eigenschaft an vielen Stellen jeweils wenige Codezeilen in bestehenden Quellcode eingebracht werden müssen.

Auf solche Probleme stoßen Softwareentwickler überall dort, wo die Sicherstellung bestimmter Funktionalitäten, Qualitäts- oder Sicherheitsanforderungen sich quer über eine Vielzahl verschiedener Module des Systems erstreckt. Entsprechend werden derlei Anforderungen auch als *querschneidende Belange* bezeichnet.

Die dargelegten Schwächen der Objektorientierung führten zur Entstehung der *Aspektorientierten Programmierung (AOP)*. AOP sollte dabei von Beginn an kein Ersatz, sondern eine Ergänzung der OOP sein.

2.2.1 Aspekte und Konzepte

Wie schon in 2.1 für den Belangbegriff festgestellt, so muss vor der näheren Erläuterung der Ideen der AOP auch hier eine Begriffsklärung erfolgen: Was versteht man unter *Aspekt*? [Loh02] unterscheidet hier zwischen einem umgangssprachlichen und einem technischen Aspektbegriff. Der *umgangssprachliche Aspektbegriff* beruht auf dem allgemeinen Verständnis der Begriffs Aspekt als Blickwinkel oder Gesichtspunkt, wird aber für jegliche Anforderungen und Eigenschaften genutzt. Im Gegensatz dazu steht der *technische Aspektbegriff*, nach dem ein Aspekt nichts anderes ist als ein querschneidender Belang (d.h. ein Modellierungskonzept, welches andere Modellierungskonzepte quer schneidet).

Daneben ist es wichtig, den Zusammenhang zwischen Konzepten, Fachkonzepten und Aspekten zu erläutern: Als modellierter Belang eines Softwaresystems kann ein *Konzept* ein Fachkonzept (ein funktionaler Bestandteil des Systems) oder ein Aspekt sein. Ein Konzept ist dann ein Aspekt, wenn es mehrere Fachkonzepte berührt.

2.2.2 Realisierung der AOP

Zur Realisierung von AOP werden stets drei Dinge benötigt:

- Zunächst benötigt man eine Sprache, mit der sich Aspekte definieren lassen: eine *Aspektdefinitionssprache*.
- Daneben werden *Realisierungstechniken* benötigt, um die Aspekte kombinieren, manipulieren, zu Fachkonzepten hinzuzufügen und wieder entfernen zu können.
- Schließlich vereinfachen *Werkzeuge* den Umgang mit Aspekten.

Das wohl bekannteste Beispiel für Aspektorientierte Programmierung mit den von der jeweiligen (objektorientierten) Programmiersprache vorgegebenen Mitteln sind die *Entwurfsmuster (Design Patterns)* der *Gang of Four*[1] [GHJV95]. Sie bestehen aus einem Namen, einem Anwendungskontext (Problembeschreibung), einem Lösungsweg und den aus ihrer Anwendung resultierenden Konsequenzen. Dies ermöglicht die Identifikation typischer Entwurfsprobleme und die Lösung derselben auf immer gleiche Weise, *ohne* dass die jeweilige Sprache in irgendeiner Form erweitert werden muss.

Im Hinblick auf das Hyperspace-Modell sind aber diejenigen Ansätze zur AOP interessant, welche Programmiersprachen erweitern. Die gemeinsame Idee dieser Ansätze ist, Aspekte mit Hilfe einer *Aspektdefinitionssprache* zu definieren und zu implementieren. Daneben wird mit einer *Aspektanwendungssprache* festgelegt, wie „die Aspekte mit den Fachkonzepten kombiniert werden sollen“ ([Loh02], S. 23). Zudem gibt es einen *Weaver*, der - wie der Name vermuten lässt - den so erzeugten Aspektcode mit dem Code der Anwendung „verwebt“.

Subject-Oriented Programming

Hier soll im folgenden das *Subject-Oriented Programming (SOP)* beschrieben werden, dessen wichtigste Ideen im Hyperspace-Modell aufgegangen sind. Es befasst sich mit der Problematik verschiedener subjektiver, vom Betrachter abhängiger Perspektiven auf ein Objekt. Diese sollen in geeigneter Form, d.h. unter Erhaltung der semantischen Integrität, zusammengeführt werden.

Jedes SOP-Programm besteht aus mehreren sog. *Subjects*. Ein Subject ist ein Objektmodell, eine Menge von Klassen(fragmenten), welche durch Aggregation, Vererbung etc. in Beziehung zueinander stehen. Jedes Subject stellt eine Sichtweise auf ein Objekt dar. Wenn mehrere Subjects auf denselben Instanzen arbeiten sollen, müssen diese zusammengeführt werden.

[Loh02] verdeutlicht dies am Beispiel einer Katze: Frau Müller hat eine Katze namens Schnurri. Natürlich hat Frau Müller auch eine bestimmte Sicht auf ihr Tier: Schnurri isst gern Dosenfutter, lässt sich gern streicheln - und hat sich in den letzten Tagen oft merkwürdig verhalten. Die Katze muss also zum Tierarzt, der eine eigene, veterinärmedizinische Sicht auf Katzen hat. Dies stellt solange kein Problem dar, wie der Tierarzt (und seine Angestellten, welche dieselbe Sicht auf Katzen haben)

[1] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides

oder Frau Müller mit Schnurri allein sind. Jedoch tritt genau dieses Problem beim ersten Termin auf: Der Tierarzt muss Schnurris Daten (Name, Alter etc.) erheben, was ohne Mitwirkung von Frau Müller unmöglich ist. Weil Schnurri in diesem Fall die Instanz ist, auf der beide arbeiten, müssen Frau Müllers Objektmodell und das des Tierarztes zusammengeführt werden, wobei weder die aus Frau Müllers Sicht wichtigen Gesichtspunkte, noch die des Arztes vernachlässigt werden dürfen.

Im Subject-Oriented Programming werden Subjects durch Anwendung von *Composition Rules* zusammengeführt. Dies sind zum einen *Correspondence Rules*, durch die gleichnamige Klassen, Methoden und Datenelemente zusammengefasst werden. *Combination Rules* kommen bei der Zusammenführung von Methodendefinitionen zum Einsatz: Haben zwei Methoden den gleichen Namen, so kann entweder eine Methodendefinition bevorzugt werden oder beide Definitionen werden miteinander kombiniert. *Correspondence-and-Combination Rules* sind schließlich eine Zusammenfassung von Correspondence Rule und Combination Rule in einer gemeinsamen Regel. In Kapitel 2.3 spielen diese Regeln wieder eine Rolle.

2.3 Das Hyperspace-Modell

Die Tyrannei der dominanten Dekomposition führt im Laufe der Zeit zu einer stark abnehmenden Verständlichkeit des Systems, zieht weitreichende Konsequenzen bei Änderungen im System nach sich und schränkt sowohl die Wiederverwendbarkeit des Systems als auch die Nachvollziehbarkeit von Änderungen ein. Wichtig für eine „saubere“ Dekomposition ist aber die Verwirklichung der so genannten *Ilities*: Durch die Trennung der Belange sollen die Verständlichkeit *(comprehensibility)* erhöht, der Einfluss von Veränderungen am System möglichst weit eingeschränkt *(limited impact of change)* sowie die Nachvollziehbarkeit *(traceability)* und Evolvierbarkeit *(evolvability)* gewährleistet werden.

Ossher und Tarr beschreiben in [OT99] und [OT00] genau diese Ziele und bieten mit ihrem Hyperspace-Modell[2] einen Ansatz, der die Aspektorientierte Programmierung insofern weiterentwickelt, dass er die Idee des Subject-Oriented Programming (s. 2.2.2) von der Implementierungsebene auf den gesamten Softwareentwicklungszyklus erweitert. Sie verwirklichen damit ihre Idee der multidimensionalen Trennung der Belange in jeder Phase der Softwareentwicklung [Loh02].

Dieses Kapitel beschreibt das Hyperspace-Modell und bietet einen kurzen Einblick in die ebenfalls von Ossher und Tarr entwickelte Realisierung *Hyper/J*.

2.3.1 Begrifflichkeiten

Software besteht aus unterschiedlichen *Artefakten*. Dies sind jegliche die Software beschreibenden Dokumente (von Anforderungslisten über Diagramme bis hin zu Quellcode), die in einer jeweils geeigneten so genannten *Artefaktsprache* verfasst sind. Artefakte setzen sich wiederum aus *Units (Grundelementen)* zusammen. Als

[2] Internet: `http://www.research.ibm.com/hyperspace/index.htm`

Unit werden syntaktische Konstrukte einer Artefaktsprache bezeichnet. In der OOP sind dies z.B. Klassen oder Methoden [Loh02].

Ein *Belangraum (concern space)* dient zur Organisation aller Units einer Software. Seine Aufgaben sind die Identifikation, Kapselung und Integration von Belangen:

- Die Auswahl und Benennung der wesentlichen Belange und die Zuordnung der Units zu diesen wird als *Identifikation* bezeichnet.
- Durch die *Kapselung* werden die Units eines einzelnen Belangs so zusammengefasst, dass dieser als eigenständige, von anderen Belangen unabhängige Einheit (first-class entity) genutzt werden kann.
- Die *Integration* von Belangen schließlich ist das Zusammenfügen eigenständiger Belange zu einem Softwaresystem.

Zur Identifikation muss hier verdeutlicht werden, dass diese in zwei Stufen erfolgt: Zunächst müssen die relevanten Belange ausgewählt, benannt und in den Belangraum eingebracht werden. Danach erst können die Units in den Belangraum eingebracht und den Belangen zugeordnet werden.

Darüber hinaus fällt auf, dass sowohl bei Ossher und Tarr als auch bei Lohmann stets bereits Belange und Units existieren, die in den Belangraum eingebracht und zugeordnet werden können. Damit bietet sich der Belangraum als Organisationswerkzeug immer dann an, wenn die Entwicklung eines Softwaresystems bereits bis zu einem gewissen Punkt vorangeschritten ist.

Dies wirft jedoch die Frage auf, wie sich der Belangraum von Anfang an sinnvoll in der Softwareentwicklung einsetzen lässt. Strenggenommen müsste man zuerst Belange definieren und daraufhin entsprechende Units entwickeln. Ein solches Vorgehen ist jedoch kaum sinnvoll, zumal es die bewährten Vorgehensweisen der Softwareentwicklung vollständig auf den Kopf stellen würde. Entsprechend sollte die Annahme bereits gegebener Belange und Dimensionen beibehalten werden.

Die Nutzung des Belangraums könnte im Prozess der Softwareentwicklung nach der Erstellung eines ersten Diagramms erfolgen. Dabei spielt es keine Rolle, um welche spezielle Art von Diagramm es sich handelt. Diagramme beschreiben die Software. Sie sind also eindeutig Artefakte und adressieren Belange, lassen sich folglich in den Belangraum einbringen. Wann immer ein Softwareentwickler ein Diagramm erstellt, adressiert er damit einen oder mehrere Belange. Sind diese benannt, können sie in den Belangraum eingebracht und die Units aus dem Diagramm den Belangen zugeordnet werden.

2.3.2 Der Hyperspace

Ein Hyperspace ist im Modell von Ossher und Tarr ein Belangraum in Gestalt eines mehrdimensionalen Raumes[3]. Sie stellen einen Hyperspace als ein Tupel *(U,M,H)* dar, bestehend aus

[3]Lohmann nennt den Hyperspace in [Loh02] einen *geometrischen* Raum. Dieser Begriff ist in der Mathematik jedoch nicht definiert und findet im Rahmen dieser Studienarbeit keine weitere Verwendung.

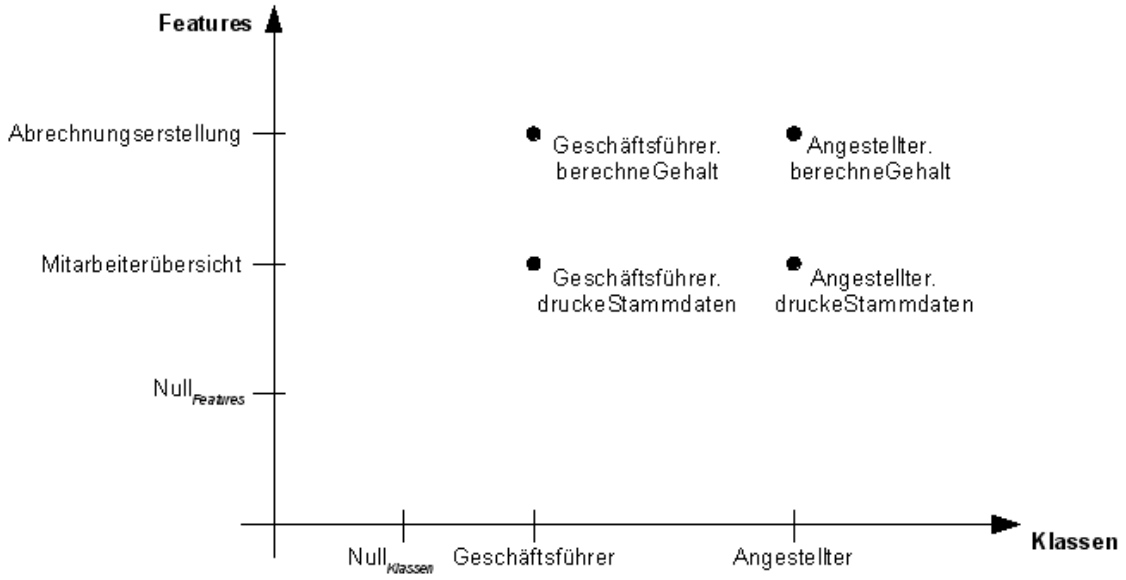

Abbildung 2.1: Beispiel eines Hyperspaces

- einer Menge U von Units,
- einer Belangmatrix M, die die Units aus U den Belangen zuordnet und somit die Identifikation und Kapselung unterstützt, sowie
- einer Menge H von Hypermodules, welche beschreiben, wie aus den Units aus U Komponenten erzeugt werden und somit die Integration von Belangen unterstützen.

Die Belangmatrix

Die Identifikation von Belangen erfolgt mit Hilfe der *Belangmatrix*. Diese repräsentiert ein Kreuzprodukt der vorhandenen Belangdimensionen. Ossher und Tarr setzen diese Matrix einem Koordinatensystem, ihrem Hyperspace, gleich. Jede Achse dieses Belangraums repräsentiert eine Belangdimension, jeder Punkt auf einer solchen Achse entspricht einem Belang aus der jeweiligen Dimension. Jede Unit wird - entsprechend ihrer Position in der Belangmatrix - genau einem Belang in jeder Dimension zugeordnet. Kann eine Unit für eine Dimension keinem Belang zugeordnet werden, so wird sie dort dem so genannten *Nullbelang* zugeordnet, der sich für solche Fälle auf jeder Achse befindet.

Anhand der Koordinaten einer Unit lässt sich somit eindeutig bestimmen, welche Belange in welchen Dimensionen von dieser Unit adressiert werden. Zugleich partitioniert jede Dimension die Menge U der Units im Hyperspace.

Zur Verdeutlichung der Idee des Hyperspace-Modells sei hier ein kleines Beispiel gegeben: Angenommen, es existiert eine (stark vereinfachte) Software zur Personalverwaltung. Softwareentwickler betrachten diese Software wohl vorrangig auf Basis der Klassenstruktur: Die Software besteht aus den zwei Klassen *Geschäftsführer* und

Angestellter. Jede Instanz dieser beiden Klassen enthält die Stammdaten des von ihr repräsentierten Mitarbeiters sowie die beiden Funktionen `berechneGehalt` (zur Berechnung des jeweiligen Monatsgehalts) und `druckeStammdaten` (zur Ausgabe der jeweiligen Stammdaten).

Rufen wir uns nun in Erinnerung, dass Belange immer Ausdruck einer Intention sind, so lassen sich hier Belange aus der Klassenstruktur herleiten: Die resultierenden Belange werden repräsentiert durch die beiden Klassen. Wir erhalten also die Belange *Geschäftsführer* und *Angestellter*. Beide Belange sind von der Art *Klassen*, entsprechend ist auch die Dimension, in der beide Belange liegen, die Dimension *Klassen*. Darüber hinaus können wir auch gleich vier Units erkennen: Jede Methode einer Klasse ist eine Unit. Wir haben also für beide Klassen jeweils die Units *berechneGehalt* und *druckeStammdaten*.

Neben der Entwicklungsabteilung gibt es in Softwarehäusern gewöhnlich auch eine Vertriebsabteilung. Den Mitarbeitern dieser Abteilung ist, genau wie den Kunden, wichtig, was die angebotene Software kann. Diese interessiert, dass die Software Gehaltsabrechnungen erstellen und Stammdaten aller Mitarbeiter ausgeben kann. Solche „Was-kann-die-Software"-Eigenschaften finden sich üblicherweise unter dem Oberbegriff *Features*. Dies ist zugleich die Dimension der Belange, welche sich hier erkennen lassen: *Abrechnungserstellung* und *Mitarbeiterübersicht*. Zusätzliche Units treten hier nicht auf.

Die gefundenen Units, die Belange und die Dimensionen werden nun wie in Abbildung 2.1 auf Seite 19 in den Hyperspace eingebracht: Die beiden Dimensionen *Klassen* und *Features* sind die Achsen des Koordinatensystems. Auf diesen Achsen werden die jeweiligen Belange *Geschäftsführer* und *Angestellter* bzw. *Abrechnungserstellung* und *Mitarbeiterübersicht* eingetragen. Schließlich werden die Units entsprechend ihrer Zuordnung zu den Belangen der beiden Dimension in das Koordinatensystem eingetragen. Die Nullbelange der beiden Dimensionen, $Null_{Klassen}$ und $Null_{Features}$, bleiben leer, da alle Units eindeutig Belangen in beiden Dimensionen zugeordnet werden können.

Hyperslices

Kann die Identifikation von Belangen noch ausschließlich mit Hilfe der Belangmatrix erfolgen, so genügt diese allein schon nicht mehr für die Kapselung von Belangen. Hier setzen die Hyperslices an.

Ein *Hyperslice* ist eine Menge von Units. Im Gegensatz zu Units, zwischen denen oft eine starke Kopplung besteht (z.B. können Methoden andere Methoden aufrufen), sind Hyperslices voneinander unabhängig und können so wie ein eigenständiger Software-Baustein genutzt werden. Dazu müssen sie jedoch die Bedingung der *deklarativen Vollständigkeit* erfüllen: Hyperslices müssen all ihre Referenzen (d.h. jede aufgerufene Methode, jede genutzte Variable etc.) deklarieren.

Zu diesem Zweck führt man so genannte *deklarative Units* ein, die nicht mit einer eigenen Implementation versehen sind, sondern die jeweils verwendete Unit lediglich referenzieren ([OT99], S. 6). Die „eigentliche" Unit mitsamt ihrer Implementation findet sich dann in einem anderen Hyperslice.

Es liegt hier eine Trennung von Deklaration und Implementation vor, bei der alle in einem Hyperslice genutzten Units zwar deklariert sein müssen, jedoch auch in einem anderen Hyperslice implementiert werden dürfen. Die so genannte *Korrespondenz (correspondence)* muss aber im Hinblick auf die Integration gewahrt bleiben: Jede deklarierte Unit muss in wenigstens (s. Hypermodules) einem Hyperslice implementiert sein. Die Korrespondenz wird im Zusammenhang mit Hypermodules noch einmal aufgegriffen.

Betrachtet man die Belangmatrix, so ist erkennbar, dass jeder Belang eine Ebene im Belangraum repräsentiert und so auch eine Menge von Units darstellt. Somit ist jeder Hyperslice auch ein Belang. Entsprechend werden die Hyperslices in eigenen Dimensionen, so genannten *Hyperslice-Dimensionen* gruppiert, die dann in den Hyperspace eingebracht werden. Dass Hyperslices auf diese Weise andere Belange überlappen, ist so gewollt: Hyperslices lassen sich auffassen als Belange, die andere Belange gruppieren.

Der Unterschied zwischen Belangen und Hyperslices besteht lediglich in der deklarativen Vollständigkeit. Daraus ergibt sich, dass jede Unit mindestens einem Hyperslice zugeordnet sein muss.

Hypermodules

Mit Hilfe von *Hypermodules* lassen sich verschiedene Hyperslices zu einem gemeinsamen Hyperslice integrieren. So lassen sich größere Hyperslices bis hin zu lauffähiger Software (die letztlich nichts anderes als ein „allumfassender Hyperslice“ ist) erzeugen. Diesen Prozess bezeichnet man als *Kombination*.

Hypermodules bestehen aus der Menge der zu integrierenden Hyperslices und einer Menge von *Kompositionsbeziehungen*, die festlegen, wie die Hyperslices integriert werden. Für jede Kompositionsbeziehung beschreibt dabei eine *Kombinationsfunktion*, wie die Units aus den verschiedenen Hyperslices zu neuen Units zusammengeführt werden müssen. Damit Hyperslices kombiniert werden können, muss die oben angesprochene Korrespondenz gewahrt sein, die - das sei explizit angemerkt - nicht auf die Ebene des Quellcodes beschränkt ist [Loh02].

Bei der Kombination verschiedener Units kann es vorkommen, dass manche Units gleich in mehreren Hyperslices implementiert sind. Dann muss ein Weg gefunden werden, die verschiedenen Implementationen miteinander zu kombinieren. Hierzu nutzt man Korrespondenz- und Kombinationsregeln (vgl. auch Composition Rules in 2.2.2): *Korrespondenzregeln* fassen zunächst gleichnamige Units zusammen. Anhand der *Kombinationsregeln* wird dann festgelegt, ob die Implementation einer Unit den Vorzug vor der einer anderen Unit erhält oder ob mehrere Implementationen miteinander kombiniert werden. In letzterem Fall werden zusätzliche Spezifikationen festgelegt, beispielsweise eine Ausführungsreihenfolge für gleichnamige Methoden.

Wie Lohmann in [Loh02], S. 46/47 anmerkt, ist es zur Erstellung einer sinnvollen lauffähigen Software natürlich erforderlich, dass die Kompatibilität zwischen den Units eines Hyperslices gegeben ist. Dies kann im Bereich der syntaktischen Kompatibilität weitgehend automatisiert erfolgen. Bezüglich der semantischen Kompatibilität ist aber zumindest mittelfristig noch der Entwickler selbst gefragt.

2.3.3 Hyper/J

Hyper/J ist eine von Ossher und Tarr entwickelte Realisierung des Hyperspace-Modells für Java. Es ermöglicht die Identifikation, Kapselung und Integration von Belangen für Artefakte in Form von Java-Code.
Vollständig auf der Basis von Java entwickelt, arbeitet Hyper/J ausschließlich mit binären Java-Paketen (`.class`/`.jar`). Dies ermöglicht seine Anwendung auf Java-Programme zu jedem Zeitpunkt der Softwareentwicklung. D.h. sowohl, dass Programme von Beginn an mit Unterstützung durch Hyper/J entwickelt werden können, als auch, dass die Anwendung von Hyper/J auf bereits bestehende Programme erfolgen kann - ohne, dass vorher mehrdimensionales Trennen der Belange erfolgt sein muss. Zudem macht die Nutzung von Binärdateien das Vorhandensein von Java-Quellcode obsolet.

Zur Anwendung von Hyper/J werden verschiedene Konfigurationsdateien benötigt:

- In der *Projekt-Spezifikation* wird festgelegt, welche Input-Units im Hyperspace angeordnet werden. Sie definiert den Namen des jeweiligen Hyperspaces, die Units (Klassen, Interfaces, Methoden, Datenelemente [Loh02]), welche der Hyperspace umfasst, sowie die `.class`-Dateien, in welchen die Units enthalten sind.
- Mit den *Concern Mappings* werden weitere Dimensionen und Belange definiert sowie festgelegt, wie die Input-Units diesen zugeordnet werden. Die Größe der Units, welche den verschiedenen Dimensionen und Belangen zugeordnet werden, kann dabei von Variablen bis hin zu ganzen Paketen reichen.
- Die *Hypermodule-Spezifikation/en* schließlich legt/legen fest, welche Hyperslices wie integriert werden. Eine Hypermodule-Spezifikation besteht aus dem Namen des resultierenden Hypermodules, der Menge der zu integrierenden Hyperslices sowie den Integrations-Relationen, die zwischen den Hyperslices und ihren Units gelten. Bei den Integrations-Relationen handelt es sich um die bereits mehrfach erwähnten Kombinations- und Korrespondenz-Regeln (Combination Rules und Correspondence Rules).

Der Ablauf der Anwendung von Hyper/J ist wie in [OT00] beschrieben vorgesehen: Der Entwickler erzeugt die zuvor beschriebenen Konfigurationsdateien. Hyper/J legt zunächst entsprechend der Projekt-Spezifikation einen Hyperspace an und ordnet alle Input-Units den Belangen in der Dimension *Class* zu. Dies ist die Ausgangsdimension für die weitere Trennung der Belange; die Belange in dieser Dimension sind entsprechend die verschiedenen Klassen. Danach erfolgt die Auswertung der Concern Mappings, anhand derer weitere Dimensionen und Belange erzeugt und die Input-Units diesen zugeordnet werden. Schließlich erfolgt die Integration der Hyperslices entsprechend den Hyperspace-Spezifikationen. Der Entwickler kann nun prüfen, ob das Ergebnis seinen Vorstellungen entspricht und durch Modifikationen an den verschiedenen Konfigurationen Schritt für Schritt bis zu dem für ihn optimalen Ergebnis kommen.

Für nähere Informationen zu Hyper/J und seiner Anwendung sei auf [Loh02], Kap. 4.2, [OT00], Kap. 5 sowie die Website zu Hyper/J[4] verwiesen. Im Kontext der vorliegenden Arbeit soll die gegebene Übersicht über Hyper/J genügen.

2.4 Das erweiterte Hyperspace-Modell

Das Hyperspace-Modell von Ossher und Tarr und seine Umsetzung in Hyper/J sind ein erster Schritt hin zur Verwirklichung der multidimensionalen Trennung der Belange. Jedoch gibt es auch hier noch Kritikpunkte.

Hyper/J ist ausschließlich auf die Anwendung auf Java-Klassen ausgelegt. Damit ist es nur in den späten Phasen der Softwareentwicklung (Implementation und Wartung) einsetzbar. Zugleich ist Java die einzige Artefaktsprache, die durch Hyper/J unterstützt wird. Zu anderen Artefakten gehörende Units können somit nicht in den Belangraum eingebracht werden. Darüber hinaus ermöglicht das Hyperspace-Modell weder die Darstellung überlappender und interagierender Belange und Dimensionen noch die Darstellung hierarchischer Beziehungen zwischen Belangen. Die Darstellung überlappender und interagierender Belange ist jedoch eines der Kriterien, die von Ossher und Tarr selbst für das simultane multidimensionale Trennen der Belange definiert worden sind.

LOHMANN stellt in [Loh02] sein *erweitertes Hyperspace-Modell* vor, das die simultane multidimensionale Trennung der Belange über den gesamten Softwareentwicklungszyklus ermöglichen soll. Als zentrale theoretische Grundlage dieser Studienarbeit wird es auf den folgenden Seiten detailliert erläutert.

Die Integration von Units verschiedener Artefaktsprachen in den Hyperspace wirft diverse Probleme auf, die auch von Lohmann in [Loh02], Kap. 5.2 sowie von Lohmann und Ebert in [LE03] verdeutlicht werden:

2.4.1 Der Hyperspace als mehrdimensionaler Raum

Ossher und Tarr verstehen „ihren“ Hyperspace als mehrdimensionalen Raum. Die der Belangmatrix zugedachte Raumeigenschaft erleichtert die formale, algorithmische Arbeit mit Hyperspaces und fördert zugleich eine saubere Dekomposition: Ist die eindeutige Zuordnung jeder Unit zu jeweils genau einem Belang in jeder Dimension schwierig, ist die vorgenommene Trennung noch nicht gründlich genug.

Mit der Raumeigenschaft geht im ursprünglichen Hyperspace-Modell auch eine Trennung von Deklaration und Definition einher. Diese entspricht zwar der Idee vieler Programmiersprachen (welche Artefaktsprachen sind) und eignet sich somit gut für die Trennung der Belange in den späten Phasen des Softwareentwurfs, verträgt sich jedoch nicht mit den in den frühen Phasen eingesetzten Artefaktsprachen wie z.B. denen der *Unified Modeling Language (UML)*. Solche Sprachen trennen Deklaration und Definition nicht, vielmehr erfolgt hier die Definition einer Unit durch die erstmalige Verwendung derselben.

[4] Internet: `http://www.alphaworks.ibm.com/tech/hyperj`

Lohmann nennt hier in [Loh02], S. 68 als Beispiel die Use-Case-Diagramme (Anwendungsfalldiagramme): Die Dimension ist in diesem Fall *Anwendungsfälle*. Jeder Anwendungsfall entspricht einem Belang in dieser Dimension, jeder Akteur einer Unit. Üblicherweise ist ein Akteur an mehreren Anwendungsfällen beteiligt und muss somit mehreren Belangen zugeordnet werden. Dies widerspricht jedoch dem ursprünglichen Hyperspace-Modell, bei dem aufgrund der Raumeigenschaft jede Unit nur genau einem Belang in jeder Dimension zugeordnet werden muss.

Um das Hyperspace-Modell im gesamten Softwareentwicklungszyklus einsetzbar zu machen, gibt Lohmann für sein erweitertes Hyperspace-Modell die Raumeigenschaft auf. Sinnvollerweise kann eine Unit damit mehreren Belangen in jeder Dimension zugeordnet werden.

2.4.2 Übertragung der Begriffe auf das neue Modell

Die für das ursprüngliche Hyperspace-Modell genutzten Begriffe können nicht vollständig für das erweiterte Hyperspace-Modell übernommen werden. Mit dem Wegfall der Raumeigenschaft erübrigen sich die *deklarativen Units*. Entsprechend werden auch die Begriffe der *deklarativen Vollständigkeit* sowie *Hyperslices* nicht mehr benötigt, damit verbunden auch die gesonderten *Hyperslice-Dimensionen*. Darüber hinaus fallen die *Nullbelange* weg: Kann eine Unit für eine bestimmte Dimension keinem Belang zugeordnet werden, wird sie schlicht nicht zugeordnet.

Im erweiterten Hyperspace-Modell werden somit nicht mehr zunächst Hyperslices gebildet, die dann zu Hypermodules gruppiert werden. Hier erfolgt eine direkte Gruppierung von Belangen zu Hypermodules.

2.4.3 Die Trennung zwischen Units und Belangen

Eine *Unit* ist eine Entität einer Artefaktsprache. Anders ausgedrückt stellt eine Unit die konkrete Instanz eines Elementtyps der jeweiligen Modellierungssprache dar. Im Gegensatz dazu ist ein *Belang*, wie in Kapitel 2.1 verdeutlicht, ein intentionales Produkt des Geistes und somit nicht als formales Konstrukt zu verstehen. Belange können jedoch mit Symbolen bezeichnet und so fassbar gemacht werden.

Häufig besteht zwischen Units und Belangen eine 1:1-Beziehung: Eine Unit ist ein syntaktischer Repräsentant des modellierten Belangs. So wird z.B. jede Klasse einer Software dem gleichnamigen Belang in der Dimension *Klassen* im Hyperspace zugeordnet. Die jeweilige Klasse als Unit steht hier für den gleichnamigen Belang. Obwohl es sich nachwievor um zwei getrennte Dinge handelt, besteht in solch einem Fall eine enge semantische Kopplung zwischen Unit und Belang. Man spricht in diesem Zusammenhang von einer *repräsentativen Unit*.

2.4.4 Units aus mehreren Belangdimensionen

Das Hyperspace-Modell unterscheidet implizit zwei verschiedene Arten von Belangdimensionen, die in [Loh02] und [LE03] unterschiedlich bezeichnet werden:

- *Artefaktbasierende Belangdimensionen*, auch *primäre Belangdimensionen* genannt, sind diejenigen Dimensionen, über die die Units direkt in den Hyperspace eingebracht werden.
  Sie beruhen auf *idealen Artefaktsprachen*, also solchen Artefaktsprachen, die jeweils genau eine Dimension von Belangen beschreiben bzw. darstellen. Die betreffenden Belange werden auch als *primäre Belange*, die zugeordneten repräsentativen Units als *primäre Units* bezeichnet. Lohmann geht für sein erweitertes Hyperspace-Modell von idealen Artefaktsprachen aus.
- *Selektierende Belangdimensionen*, auch *sekundäre Belangdimensionen* genannt, sind Dimensionen, über die keine neuen Units in den Hyperspace eingebracht werden, sondern die eine andere Sicht auf den Hyperspace darstellen.
  Bei solchen Dimensionen werden die zugehörigen Belange (sog. *sekundäre Belange*) nicht durch eigene Artefakte repräsentiert. Die bereits vorhandenen Units werden hier auf die sekundären Belange abgebildet, so dass je nach Bedarf eine alternative Trennung der Belange gegeben ist.

Metamodelle

Die Artefaktsprachen in den späteren Phasen der Softwareentwicklung sind gemeinhin *textuelle Sprachen*, z.B. Programmiersprachen. Für die Erstellung von Instanzen solcher Sprachen kann man formale *Grammatiken* nutzen, die genau festlegen, wie eine gültige Instanz der jeweiligen Sprache aufgebaut sein muss.

In den frühen Phasen des Softwareentwicklungsprozesses kommen jedoch überwiegend *visuelle Sprachen* wie UML zum Einsatz. Solche Sprachen können nicht durch Grammatiken beschrieben werden. Hier werden *Metamodelle* genutzt, die die abstrakte Syntax der jeweiligen Artefaktsprache repräsentieren. Jeder Unittyp wird durch eine Metaklasse repräsentiert, jede Beziehung zwischen Unittypen durch eine Meta-Assoziation [LE03].

Stellt man sich für das Beispiel der Software zur Personalverwaltung aus Kapitel 2.3.2 ein UML-Klassendiagramm vor, so lassen sich die Klassen *Geschäftsführer* und *Angestellter* durch eine Metaklasse *Klasse* repräsentieren. Beide Klassen sind dann syntaktische Konzepte dieser Metaklasse. Zusätzlich angenommene Beziehungen *stellt ein* oder *kündigt* zwischen *Geschäftsführer* und *Angestellter* können in diesem Beispiel syntaktische Konstrukte einer Meta-Assoziation *in Relation zu* auf der Metaklasse *Klasse* sein.

Auch textuelle Sprachen lassen sich anhand eines Metamodells beschreiben. Dies ist im Hinblick auf die Trennung der Belange im gesamten Softwareentwicklungsprozess relevant, wie dieses Kapitel im folgenden zeigt.

Belangdimensionen in Hyper/J

Für den Einsatz von Hyper/J bedeutet die Unterscheidung zwischen artefaktbasierenden und selektierenden Belangdimensionen konkret, dass es genau eine artefaktbasierende Belangdimension gibt: die Dimension *Klassen*. Diese ergibt sich aus der Projekt-Spezifikation. Alle anderen Dimensionen in Hyper/J sind selektierende Be-

langdimensionen, die in den Concern Mappings definiert werden. Die eingebrachten Units sind Instanzen der im Java-Metamodell definierten Belangdimension. Das dem Hyperspace in Hyper/J zugrundeliegende Metamodell entspricht dem Metamodell der artefaktbasierenden Belangdimension.

Belangdimensionen im erweiterten Modell

Hier liegt nun die Schwierigkeit der Übertragung auf das erweiterte Hyperspace-Modell: Im Verlauf des Softwareentwicklungsprozesses treten *diverse Artefaktsprachen* auf, denen *verschiedene Metamodelle* zugrundeliegen. Für den Hyperspace müssen also statt einem mehrere Metamodelle berücksichtigt werden, die darüber hinaus nicht vollständig voneinander isoliert sein müssen, so dass auch Kopplungen zwischen den Metamodellen bestehen können. Solche Kopplungen können entsprechend auch zwischen den zugehörigen Units vorkommen. Es ist daher zwingend erforderlich, die verschiedenen Metamodelle zu einem *gemeinsamen Metamodell* zu integrieren, welches dann die Grundlage des Hyperspaces darstellt. Nur so kann der Hyperspace den Anforderungen aller beteiligten Artefaktsprachen gerecht werden.

Ablauf

Statt wie im Falle von Hyper/J das Metamodell des Java-Modells als Metamodell des Hyperspaces zu nutzen und das Java-Modell so in den Hyperspace zu integrieren (*Integration*), je nach Bedarf selektierende Belangdimensionen zu erzeugen und die Units diesen zuzuordnen (*Selektion*) und schließlich die Belange zu Hypermodules zu kombinieren (*Kombination*), müssen bei der Nutzung des erweiterten Modells zusätzliche Schritte durchlaufen werden:

Zunächst erfolgt die nun umfangreichere *Integration*: Das Metamodell eines Modells wird in das Metamodell des Hyperspaces integriert. Eine geeignete artefaktbasierende Belangdimension wird ausgewählt oder erzeugt (falls noch nicht vorhanden). Die Belange und Units werden in den Hyperspace eingefügt sowie die Beziehungen zwischen den neu eingefügten Units und den bereits vorhandenen Units anderer artefaktbasierender Belangdimensionen definiert.

Im Rahmen der *Selektion* werden, wie bei der Arbeit mit Hyper/J, selektierende Belangdimensionen erzeugt und die vorhandenen Units den sekundären Belangen zugeordnet.

Im Anschluss erfolgt die *Kombination*: Hier werden die zu kombinierenden Belange ausgewählt, wobei die Menge der zugehörigen Units nicht zwangsläufig ein gültiges Modell des integrierten Metamodells darstellt. In einem solchen Fall müssen die Units mit Hilfe eines Transformationsprozesses in ein gültiges Modell, das *Ausgabemodell*, überführt werden.

Schließlich folgt ein zusätzlicher Schritt, die *Projektion*: Das Ausgabemodell ist ein gültiges Modell des integrierten Metamodells des Hyperspace, in den seltensten Fällen jedoch auch ein gültiges Modell der Metamodelle der geeigneten Artefaktsprachen. Damit es weiterverarbeitet oder grafisch dargestellt werden kann, muss

das Ausgabemodell also wieder auf Einzelmodelle der jeweiligen Artefaktsprachen abgebildet werden.

2.4.5 Zusammenfassung

Um das ursprüngliche Hyperspace-Modell in allen Phasen der Softwareentwicklung anwendbar zu machen, nimmt Lohmann einige Änderungen vor, die in seinem erweiterten Hyperspace-Modell münden. Zunächst wird die Raumeigenschaft des Hyperspaces aufgegeben, um die manchmal nötige Zuordnung von Units zu mehreren Belangen einer Dimension zu ermöglichen.

Daneben zeigt sich, dass wir zwei verschiedene Arten von Belangdimensionen unterscheiden müssen: Den artefaktbasierenden Belangdimensionen liegen Artefaktsprachen zugrunde. Über diese Dimensionen werden neue Units in den Hyperspace eingebracht. Die selektierenden Belangdimensionen stellen hingegen lediglich eine weitere Sicht auf das Softwaresystem bereit. Ihnen liegt keine Artefaktsprache zugrunde.

Möchte man mehrere artefaktbasierende Belangdimensionen in den Hyperspace einbringen, spielen die Metamodelle der jeweiligen Artefaktsprachen eine entscheidende Rolle. Vor der Integration von Artefakten in den Hyperspace müssen diese unterschiedlichen Metamodelle in ein gemeinsames Metamodell (das des Hyperspaces) überführt werden. Gegenüber der multidimensionalen Trennung der Belange mit Hyper/J ergeben sich dadurch zusätzliche notwendige Schritte im Ablauf der multidimensionalen Trennung der Belange.

2.5 Slices

Wie bereits in 1.2 erwähnt, ist ein Ziel der vorliegenden Arbeit die Bestimmung von *Slices*. Um diese Funktionalität verwirklichen zu können, muss definiert sein, was ein Slice ist. Als Abschluss des Kapitels zu den theoretischen Grundlagen der Studienarbeit wird hier der Slice-Begriff diskutiert und schließlich eine Definition gegeben.

2.5.1 Hintergründe

Der Begriff des *Slicing* wird in der Softwaretechnik in verschiedenen Zusammenhängen genutzt. Dabei spielen die in 2.3.2 vorgestellten *Hyperslices* eine untergeordnete Rolle. Von Bedeutung ist hier vor allem das *Program Slicing*, welches erstmals in den frühen 1980er Jahren von Mark Weiser vorgestellt wurde [Wei81] und unter anderem auch bei Hannes Schwarz in [Sch06] Anwendung findet.

Die Idee des Program Slicing ist, ein Programm auf eine minimale Form zu reduzieren, welche dasselbe Verhalten erzeugt wie ein gegebener Programmausschnitt. Die Vergleichbarkeit des Verhaltens wird durch ein so genanntes *Slicing-Kriterium* gewährleistet.

In der Praxis legt man zunächst das Slicing-Kriterium fest. Dies kann z.B. eine Menge von Variablen sein. Im Anschluss werden all die Teile eines gegebenen Programms bestimmt, die eine Wirkung auf das Slicing-Kriterium besitzen oder auf die sich das Slicing-Kriterium auswirkt. Hierunter versteht man z.B. Methoden, die den Wert einer Variablen aus dem Slicing-Kriterium verändern, oder Variablen, die durch Änderungen am Slicing-Kriterium ebenfalls verändert werden.

Dabei wird unterschieden zwischen *Rückwärts-Slicing*, bei dem die Anweisungen und Variablen bestimmt werden, die eine Wirkung auf das Slicing-Kriterium haben, und *Vorwärts-Slicing*, bei dem die Anweisungen und Variablen bestimmt werden, auf die das Slicing-Kriterium eine Wirkung hat. Die resultierenden Slices werden entsprechend als *Rückwärts-Slices* (auch *rückwärtsgerichtete Slices*) bzw. *Vorwärts-Slices* (auch *vorwärtsgerichtete Slices*) bezeichnet.

Aus der Sicht der Softwarearchitektur findet das *Architectural Slicing* Anwendung, wie es z.B. in [Zha97] von Jianjun Zhao vorgestellt wird. Zhao erstellt einen *Software Architectural Dependence Graph*, also einen Abhängigkeitsgraphen für eine Softwarearchitektur, und wendet dann die Methoden des Program Slicing darauf an.

Beide Arten des Slicing haben zum Ziel, die Wartbarkeit und das Testen von Software zu erleichtern bzw. zu verbessern. Zugleich verbessern sie die Verständlichkeit von Software.

Die Arbeitsgruppe Softwaretechnik der Universität Koblenz-Landau verwendet Slicing z.B. im Rahmen des *ReDSeeDS*-Projekts[5]. In [BERS08] stellen Bildhauer et al. ein Konzept zur Verwaltung von Softwareartefakten und ihrer Beziehungen untereinander in graphenbasierten Repositories vor, mit dessen Hilfe die Wiederverwendung von Software gefördert werden soll. Die Artefakte werden in einem sog. *Artefakt-Repository* hinterlegt. Um die Beziehungen und Abhängigkeiten zwischen den verschiedenen Artefakten verwalten zu können, müssen die Artefakte und ihre Beziehungen über ein gemeinsames Metamodell abstrahiert werden. Diese Abstraktionen werden in einem sog. *Fakten-Repository* abgelegt.

Anhand des Fakten-Repositorys können Analysen zur Ähnlichkeit von verschiedenen sog. *Software Cases*, dies sind Mengen von auf bestimmten Anforderungen basierenden Artefakten, durchgeführt werden. Das Ergebnis sind diejenigen Anforderungen, welche für die verglichenen Software Cases identisch sind. Sie bilden das Slicing-Kriterium, für das dann mittels Slicing alle für einen noch in der Entwicklung befindlichen Software Case verwendbaren Teile des bzw. der bereits fertigen Software Cases bestimmt werden. Der Slicing-Prozess nutzt dabei die im Fakten-Repository gespeicherten Abhängigkeitsinformationen.

2.5.2 Slicing im Hyperspace-Kontext

Die oben erläuterten Ideen zum Slice-Begriff lassen sich in abgewandelter Form auch auf das erweiterte Hyperspace-Modell übertragen. Ausgehend von einem gewählten Belang, der hier das Slicing-Kriterium darstellt, werden all die Belange und Units

[5] Internet: `www.redseeds.eu`

ausgewählt, die direkt oder transitiv mit diesem Belang verbunden sind. Diese bilden zusammen einen Slice.

Die Grundidee ist mit der des Slicing in [BERS08] eng verwandt: Im erweiterten Hyperspace-Modell ermöglicht ein gemeinsames Metamodell der verschiedenen Artefaktsprachen das Einbringen der Units in den Hyperspace. Anhand dieses Metamodells erfolgt die Zuordnung der Units zu den Belangen in den verschiedenen Dimensionen sowie die Zuordnung der Belange und ihrer Beziehungen und Abhängigkeiten zueinander.

Für den Hyperspace des jeweiligen Softwareprojektes entsteht so ein Graph, der ein gültiges Modell des gemeinsamen Metamodells darstellt. Knoten repräsentieren in diesem Modell Belange und Units, die verschiedenen Beziehungen zwischen diesen werden als Kanten unterschiedlichen Typs repräsentiert.

Das Bestimmen eines Slices kommt im Kontext des erweiterten Hyperspace-Modells der Bestimmung der transitiven Hülle eines Belangs gleich. Dabei kann zusätzlich ausgewählt werden, welche Arten von Beziehungen (dies spiegeln die verschiedenen Kantenarten wider) für die Slice-Bestimmung berücksichtigt werden - je nachdem, wie detailliert der resultierende Slice sein soll.

Eine Unterscheidung von Vorwärts- und Rückwärts-Slices ist zwar auch für das erweiterte Hyperspace-Modell theoretisch möglich, scheint jedoch nicht sinnvoll.

Schließlich lässt sich die Idee des Slicing noch erweitern: Statt einem Belang lässt sich als Slicing-Kriterium auch eine Unit wählen. Diese Unit ist direkt einem oder mehreren Belangen zugeordnet, die wiederum direkt oder transitiv mit anderen Belangen verbunden sind. Auch auf diese Weise ist Slicing möglich.

Zusammenfassend lässt sich für das erweiterte Hyperspace-Modell ein Slice anhand der obigen Erläuterungen wie folgt definieren:

> Ein Slice ist die Menge der Belange und Units, die direkt oder transitiv mit einem Slicing-Kriterium verbunden sind. Das Slicing-Kriterium kann dabei ein Belang oder eine Unit sein. Zusätzlich ist es möglich, das Slicing auf bestimmte Abhängigkeiten und Beziehungen zu reduzieren.

3 Praktische Grundlagen

Nachdem im vorangegangenen Kapitel 2 die theoretischen Grundlagen dieser Studienarbeit erläutert worden sind, folgen nun die praktischen Grundlagen. Neben der zu erweiternden bzw. zu ergänzenden Software *Enterprise Architect* von Sparx Systems werden das API *JGraLab*, das von Elmar Brauch entwickelte *Überführungstool* sowie die Graphenanfragesprache *GreQL* vorgestellt.

Die Erläuterungen beschränken sich dabei auf die für das Verständnis der verschiedenen Tools wesentlichen Informationen, welche im weiteren Verlauf dieser Ausarbeitung wo nötig ergänzt oder detaillierter erläutert werden.

3.1 Enterprise Architect

Enterprise Architect (EA) ist eine Modellierungssoftware des australischen Unternehmens *Sparx Systems*, welche die 13 Diagrammsprachen der *Unified Modeling Language 2 (UML 2)* umfasst. Im August 2000 vorgestellt, wurde das kommerzielle Programm inzwischen bis zur Version 7.5 weiterentwickelt und wird mittlerweile von mehr als 100.000 Nutzern weltweit verwendet [Spa07]. Die Arbeitsgruppe Softwaretechnik nutzt die *Corporate Edition* in Version 7.0.

Neben der Modellierung mit UML bietet EA unter anderem Funktionalitäten zum Design von Benutzerschnittstellen, dem Anforderungsmanagement sowie dem Forward- und Reverse-Code-Engineering. Desweiteren ermöglicht das Programm Individualisierungen und Erweiterungen des Funktionsumfangs: Neben der Erstellung eigener *Sichten (Views)* auf Modelle lassen sich über das *Software Developers Kit (SDK)* zusätzliche *Stereotypen* für UML-Diagramme erzeugen, die in ihrer Darstellung mit *Shape Scripts* verändert und in *UML-Profilen (UML Profiles)* gespeichert werden können. Darüber hinaus ist im SDK ein API u.a. für Java enthalten, mit dessen Hilfe sich im Enterprise Architect erzeugte Projekte modifizieren lassen und Funktionalitäten des EA angesteuert und erweitert werden können.

Die für die Umsetzung des erweiterten Hyperspace-Modells und die Bestimmung von Slices notwendigen Features werden im Folgenden vorgestellt.

3.1.1 API

Zusammen mit dem Enterprise Architect wird ein Java-API für das *Enterprise Architect Object Model* bereitgestellt. Dieses API ermöglicht über das *Automation In-*

terface unter anderem den Zugriff auf das Repository des EA[1]. Sowohl der *Enterprise Architect User Guide* [Spa08] (S. 1592ff) als auch Brauch in [Bra07] (S. 3-9) bieten einen Überblick über die Klassen des API. Analog zur letztgenannten Quelle werden hier die wichtigsten Klassen kurz vorgestellt. Abbildung 3.1 bietet einen grafischen Überblick.

Die wichtigsten Klassen des EA-API

Repository. Die Klasse `Repository` ist die Containerklasse für Modelle. Mit EA erzeugte Projekte werden in `.eap`-Dateien gespeichert. Nach dem Öffnen einer solchen Datei mit der Methode `OpenFile` hält die jeweilige Instanz der Klasse `Repository` die Referenz auf die Inhalte der Datei.

Package. Eine `Repository`-Instanz kann beliebig viele Instanzen der Klasse `Package` enthalten. Diese Pakete repräsentieren die Modelle und werden entsprechend als *Models* bezeichnet. Pakete können Unterpakete enthalten.

Element. Die Klasse `Element` ist die wichtigste Klasse zur Modellierung. Instanzen von `Element` können z.B. Klassen, Anwendungsfälle, Knoten oder Komponenten entsprechen. Dazu muss das zu `Element` gehörende Attribut `Type`, also der Typ des Elements, entsprechend gesetzt werden.

Attribute. Eine `Element`-Instanz kann eine Aggregation von Instanzen der Klasse `Attribute` enthalten. Diese entsprechen den zu einem Element gehörenden Attributen.

Method. Wie eine Aggregation von `Attribute` kann eine Instanz der Klasse `Element` auch eine Aggregation von Instanzen der Klasse `Method` enthalten, welche den zu einem Element gehörenden Methoden entsprechen.

Connector. Neben der Klasse `Element` hat die Klasse `Connector` entscheidende Bedeutung. Instanzen der Klasse `Connector` repräsentieren die verschiedenen Arten von Beziehungen zwischen UML-Elementen. Wie bei `Element` muss auch hier das Attribute `Type` gesetzt werden.

Diagram. Die bislang aufgezählten Klassen repräsentieren strukturelle Aspekte, erlauben allein aber noch nicht die Darstellung in Diagrammform. Dies wird erst möglich durch die Klasse `Diagram`. Jede Instanz dieser Klasse entspricht einem Diagramm im Enterprise Architect.

[1] Neben dem Zugriff über das Java-API sind auch Zugriffsmöglichkeiten mittels Visual Basic, Delphi und C# gegeben. Diese bieten einen größeren Funktionsumfang als das API für Java, werden hier aber nicht näher erläutert. Stattdessen sei für weitere Informationen auf den Enterprise Architect User Guide [Spa08] (S. 1586-1680) verwiesen.

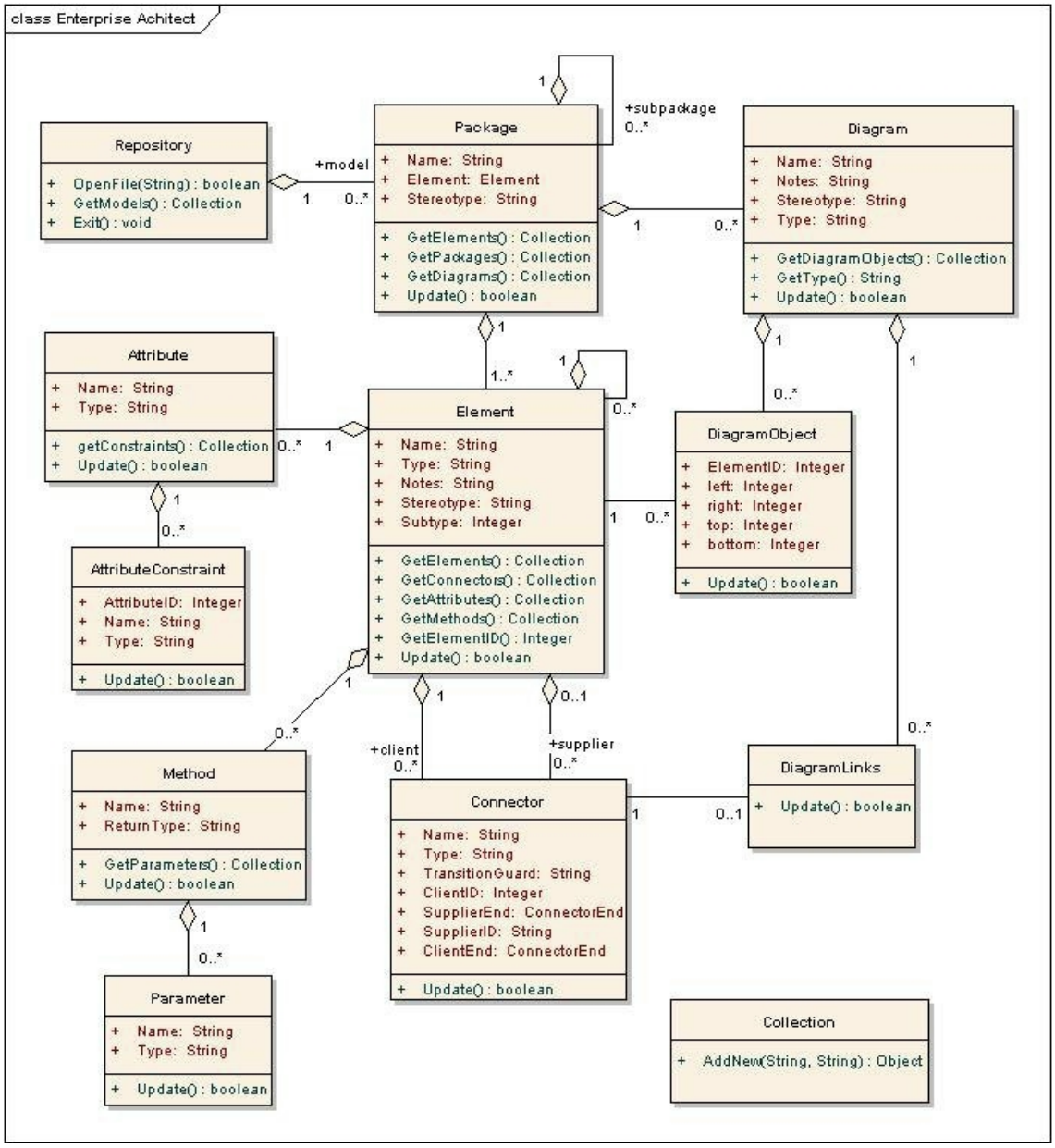

Abbildung 3.1: Übersicht über die wichtigsten API-Klassen des Enterprise Architect (Quelle: [Bra07], S. 4, Abb. 2.1)

Zu jeder Instanz der Klasse `Element` gehören eine oder mehrere Instanzen der Klasse `DiagramObject`, zu jeder Instanz der Klasse `Connector` gehört eine Instanz der Klasse `DiagramLinks`. `Diagram` hält Referenzen auf diese Klassen.

Collection. Die Klasse `Collection` dient als Klasse für Aggregationen. Soll ein Objekt eine Menge von Instanzen enthalten, wird dies mit Hilfe der Klasse `Collection` realisiert.

Vereinfachend lässt sich zusammenfassen: Die Klasse `Repository` enthält ein oder mehrere Modelle. Die Modelle enthalten ihrerseits verschiedene Elemente, die z.B. Klassen entsprechen. Elemente können Attribute und Methoden sowie Referenzen auf Beziehungen enthalten. Durch diese Kombination wird die Struktur der Modelle im Projekt wiedergegeben.

Daneben enthält jedes Modell eine Instanz von `Diagram`. Diese enthält Instanzen von `DiagramObject`, welche die Elemente darstellen, und `DiagramConnector`, welche die Konnektoren widerspiegeln. Über diese Repräsentation erfolgt die Darstellung der Modelle in Diagrammen.

3.1.2 Sichten

Enterprise Architect bietet Nutzern die Möglichkeit, eigene Sichten (Views) zu erzeugen. Dies sind quasi eigene Diagrammtypen, die andere, individualisierte Perspektiven auf ein Projekt ermöglichen.

Im Rahmen der Studienarbeit bildet eine Sicht den Rahmen für das Diagramm zum erweiterten Hyperspace-Modell.

3.1.3 Stereotypen

Eine Vielzahl von Stereotypen stehen in Enterprise Architect von vornherein zur Verfügung. Sie weisen Elementen in Modellen z.B. weitere Eigenschaften und andere Bedeutungen zu. So macht der Stereotyp `interface` aus einer Klasse ein Interface.

Stereotypen dienen im Rahmen dieser Studienarbeit zur Definition von Dimensionen, Belangen und Konnektoren im Hyperspace-Kontext.

3.1.4 Shape Scripts

Mit *Shape Scripts* lässt sich die Darstellung der Instanzen eines Elements verändern. Die in solchen Scripts festgelegten Eigenschaften werden statt der Standardformen der UML angewandt. Es gibt u.a. Befehle für Form, Farbe, Größe, Position und Beschriftung.

Die für das erweiterte Hyperspace-Modell erzeugten Elemente werden mit Hilfe von Shape Scripts in ihrer Darstellung angepasst.

3.1.5 UML-Profile

UML-Profile basieren auf Mengen von zusätzlichen Stereotypen und angehängten Werten[2], die auf Elemente angewendet werden (s. [Spa08], S. 489: *„UML Profiles [...] are based on additional stereotypes and Tagged Values that are applied to elements [...]“*). Mit solchen Profilen lassen sich zuvor erzeugte Stereotypen und Werte zur Verwendung in weiteren Projekten importieren.

Im Rahmen dieser Studienarbeit werden die für das Hyperspace-Modell erzeugten Stereotypen mitsamt der damit verknüpften Shape Scripts in einem UML-Profil gespeichert. Dieses lässt sich in andere Projekte importieren, so dass auch in diesen das erweiterte Hyperspace-Modell umgesetzt werden kann. Erzeugt man zusätzlich eine „Hyperspace-Sicht“ für ein Modell, lässt sich so ein Diagramm des erweiterten Hyperspaces erzeugen.

3.2 JGraLab

JGraLab ist die Java-Umsetzung des am Institut für Softwaretechnik der Universität Koblenz-Landau entwickelten *GraLab Graphenlabors*. Es ist ein API zur Erstellung und Manipulation von *TGraphen*. Bei TGraphen handelt es sich um typisierte, attributierte, gerichtete, angeordnete Graphen. Der Einfachheit halber wird nachfolgend häufig statt von TGraphen schlicht von Graphen gesprochen.

JGraLab baut auf *grUML (Graph UML)* [BHR+09] auf. Dies ist eine speziell auf TGraphen ausgelegte Variante der UML. Die Arbeit mit Graphen in JGraLab findet auf den Ebenen *M1* und *M2* des 4-Schichten-Modells der Object Management Group (OMG)[3] statt und besteht in der Regel aus zwei Schritten:

- Zunächst muss das *Schema* des Graphen erstellt werden. Dabei handelt es sich um das dem Graphen zugrundeliegende *Metamodell* (*M2*). Nach der Fertigstellung des Schemas erzeugt JGraLab dazu passende Java-Klassen.
- Mit den durch JGraLab erzeugten Java-Klassen wird vom Nutzer im Rahmen der *Instanziierung* ein dem Schema entsprechender Graph kreiert (*M1*).

JGraLab nutzt in erheblichem Umfang *Factory-Methoden* entsprechend [GHJV95]. Interfaces legen die Strukturen entsprechend grUML fest, Klassen enthalten die jeweils dazugehörende Implementation. Dies wird in den folgenden Erläuterungen verdeutlicht.

3.2.1 Das Schema

Vereinfachend dargestellt beschreibt das Schema eines Graphen, Instanzen welcher Knoten- und Kanten-Klassen zur Kreation eines Graphen verwendet werden dürfen. Die zugrundeliegende Struktur ist dabei die folgende (vgl. Abb. 3.2 auf S. 36):

[2] sog. *Tagged Values*, die Elementen zugeordnet werden, um zusätzliche Informationen zu liefern, die nicht in der UML vorgesehen sind

[3] Internet: `http://www.omg.org/`

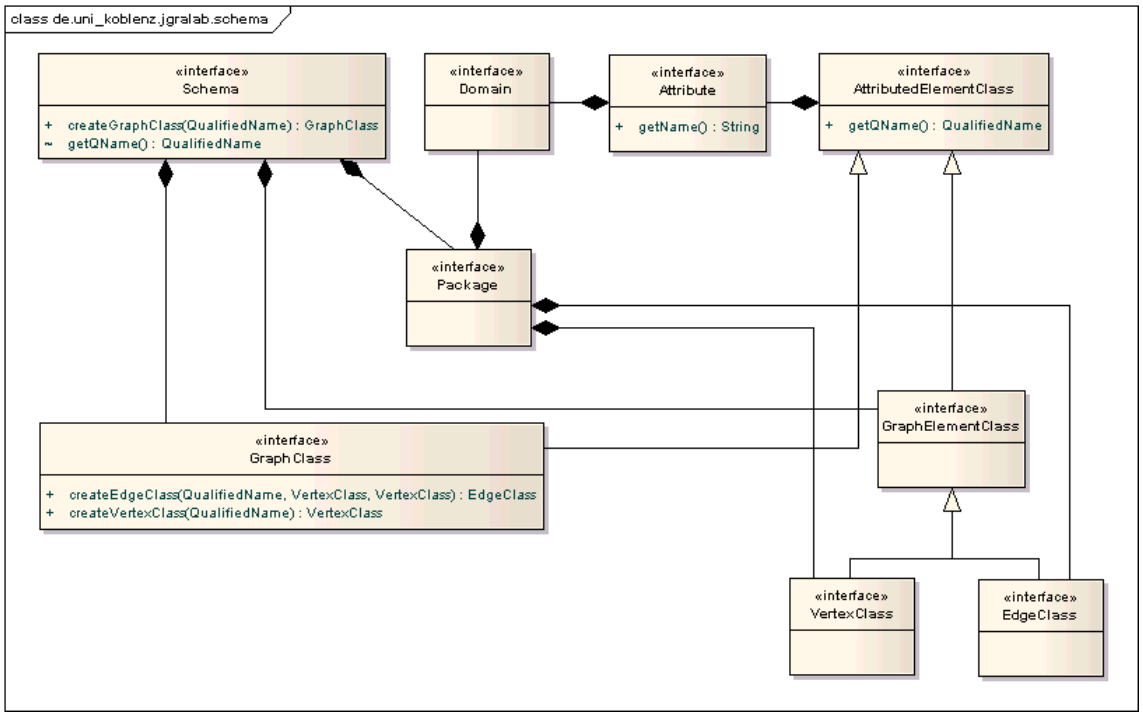

Abbildung 3.2: Übersicht über die wichtigsten Interfaces in JGraLab

Ein *Schema* definiert neben dem Namen des Schemas ein *Default-Paket* und genau eine *Graph-Klasse*. Die Graph-Klasse definiert den Typ des Graphen. Daneben werden aus der Instanz der Graph-Klasse heraus die verschiedenen *Graphelement-Klassen*, d.h. die möglichen *Knoten-* und *Kanten-Klassen* des Graphen erzeugt.

In JGraLab stehen als *Domänen* (Wertebereiche) standardmäßig verschiedene Datentypen wie z.B. *Boolean*, *Integer* und *String* zur Verfügung. Ein Paket kann zusätzlich die Struktur *zusammengesetzter Domänen*, d.h. die in *Lists*, *Records* oder *Sets* erlaubten Wertebereiche, definieren. Daneben kann ein Paket Unterpakete enthalten.

Die Erstellung eines Schemas in JGraLab stellt sich in der Regel wie folgt dar: Zunächst definiert man eine das Interface `Schema` implementierende Instanz. Aus dieser kann man nun durch Aufruf der Factory-Methode `createGraphClass()` eine Instanz erzeugen, die das Interface `GraphClass` implementiert.

Mit Hilfe der in `GraphClass` enthaltenen *create()*-Methoden werden aus der `GraphClass`-Instanz heraus die verschiedenen die Interfaces `EdgeClass` und `VertexClass` implementierenden Instanzen, d.h. die Instanzen der Kanten- und Knoten-Klassen, erzeugt.

Durch Aufruf einer *commit()*-Methode von `Schema` (z.B. `commit(String path)`) wird die Erzeugung des Schemas abgeschlossen. JGraLab generiert nun die Java-Klassen.

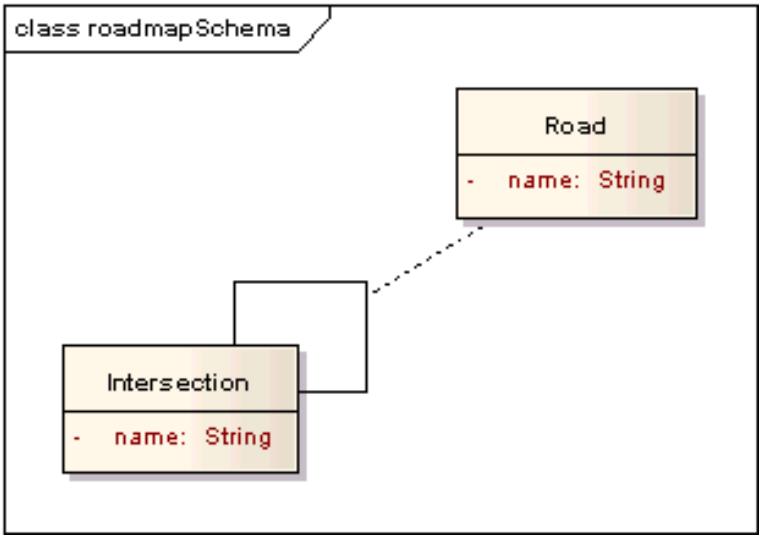

Abbildung 3.3: Klassendiagramm zum Beispielschema

Das folgende Listing zeigt ein Beispiel zur Erstellung eines Schemas für einfache Straßenkarten. Knoten repräsentieren Kreuzungen und Kanten Straßen. Abbildung 3.3 zeigt das zum Schema gehörende Klassendiagramm.

```
1  // create the Schema:
2  Schema roadmapSchema = new SchemaImpl("RoadmapSchema", "roadmap.schema"
       );
3
4  // create the GraphClass:
5  GraphClass roadmapDiagram = roadmapSchema.createGraphClass("RoadmapDiagram
       ");
6
7  // create a VertexClass for intersections:
8  VertexClass vertexIntersection = roadmapDiagram.createVertexClass("Intersection");
9  vertexIntersection.addAttribute("name", roadmapSchema.getDomain("String");
10
11 // create an EdgeClass for roads:
12 EdgeClass edgeRoad = roadmapDiagram.createEdgeClass("Road", vertexIntersection,
       vertexIntersection);
13 edgeRoad.addAttribute("name", roadmapSchema.getDomain("String");
14
15 // commit the Schema, create Java classes:
16 roadmapSchema.commit("./generated");
```

`SchemaImpl` ist eine Implementierung von `Schema`. Der Konstruktoraufruf in Zeile 2 weist dem Schema den Namen `RoadmapSchema` zu. Daneben wird festgelegt, dass das dazugehörende Paket den Namen `roadmap.schema` trägt. Dieses enthält später die erzeugten Java-Klassen.

Danach folgt die Erzeugung der Graph- sowie der Knoten-Klasse. In Zeile 12 wird die Kanten-Klasse für Straßen definiert. Der genutzte Konstruktor weist der Kanten-Klasse einen Namen sowie die Start- und Zielknoten-Klasse zu.

3.2.2 Die Instanziierung

Von den durch JGraLab erzeugten Java-Klassen sind für die Erstellung eines Graphen hauptsächlich die der Graph-Klasse sowie die der Knoten- und Kanten-Klassen von Bedeutung. Sie tragen die Namen, die im Rahmen der Schema-Erstellung den jeweiligen Konstruktoren übergeben wurden.

Um mit den Klassen einen Graphen zu kreieren, muss zunächst eine Instanz der Graph-Klasse erzeugt werden. Die Schema-Klasse enthält dazu eine *create()*-Methode, welche über ihre Instanz aufgerufen werden muss. Der Methodenaufruf erfolgt nach dem Muster `<Schema-Klasse>.instance().create<Graph-Klassenname>(<Graph-Name>, Knotenzahl, Kantenzahl)`. Dabei wird dem Graphen ein Name zugewiesen; zudem werden zum Zweck einer effizienten Speicherverwaltung eine (vorläufige) Knoten- und Kantenzahl angegeben.

Die Graph-Klasse enthält *create()*-Methoden für die verschiedenen Knoten- und Kanten-Klassen. Mit diesen parameterlosen Methoden, deren Name sich ebenfalls aus `create` und dem jeweiligen Klassennamen zusammensetzt, werden die Instanzen der Knoten- und Kanten-Klassen erzeugt. Sind im Schema Attribute für diese Klassen definiert, so finden sich in den jeweiligen Klassen entsprechende *set()*-Methoden.

Gespeichert wird der Graph mit der statischen Methode `saveGraphToFile()` bzw. `saveGraphToStream()` aus der Klasse `GraphIO`.

Das folgende Listing zeigt die Instanziierung eines Graphen zum zuvor erstellten Schema. Abbildung 3.4 stellt das Objektdiagramm des so erzeugten Graphen dar.

```
1  // create the RoadmapDiagram "Roadmap" with each 50 initial vertices and edges:
2  RoadmapDiagram roadmapDiagram = RoadmapSchema.instance().
       createRoadmapDiagram("Roadmap", 50, 50);
3
4  // create intersections and set their names to "IntersectionA", "IntersectionB", "
       IntersectionC", and "IntersectionD":
5  Intersection intersectionA = roadmapDiagram.createIntersection();
6  intersectionA.set_name("IntersectionA");
7
8  Intersection intersectionB = roadmapDiagram.createIntersection();
9  intersectionB.set_name("IntersectionB");
10
11 Intersection intersectionC = roadmapDiagram.createIntersection();
12 intersectionC.set_name("IntersectionC");
13
14 Intersection intersectionD = roadmapDiagram.createIntersection();
15 intersectionD.set_name("IntersectionD");
16
```

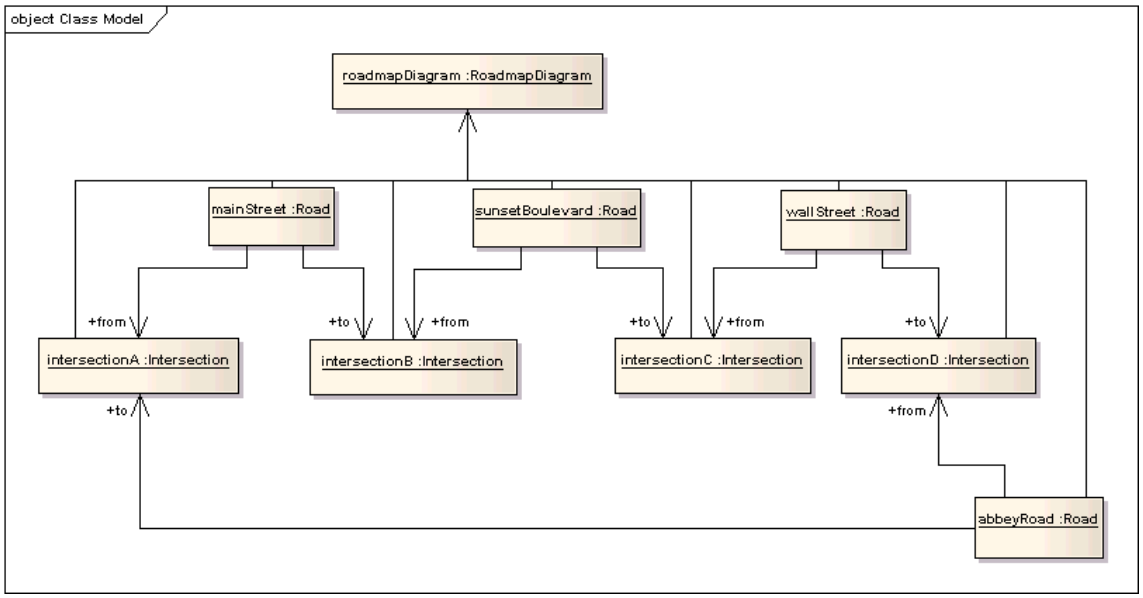

Abbildung 3.4: Objektdiagramm zum Instanziierungsbeispiel

```
17 // create a road from IntersectionA to IntersectionB and set its name to "MainStreet
       ":
18 Road mainStreet = roadmapDiagram.createRoad(intersectionA, intersectionB);
19 mainStreet.set_name("MainStreet");
20
21 // create a road from IntersectionB to IntersectionC and set its name to "
       SunsetBoulevard":
22 Road sunsetBoulevard = roadmapDiagram.createRoad(intersectionB, intersectionC);
23 sunsetBoulevard.set_name("SunsetBoulevard");
24
25 // create a road from IntersectionC to IntersectionD and set its name to "WallStreet
       ":
26 Road wallStreet = roadmapDiagram.createRoad(intersectionC, intersectionD);
27 wallStreet.set_name("WallStreet");
28
29 // create a road from IntersectionD to IntersectionA and set its name to "AbbeyRoad
       ":
30 Road abbeyRoad = roadmapDiagram.createRoad(intersectionD, intersectionA);
31 abbeyRoad.set_name("AbbeyRoad");
32
33 // save the graph:
34 GraphIO.saveGraphToFile("./roadmap.tg", roadmapDiagram, null);
```

Beispiele zur Erstellung von Schema und Graph sowie weitere Informationen zur Arbeit mit JGraLab finden sich z.B. in [BS07], [Bra07] sowie [BHR+09].

3.2.3 GReQL-Anfragen

Im Zusammenhang mit dieser Studienarbeit sind die in JGraLab zur Verfügung stehenden Fähigkeiten zum Lesen und Verarbeiten von GReQL-Anfragen von Bedeutung. Diese Funktionalitäten finden sich in den `greql2`-Paketen des JGraLab-API.

Um eine GReQL-Anfrage auf einem Graphen auszuwerten, muss zunächst eine Instanz der Klasse `GreqlEvaluator` erzeugt werden. Die Anfrage kann in Form einer externen Datei beim Konstruktoraufruf übergeben werden. Der Aufruf der Methode `startEvaluation()` stößt die Auswertung der Anfrage an. Auf das Ergebnis lässt sich mit `getEvaluationResult()` zugreifen.

Die Methode `getEvaluationResult()` liefert das Ergebnis der Anfrageauswertung gekapselt in einem `JValue`-Objekt zurück. Mit Hilfe der in `JValue` enthaltenen boolschen Methoden `isGraph()`, `isEdge()`, `isVertex()` usw. kann der Typ des Ergebnisses abgefragt werden. Es wird mit Transformationsmethoden (`toGraph()`, `toEdge()`, `toVertex()` etc.) in ein Objekt des entsprechenden Typs überführt und kann danach mit den für die jeweilige Typklasse zur Verfügung stehenden Methoden weiter ver- und bearbeitet werden.

Abschnitt 3.4 erläutert das Prinzip von GReQL-Anfragen. 4.5.1 zeigt die Nutzung der entsprechenden JGraLab-Funktionalitäten im Kontext dieser Studienarbeit.

3.3 Das Überführungstool

Um die Funktionalitäten aus JGraLab auch auf EA-Projekte anwenden zu können, muss eine Möglichkeit bestehen, die im Enterprise Architect erstellten Graphen in TGraphen zu überführen. Auf so überführte Graphen kann dann mit JGraLab zugegriffen werden. Nach der Bearbeitung von TGraphen müssen diese wieder rücküberführt werden, damit mit den veränderten Graphen in EA weitergearbeitet werden kann.

Die Studienarbeit *Überführung von UML-Modellen aus dem Enterprise Architect nach JGraLab* von Elmar Brauch [Bra07] beschäftigt sich mit dieser Problemstellung. Brauch beschreibt darin das von ihm entwickelte Tool, mit dem der Inhalt eines EA-Repositorys (vorliegend in Form einer `eap`-Datei) ausgelesen und in einen TGraphen überführt werden kann. Auch die Überführung in die umgekehrte Richtung ist möglich.

Das Tool ist in Form eines `jar`-Pakets als eigenständige Anwendung auf einer Java Virtual Machine lauffähig. Von Bedeutung ist im Hinblick auf die Entwicklung einer Funktionalität zur Bestimmung von Slices jedoch das zur Verfügung stehende API.

Das Überführungstool besteht aus vier Komponenten: dem *GUI*, dem *Controller* sowie je einem *Transformer* für die Überführung von EA nach JGraLab und umgekehrt. Im Kontext dieser Arbeit ist die GUI-Komponente nicht relevant. Die Controller-Komponente sowie die beiden Transformer-Komponenten hingegen setzen die benötigte Überführungsfunktionalität um. Abbildung 3.5 zeigt die Komponentensicht nach [Bra07].

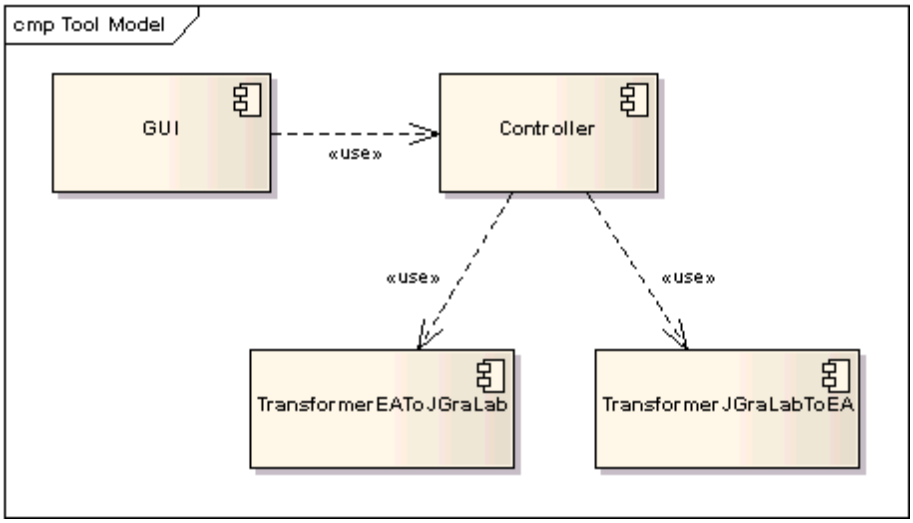

Abbildung 3.5: Komponentensicht auf das Überführungstool nach [Bra07], S. 31, Abbildung 4.2

Während die Controller-Komponente einzig durch die Klasse `Controller` realisiert ist, setzen sich die Transformer aus mehreren Klassen zusammen: Die jeweilige `Transformer`-Klasse stellt die benötigte Methode `transform()` bereit und legt fest, wie die Umwandlung von einem in den jeweils anderen Repository-Inhalt erfolgt. Sie hat Zugriff auf die entsprechende `Writer`-Klasse, welche die Funktionalität zum Schreiben des jeweiligen Repository-Inhalts bereitstellt. Abbildung 3.6 auf Seite 42 zeigt das zum Überführungstool gehörende Klassendiagramm.

Die Nutzung der API-Klassen erfolgt in zwei Schritten: Man erzeugt eine Instanz der `Controller`-Klasse und ruft die darin enthaltene Methode `transform()` auf. Dabei übergibt man den Pfad der originalen Repository-Datei und den der Zieldatei. `transform()` führt dann, je nach Quell- und Zielpfad, die Umwandlung von einer in die andere Repository-Form durch.

Zur Nutzung der Überführungsfunktionalität in eigenen Anwendungen müssen die Pakete `getopt.jar` und `jgralab.jar` installiert sein. Daneben muss auch auf `SSJavaCOM.dll` und `eaapi.jar` zugegriffen werden.

3.4 GReQL - Die Graph Repository Query Language

Die *Graph Repository Query Language (GReQL)*, hier verwendet in der Version *GReQL 2*, ist eine Graph-Anfragesprache. Sie dient der Extraktion von inhaltlichen, strukturellen oder aggregierten Informationen aus TGraphen. Da GReQL bereits in verschiedensten Publikationen der Arbeitsgruppe Softwaretechnik der Universität Koblenz-Landau ausführlich erläutert worden ist, werden hier bewusst nur die

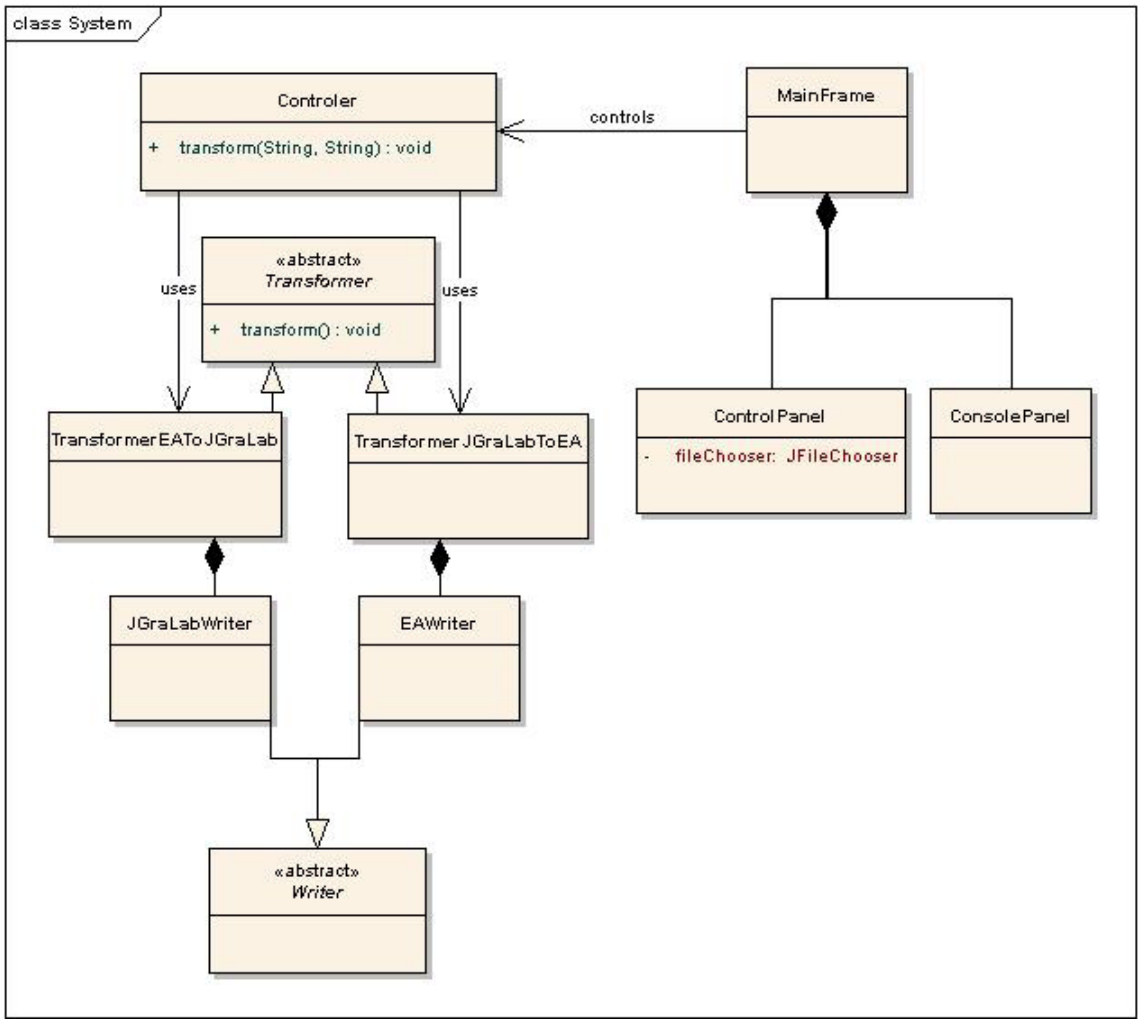

Abbildung 3.6: Klassendiagramm des Überführungstools (Quelle: [Bra07], S. 33, Abbildung 4.3)

elementaren Bestandteile und Funktionen, basierend auf den Diplomarbeiten von Katrin Marchewka [Mar06] und Daniel Bildhauer [Bil06], vorgestellt.

3.4.1 FWR-Ausdrücke

Das wichtigste Sprachelement der GReQL sind sogenannte *FWR-Ausdrücke*. Die Kombination *FWR* spiegelt dabei den üblichen Aufbau einer GReQL-Anfrage wider: Diese setzt sich zusammen aus einer `from`-, einer `with`- und einer `report`-Klausel, welche anhand des folgenden Beispiels beschrieben werden.

```
from road:E{Road}
with road.length > 500 and road.length < 1000
report road.name
end
```

Die `from`-Klausel deklariert die im weiteren Verlauf der Anfrage verwendeten Variablen. Sie legt den bzw. die Namen der Variablen fest. Dabei wird jeweils bestimmt, ob es sich um einen Knoten (`V`) oder eine Kante (`E`) handelt. Schließlich wird die jeweilige Art des Knotens bzw. der Kante festgelegt. Der Aufbau einer from-Klausel stellt sich wie folgt dar: `from <Variablenname>: <V/E>{<Knoten-/Kantenart>}`. Mehrere Variablendeklarationen werden durch Kommata getrennt.

Auf die `from`- folgt optional die `with`-Klausel. Sie stellt über eine Formel der Prädikatenlogik Bedingungen an die Variablen aus der `from`-Klausel. Neben den Operatoren `and`, `or` und `not` stehen der All- (`forall`) sowie der Existenzquantor (`exists`) zur Verfügung. Auch Pfadausdrücke sind zur Formulierung einer Bedingung möglich.

Die `report`-Klausel schließlich beschreibt, wie das Ergebnis der Anfrage aufgebaut ist.

Im Beispiel werden für die Variable `road`, welche Kanten (`E`) vom Typ `Road` repräsentiert, all die Instanzen ausgewählt, deren Attribut `length` zwischen `500` und `1000` liegt. Für diese wird jeweils der Wert des Attributs `name` gespeichert. Es werden also die Namen der Straßen gespeichert, die zwischen 500 und 1000 (z.B. Meter) lang sind.

3.4.2 `store as` und `using`

Statt der `report`- kann eine `store as`-Klausel verwendet werden. Das Ergebnis der Anfrage wird dann einer Variable zugewiesen und so für folgende Anfragen persistent gemacht.

Um ein so zwischengespeichertes Ergebnis in einer folgenden Anfrage zu nutzen, wird die `using`-Klausel benötigt. Sie wird der Anfrage vorangestellt und besteht aus dem Schlüsselwort `using` sowie dem Namen der zu verwendenden Variable.

GReQL-Anfragen werden im Kontext dieser Studienarbeit bei der Bestimmung von Slices zum Einsatz kommen. Für Anwendungsbeispiele und nähere Informationen zu GReQL, auch und insbesondere im Hinblick auf Pfadausdrücke, sei hier auf die beiden im einleitenden Absatz dieses Abschnitts genannten Diplomarbeiten verwiesen.

4 Anforderungen und Entwurf

Dieses Kapitel behandelt den Entwicklungsprozess der zu entwickelnden Software. Zunächst werden mögliche Anwendungsszenarien und -fälle betrachtet. Darauf folgen die Anforderungsdefinition sowie die Erläuterungen zur Umsetzung des Hyperspace-Modells im Enterprise Architect. Abgeschlossen wird das Kapitel durch die Erläuterungen zum Entwurf und der Architektur der Software, die von nun an den Namen *Hynalysis* (zusammengesetzt aus *Hy*perspace und a*nalysis*) tragen soll.

4.1 Einsatzszenarien und Anwendungsfälle

Bevor in Abschnitt 4.2 die Anforderungen an Hynalysis definiert werden, werden hier zunächst die verschiedenen denkbaren Einsatzszenarien für die Software betrachtet. Diesen folgen Erläuterungen zu einem möglichen Programmdurchlauf. Abgeschlossen wird dieser Abschnitt durch die Vorstellung möglicher Anwendungsfälle.

4.1.1 Szenarien

In allen hier zu betrachtenden Szenarien kommt Enterprise Architect mit zusätzlichen zu erstellenden Funktionalitäten zum Einsatz. Daneben wird der Nutzer für einen Teil der Szenarien Hynalysis verwenden.

Erstellung des Hyperspace-Diagramms

Die Erstellung eines Hyperspace-Diagramms erfolgt ausschließlich innerhalb des Enterprise Architect. Hynalysis kommt hier noch nicht zum Einsatz. Zwei mögliche Ausgangssituationen sind denkbar:

1. Der Nutzer entwirft sein Softwaresystem von Beginn an unter Einbeziehung des Hyperspace-Diagramms.
2. Der Nutzer hat bereits mit dem Entwurf des Softwaresystems begonnen. Er hat ein oder mehrere UML-Diagramme erstellt und bindet nun zusätzlich ein Hyperspace-Diagramm in das Projekt ein.

Zur Erstellung des Hyperspace-Diagramms werden verschiedene Elemente sowie eine zusätzliche Sicht benötigt. Diese können auf zwei Wegen zugänglich gemacht werden: Der Nutzer kann in ein EA-Projekt ohne Hyperspace-Diagramm ein UML-Profil mit für die Erstellung eines Hyperspace-Diagramms nötigen Elementen importieren und eine zusätzliche Sicht für das Hyperspace-Diagramm erzeugen oder aber ein EA-

Package verwenden, welches bereits die benötigte Sicht sowie die nötigen Elemente enthält.

Während der Nutzer in der erstgenannten Ausgangssituation beide Varianten nutzen kann, um in EA die Software von Beginn an unter Einbeziehung des Hyperspace-Diagramms zu entwerfen, besteht in der zweiten vorgestellten Situation bereits ein EA-Projekt, in dem der Nutzer verschiedene UML-Diagramme erstellt hat. Somit bleibt nur die Option des Imports des UML-Profils: Der Nutzer muss in EA eine zusätzliche Sicht für das Hyperspace-Diagramm erzeugen und die für das Hyperspace-Diagramm benötigten Elemente durch den Import des entsprechenden UML-Profils verfügbar machen.

Die technische Komponente der Umsetzung dieses Szenarios wird in 4.3 erläutert.

Untersuchung des Hyperspaces

Zur Untersuchung des Hyperspaces muss der Nutzer bereits ein Hyperspace-Diagramm im Enterprise Architect erstellt haben. Dieses Szenario baut somit auf dem zuvor beschriebenen auf.

Wesentlicher Bestandteil ist das Stellen und die Auswertung von GReQL-Anfragen an den Hyperspace. Diese können im Rahmen der Nutzung des Enterprise Architect mit Hynalysis verschiedene Zwecke erfüllen:

- Eine GReQL-Anfrage kann zur Überprüfung der Existenz von Diagrammelementen dienen. Im Kontext dieses Projekts lassen sich so Fragen wie *„Existiert Anwendungsfall A?“* beantworten.
- Sie kann zur Ermittlung der Erreichbarkeit eingesetzt werden. Damit lassen sich Fragen wie *„Steht Klasse X in Verbindung zu Anwendungsfall Y?“*, aber auch Fragen der Art *„Welche Klassen und Anwendungsfälle stehen mit Klasse Z in Verbindung?“* beantworten.
- Schließlich können im Zusammenhang mit der Erreichbarkeit auch die dazugehörenden vollständigen Pfade ermittelt werden. Die Menge all dieser Pfade repräsentiert den *Slice* des Elements.

Für die Bestimmung eines Slices kann man wiederum drei Fälle unterscheiden:

- Der Nutzer kann Elemente eines Diagramms auswählen. Darauf basierend wird zur Bestimmung des dazugehörenden Slices eine in Hynalysis enthaltene GReQL-Anfrage gestellt.
- Der Nutzer kann Elemente eines Diagramms auswählen und zusätzlich durch einen Pfadausdruck die für die Slice-Bestimmung zu berücksichtigenden Knoten- und Kantenarten einschränken. Zur Bestimmung des Slices wird dann eine auf dem Pfadausdruck und den ausgewählten Elementen basierende GReQL-Anfrage gestellt.
- Der Nutzer erstellt eine zur Slice-Bestimmung passende GReQL-Anfrage in einer `greql`-Datei, ohne zuvor Diagrammelemente auszuwählen.

Für den erstgenannten Fall gestaltet sich der Ablauf wie folgt: Der Nutzer wählt zunächst im Enterprise Architect ein oder mehrere Elemente eines Diagramms aus.

Dann startet er Hynalysis aus dem EA und in Hynalysis die Slice-Bestimmung für die ausgewählten Elemente. Hynalysis greift daraufhin zunächst auf das Überführungstool zu, welches einen dem EA-Repository entsprechenden TGraphen generiert. Anschließend wird in Hynalysis eine GReQL-Anfrage zur Slice-Bestimmung erzeugt, die auf den vom Nutzer ausgewählten Diagrammelementen basiert. Diese Anfrage wird dann mit JGraLab ausgewertet.

Im zweiten Fall ist der Ablauf ähnlich: Der Nutzer wählt zunächst im Enterprise Architect ein oder mehrere Elemente des Diagramms aus. Dann startet er Hynalysis aus dem EA und gibt darin den gewünschten Pfadausdruck an. Im Anschluss startet er die Slice-Bestimmung für den gegebenene Pfadausdruck und die ausgewählten Elemente. Hynalysis überführt daraufhin mit Hilfe des Überführungstools das EA-Repository in einen entsprechenden TGraphen und generiert für die ausgewählten Diagrammelemente und den gegebenen Pfadausdruck eine GReQL-Anfrage zur Slice-Bestimmung, die dann mit JGraLab ausgewertet wird.

Der Ablauf des letztgenannten Falls entspricht dem für die GReQL-Anfragen zu Existenz und Erreichbarkeit: Der Nutzer erzeugt in einem beliebigen Editor eine `greql`-Datei mit einer Anfrage. Aus dem Enterprise Architect öffnet er Hynalysis und übergibt dem Tool den Pfad der Datei. Die Anfrageauswertung wird dann aus Hynalysis gestartet. Dazu wird mit Hilfe des Überführungstools der Inhalt des EA-Repositorys in einen TGraphen überführt, auf dem dann in JGraLab die GReQL-Anfrage ausgewertet wird.

Da JGraLab die Auswertungsergebnisse als Instanzen von JValue liefert, müssen diese sowohl im Falle der auf der Elementauswahl basierenden Slice-Bestimmung als auch im Falle der „gewöhnlichen“ Auswertung von GReQL-Anfragen im Anschluss in Hynalysis unter Nutzung von JGraLab für die Ergebnisausgabe angepasst werden. Um einen Export der Ergebisse als `tg`- oder `eap`-Datei zu ermöglichen, müssen diese so in einen TGraphen überführt werden, dass dieser mit Hilfe des Überführungstools wieder in ein EA-Repository umgewandelt werden kann.

Export der Untersuchungsergebnisse

Ist die Untersuchung abgeschlossen und sind die Ergebnisse für die Ausgabe vorbereitet, so kann der Export der Ergebnisse erfolgen. Zwei Varianten sind hier vorstellbar: der Export im TGraph-Dateiformat (`*.tg`) sowie der Export als EA-Projekt (`*.eap`).

Vor dem Start der Auswertung muss der Nutzer in Hynalysis die gewünschten Exportvarianten auswählen und jeweils den Pfad der Exportdatei angeben. Hynalysis kann dann nach der Anpassung der Untersuchungsergebnisse den erzeugten TGraphen speichern oder diesen mit dem Überführungstool in ein EA-Repository umwandeln. Dabei wird dem Überführungstool gleich der Pfad der Exportdatei als Zieldatei übergeben, so dass für diese Exportart kein zusätzlicher Arbeitsschritt nötig ist.

Darstellung der Untersuchungsergebnisse

Neben dem Export von Untersuchungsergebnissen kann auch eine Darstellung derselben im Enterprise Architect erfolgen. Für diese Darstellung kommen sowohl die Einfärbung der Ergebniselemente im Originalprojekt als auch die Darstellung der Ergebniselemente in einem separatem Diagramm oder aber in Form eines eigenen Projekts infrage.

Das Einfärben der Ergebniselemente im Originalprojekt erfordert einen Vergleich der Ergebnismenge der Auswertung mit dem Originalprojekt in Hynalysis. Für die Darstellung der Ergebniselemente in einem separatem Diagramm ist die Erzeugung eines solchen und das Einbinden desselben in das Originalprojekt in Hynalysis notwendig. Die Darstellung in Form eines eigenen EA-Projekts lässt sich unmittelbar mit dem Export der Untersuchungsergebnisse als EA-Projekt verknüpfen, unterscheidet sie sich von diesem doch lediglich um das zusätzliche Öffnen der exportierten `eap`-Datei mit Enterprise Architect aus Hynalysis.

Export des EA-Projekts als TGraph

Ein weiteres vorstellbares Szenario ist der Export des Originalprojekts als TGraph. Diese Funktionalität wird zwar bereits von Elmar Brauchs Überführungstool geboten. Da eine Transformation von EA nach JGraLab aber ohnehin zur Untersuchung des Hyperspaces notwendig ist, bietet sich die Aufnahme einer entsprechenden Exportmöglichkeit in Hynalysis an.

Der Nutzer muss in diesem Fall in Hynalysis den Pfad der Zieldatei angeben. Dieser wird dem Überführungstool übergeben, wenn dieses aus Hynalysis aufgerufen wird.

4.1.2 Beispieldurchlauf von Hynalysis

Nach Betrachtung der verschiedenen Szenarien wird nun ein möglicher Programmdurchlauf von Hynalysis betrachtet. Ausgehend davon, dass der Nutzer in Enterprise Architect neben anderen Diagrammen bereits ein Diagramm des Hyperspace erstellt hat, ergibt sich folgender Ablauf:

Der Nutzer wählt in EA durch Anklicken ein oder mehrere Diagrammelemente aus und öffnet dann Hynalysis. In Hynalysis gibt er einen Pfadausdruck an, mit dem die für die Slice-Bestimmung zu berücksichtigenden Knoten- und Kantenarten festgelegt werden. Alternativ zu Elementauswahl und Pfadangabe erstellt er eine GReQL-Datei mit einer Anfrage, öffnet Hynalysis und gibt in diesem den Pfad der Datei an.

Im Anschluss wählt der Nutzer in Hynalysis aus, welche Zwischen- und Endergebnisse er exportieren möchte. Möglich sind neben der TGraph-Version des originalen EA-Repositorys der Export des TGraphen oder des EA-Repositorys der Untersuchungsergebnisse. Für alle gewünschten Exportvarianten muss der Nutzer eine Zieldatei angeben.

Schließlich kann der Nutzer entscheiden, ob die Untersuchungsergebnisse durch Einfärben der betreffenden Elemente im Original-Repository, in einem zusätzlichen Diagramm im Original-Repository oder in einem gesonderten Repository im Enterprise Architect dargestellt werden.

Hat der Nutzer seine Auswahl z.B. durch einen Klick auf einen *[OK]*-Button abgeschlossen, startet der interne Programmablauf: Der Prüfung der Validität der vom Nutzer gemachten Pfadangaben in Hynalysis folgt die Umwandlung der Inhalte des EA-Repositorys in einen TGraphen mit Hilfe des Überführungstools. Dieser wird - gegebenenfalls an der vom Nutzer spezifizierten Stelle - in einer `tg`-Datei gespeichert.

An die Umwandlung schließt sich zunächst die Erzeugung einer GReQL-Anfrage zur Slice-Bestimmung in Hynalysis an, falls der Nutzer im Enterprise-Architect Diagrammelemente ausgewählt und in Hynalysis keine `greql`-Datei angegeben hat. Danach folgt die Auswertung der GReQL-Anfrage mit JGraLab. Die Auswertungsergebnisse müssen dann in Hynalysis für die Ausgabe vorbereitet werden. Hat der Nutzer den Export der Untersuchungsergebnisse als `tg`-Datei gewählt, wird der so erzeugte TGraph in der vorgegebenen Zieldatei gespeichert.

Abschließend erfolgt mit Hilfe des Überführungstools die Transformation des TGraphen in ein EA-Repository. Dieses wird als `eap`-Datei gespeichert, wobei dem Überführungstool beim Aufruf der Pfad der `eap`-Datei als Zielpfad übergeben wird. Je nach gewählter Ausgabemethode wird das so erzeugte separate Ergebnis-Repository mit EA geöffnet, ein zusätzliches Diagramm im Original-Repository eingefügt oder die Elemente des Ergebnis-Repositorys werden im Original-Repository eingefärbt. In den beiden letzteren Fällen wird im Anschluss das Original-Repository im Enterprise Architect geöffnet. Gegebenenfalls werden in einem letzten Schritt nicht mehr benötigte Dateien mit Zwischenergebnissen gelöscht.

Mit Hilfe der beiden Abbildungen 4.1 und 4.2 soll der Programmdurchlauf noch einmal visualisiert werden. Zugleich verdeutlichen diese Abbildungen auch, welche Software für welche Aktivitäten und Szenarien zum Einsatz kommt. Abbildung 4.1 auf Seite 50 zeigt einen Beispieldurchlauf mit Selektion von Diagrammelementen, Abbildung 4.2 auf Seite 51 den Durchlauf bei Auswertung einer GReQL-Anfrage. In beiden Beispielen werden alle Exportmöglichkeiten in Anspruch genommen.

4.1.3 Anwendungsfälle

Aus den vorangegangenen Überlegungen lassen sich sechs Anwendungsfälle herleiten, welche auch in Abbildung 4.3 auf Seite 52 dargestellt sind:

1. *Erstellen eines Hyperspace-Diagramms (Create Hyperspace Diagram)*: Der Nutzer erstellt ein Hyperspace-Diagramm im Enterprise Architect.
2. *Export eines EA-Repositorys als TGraph (Export Repository as TGraph)*: Ein Repository wird nach der Überführung von EA nach JGraLab gespeichert.
3. *Untersuchung des Hyperspaces (Analyze Hyperspace)*: Der Nutzer stellt eine GReQL-Anfrage an den Hyperspace oder wählt Diagrammelemente aus, auf denen basierend eine GReQL-Anfrage ausgewertet wird.

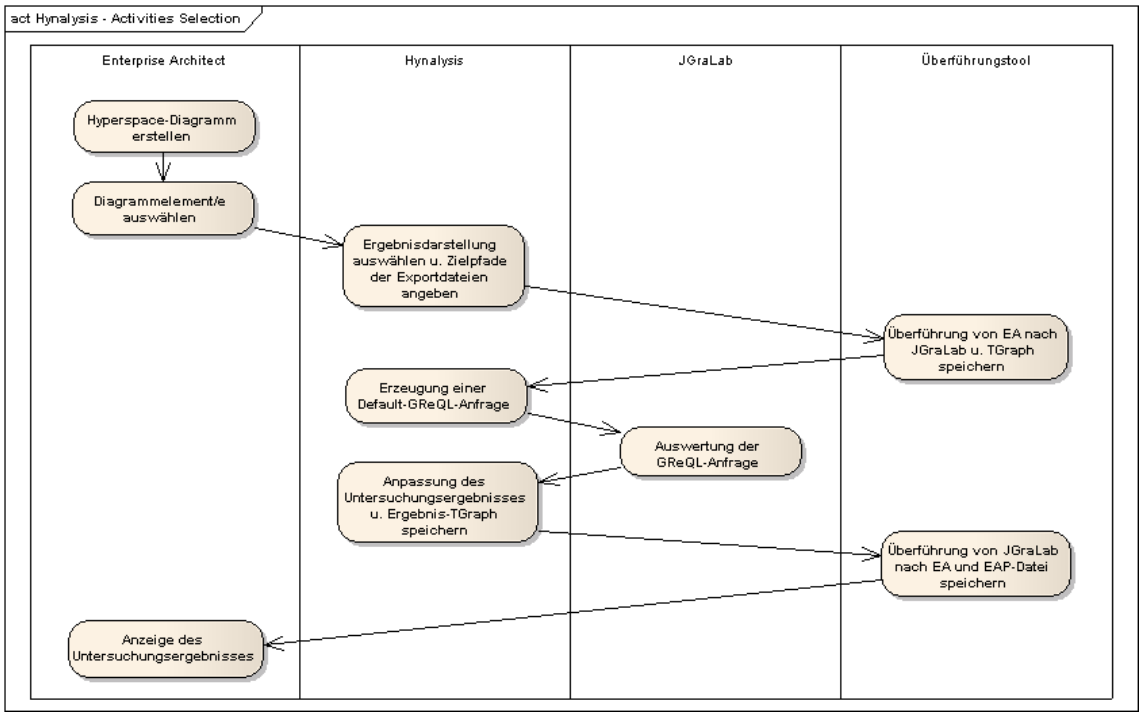

Abbildung 4.1: Hynalysis - Möglicher Programmdurchlauf mit Auswahl von Diagrammelementen

4. *Export des Untersuchungsergebnisses als TGraph (Export Analysis Result as TGraph)*: Das Untersuchungsergebnis wird in einer `tg`-Datei gespeichert.
5. *Export des Untersuchungsergebnisses als EA-Repository (Export Analysis Result as eap File)*: Das Untersuchungsergebnis wird in einer `eap`-Datei gespeichert.
6. *Anzeigen des Untersuchungsergebnisses (Display Analysis Result)*: Das Untersuchungsergebnis wird als separates Repository, als zusätzliches Diagramm im Original-Repository oder durch Einfärben der Ergebniselemente im Original-Repository im Enterprise Architect dargestellt.

Die tatsächlich umzusetzenden Use Cases werden in der Anforderungsdefinition in Abschnitt 4.2 festgelegt. Abhängig davon können sich Änderungen am vorgestellten möglichen Ablauf der Toolanwendung ergeben.

4.2 Anforderungen

An die zu entwickelnde Software werden verschiedene Anforderungen gestellt, die sich in vier Gruppen gliedern lassen:

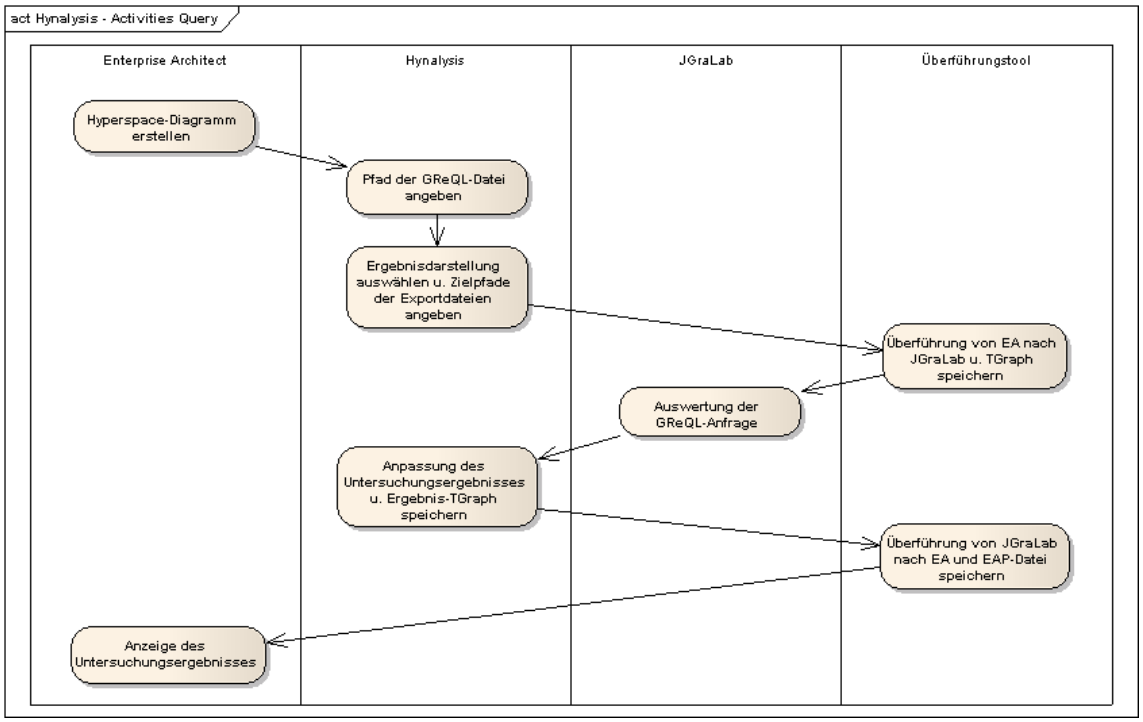

Abbildung 4.2: Hynalysis - Möglicher Programmdurchlauf bei einer GReQL-Anfrage

4.2.1 Funktionale Anforderungen

1. Die Software muss das erweiterte Hyperspace-Modell von Lohmann anwendbar für den Enterprise Architect machen.
2. Der Anwender muss Belangdimensionen und Belange erstellen können.
3. Die Units der verschiedenen im Enterprise Architect erstellten Artefakte müssen den Belangen zugeordnet werden können.
4. Die Software muss die Bestimmung von Slices basierend auf im Enterprise Architect ausgewählten Diagrammelementen ermöglichen.
5. Die Software muss die Auswertung von GReQL-Anfragen ermöglichen.
6. Auszuwertende GReQL-Anfragen sollen aus externen `.greql`-Dateien importiert werden können.
7. Für die Bestimmung von Slices sollen zu berücksichtigende Verbindungs- oder Knotentypen ausgewählt werden können.
8. Das Ergebnis der Auswertung der GReQL-Anfrage muss durch Einfärben der Ergebniselemente im Original-Repository im Enterprise Architect angezeigt werden.
9. Das Ergebnis der Auswertung der GReQL-Anfrage muss in Form einer `tg`-Datei exportiert werden können.
10. Das Ergebnis der Auswertung der GReQL-Anfrage soll als zusätzliches Diagramm im Original-Repository im Enterprise Architect angezeigt werden.

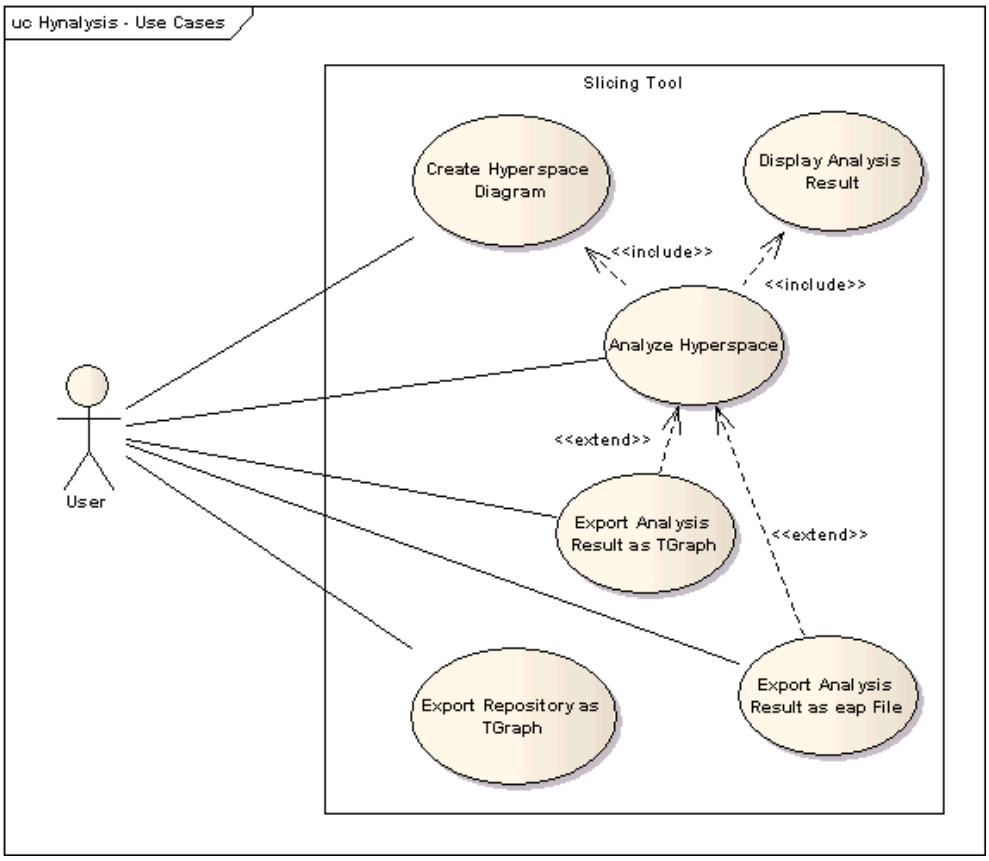

Abbildung 4.3: Hynalysis - Mögliche Anwendungsfälle

11. Das Ergebnis der Auswertung der GReQL-Anfrage kann in Form eines zusätzlichen EA-Repositorys exportiert und im Enterprise Architect angezeigt werden.
12. Die Funktionalität zur Bestimmung von Slices und zur Auswertung von GReQL-Anfragen soll aus dem Enterprise Architect heraus angestoßen werden können.

4.2.2 Technische Anforderungen

1. Die Software muss kompatibel zu *Enterprise Architect, Version 7.0* sein.
2. Die Software muss auf *Java 6* basieren.
3. Die Software muss das von der Arbeitsgruppe Softwaretechnik entwickelte API *JGraLab* und die Graphenanfragesprache *GReQL* für die Slice-Bestimmung verwenden.
4. Die Software soll das von Elmar Brauch entwickelte Tool zur Überführung von Graphen zwischen JGraLab und dem Enterprise Architect verwenden.
5. Die Kommentare müssen in *Javadoc* abgefasst sein.
6. Die Software soll mit der Entwicklungsumgebung *Eclipse Ganymede* entwickelt werden.

4.2.3 Qualitätsanforderungen

1. Die zu entwickelnde Software soll stabil laufen.
2. Der erstellte Quellcode muss hinreichend kommentiert sein.
3. Eine Dokumentation der wesentlichen Bestandteile der zu entwickelnden Software muss in die Studienarbeit eingebunden werden.
4. Die zu erstellende Software soll über eine Fehlerbehandlung verfügen.

4.2.4 Sonstige Anforderungen

1. Die zu verwendende Sprache für Bezeichner, Kommentare, Fehlermeldungen und das Nutzer-Interface ist englisch.

4.3 Umsetzung des erweiterten Hyperspace-Modells in Enterprise Architect

Die Umsetzung des erweiterten Hyperspace-Modells im Enterprise Architect erfolgt mittels UML-Profilen, Stereotypen, Shape Scripts und Views. Dieser Abschnitt beschreibt zunächst, welche Bestandteile im einzelnen für die Umsetzung des Modells benötigt werden. Danach folgt die Beschreibung der Erstellung dieser Bestandteile sowie ihrer Nutzung in EA. Die Einsatzweise wird im folgenden Kapitel anhand eines Beispiels erläutert.

4.3.1 Das Hyperspace-Profil

Die zur Umsetzung des erweiterten Hyperspace-Modells benötigten Elemente werden in Form eines UML-Profils bereitgestellt. Zur Erstellung dieser Elemente werden Stereotypen definiert, deren Darstellung mit Hilfe von Shape Scripts angepasst wird.

Stereotypen

Zur Identifikation der benötigten Stereotypen werden die im erweiterten Hyperspace-Modell Verwendung findenden Elemente betrachtet. Die UML sieht standardmäßig lediglich Units, nicht aber Dimensionen und Belange vor. Ebenso sind die zugehörigen Konnektoren zwischen Dimensionen und Belangen sowie zwischen Belangen und Units nicht berücksichtigt.

Entsprechend sind die zu definierenden Stereotypen die folgenden:

- *Dimension* zur Beschreibung und Darstellung von Dimensionen,
- *Concern* zur Beschreibung und Darstellung von Belangen,
- *PrimaryConcernUnitConnector* zur Beschreibung und Darstellung von Verbindungen zwischen Belangen einer *artefaktbasierenden* Belangdimension und Units sowie

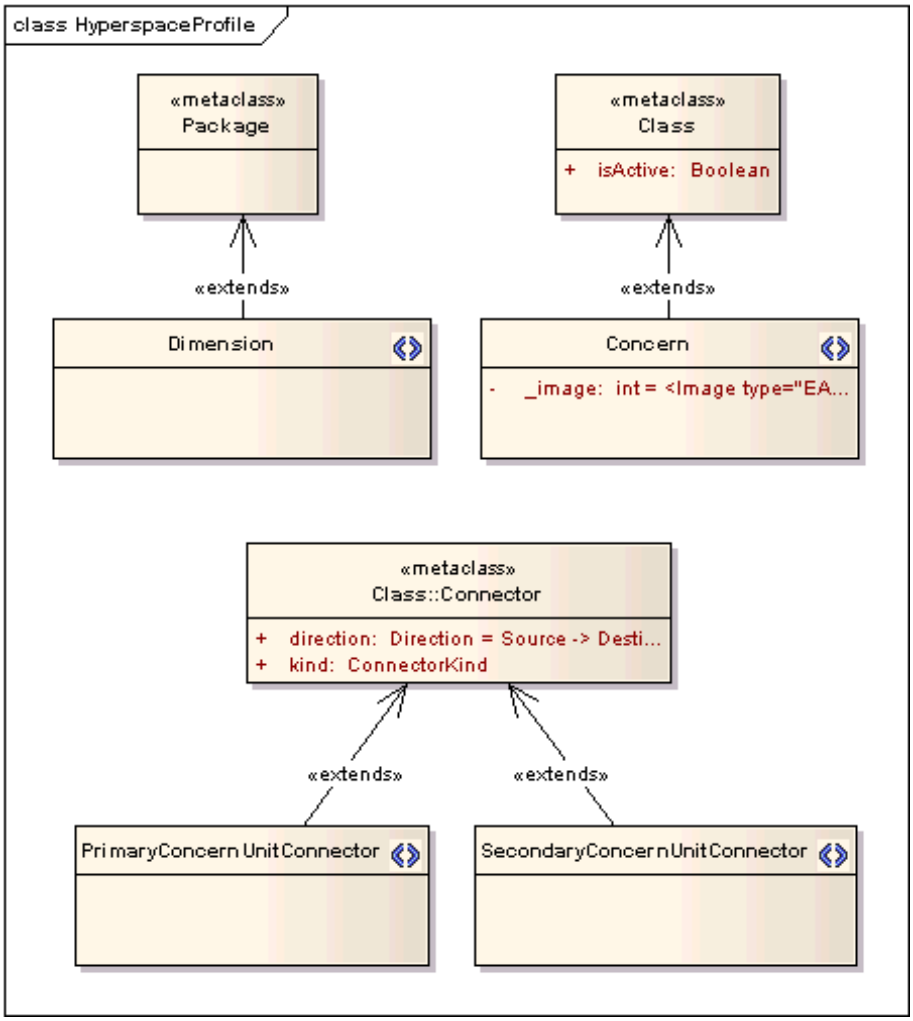

Abbildung 4.4: Übersicht über die Stereotypen des erweiterten Hyperspace-Modells

- *SecondaryConcernUnitConnector* zur Beschreibung und Darstellung von Verbindungen zwischen Belangen einer *selektierenden* Belangdimension und Units.

Die Stereotypen werden von verschiedenen Metaklassen abgeleitet: Den Stereotypen `PrimaryConcernUnitConnector` und `SecondaryConcernUnitConnector` liegt die Metaklasse `Connector` zugrunde. `Dimension` ist eine Spezialisierung der Metaklasse `Package`, `Concern` eine Spezialisierung von `Class`.

Für die Konnektoren zwischen Dimensionen und Belangen muss kein eigener Stereotyp erzeugt werden: Ebenso wie Klassen einem Paket zugeordnet werden, werden auch Belange einer Dimension zugeordnet. Abbildung 4.4 zeigt eine Übersicht der Vererbungshierarchie für die Stereotypen, wie sie sich während der Erzeugung in EA darstellt.

Abbildung 4.5: Dialog *New Package*

Shape Scripts

Die Realisierung des erweiterten Hyperspace-Modells erfolgt mit zwei *Diagrammtypen*: Die Dimensionen werden als Pakete in einem *Paketdiagramm* dargestellt. Dabei wird ihr Erscheinungsbild gegenüber dem eines „herkömmlichen“ Pakets nicht geändert. So zeigt das Paketdiagramm die Dimensionen mitsamt der zugehörigen Belange.

Wie bei allen Paketen gehört zu jeder erzeugten Dimension ein *Klassendiagramm*. Diese Diagramme enthalten die zu der jeweiligen Dimension gehörenden Belange sowie die diesen Belangen zugeordneten Artefakte.

Im Gegensatz zur Darstellung der Dimension wird die des Stereotyps `Concern` mit einem Shape Script angepasst. Belange werden im Klassendiagramm einer Dimension als grüne Ellipsen dargestellt. Die Darstellung der Konnektoren zwischen Belangen und Artefakten entspricht der Standarddarstellung von `Connector`-Elementen. Auch die Darstellung der Artefakte wird nicht verändert.

UML-Profile

Zur Umsetzung des erweiterten Hyperspace-Modells im Enterprise Architect wird ein UML-Profil benötigt. Dieses dient als Container für die zuvor erläuterten Stereotypen und die dazugehörenden Shape Scripts. Durch Einbinden dieses Profils, des sog. *HyperspaceProfile*, in ein EA-Projekt werden die zur Erstellung eines Hyperspace-Diagramms benötigten Elemente für dieses Projekt verfügbar gemacht.

Erstellung des UML-Profils

Zur Erstellung des Hyperspace-Profils sind mehrere Schritte nötig: Zunächst werden ein Paketdiagramm und die Werkzeugleiste *Profile* geöffnet. Aus der Werkzeugleiste wird der Menüpunkt *Profile* in das Diagramm gezogen. Es öffnet sich ein Dialogfenster (*New Package*), in dem der Name des zu erzeugenden Profils (hier: `HyperspaceProfile`) eingegeben wird (Abbildung 4.5). Nach einem Klick auf *[OK]* wird das definierte Paket erzeugt. Ein Doppelklick auf das neu erzeugte Paket öffnet das zu diesem Paket gehörende Klassendiagramm. Im Klassendiagramm werden die Stereotypen definiert. `Dimension` wird von der Metaklasse `Package`, `Concern` von der Metaklasse `Class` und `PrimaryConcernUnitConnector`

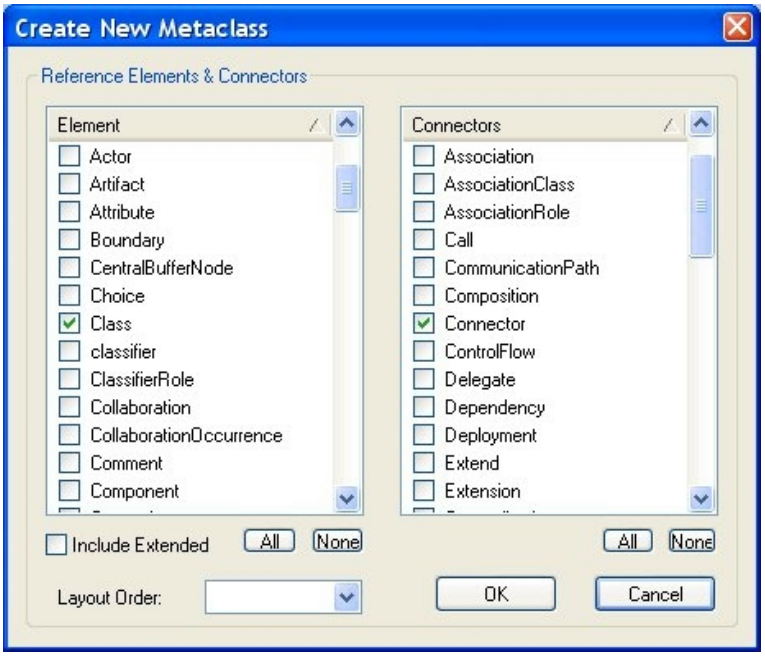

Abbildung 4.6: Dialog *Create New Metaclass*

und `SecondaryConcernUnitConnector` von `Connector` abgeleitet. Dazu wird zunächst der Menüpunkt *Metaclass* aus der Werkzeugleiste *Profile* in das Diagramm gezogen. Das Dialogfenster *Create New Metaclass* öffnet sich, in dem die Metaklassen, aus denen abgeleitet werden soll, ausgewählt werden. In diesem Fall sind dies `Package` und `Class` aus der Spalte *Element* und `Connector` aus der Spalte *Connectors* (Abbildung 4.6). Die drei Metaklassen erscheinen nach einem Klick auf *[OK]* im Diagramm. Durch Ziehen des Menüpunktes *Stereotype* in das Diagramm wird ein neues `Stereotype`-Element erzeugt. Im dazugehörenden *Properties*-Dialogfenster legt man den Namen des Stereotyps fest. Nach Erzeugung des Stereotyps muss noch festgelegt werden, von welcher Metaklasse dieser erbt. Der Beziehungstyp *Extension* aus der Werkzeugleiste wird ausgewählt und im Diagramm vom Stereotyp hin zur Metaklasse gezogen. Die Erzeugung eines neuen Stereotyps muss im vorliegenden Fall offensichtlich viermal durchgeführt werden. Man erhält nach Erzeugung der Stereotypen das bereits zuvor erwähnte Diagramm aus Abbildung 4.4.

Für den Stereotyp `Concern` muss zusätzlich die Darstellung angepasst werden. Aus dem sich durch einen Rechtsklick auf *Concern* im Diagramm öffnenden Kontextmenü wird die Option *Attributes* ausgewählt. Es öffnet sich ein Dialog *Concern Attributes*, in dem `_image` in das Feld *Name* eingegeben wird. Ein Klick auf den Button *[...]* neben dem Feld *Initial* öffnet das Dialogfenster *Shape Editor*, in dem die Darstellung der *Concern*-Elemente durch ein Shape Script festgelegt wird:

```
1 shape main{
2         editableField = "name";
3         h_align = "center";
4         v_align = "center";
5         setfillcolor(0,255,0);
```

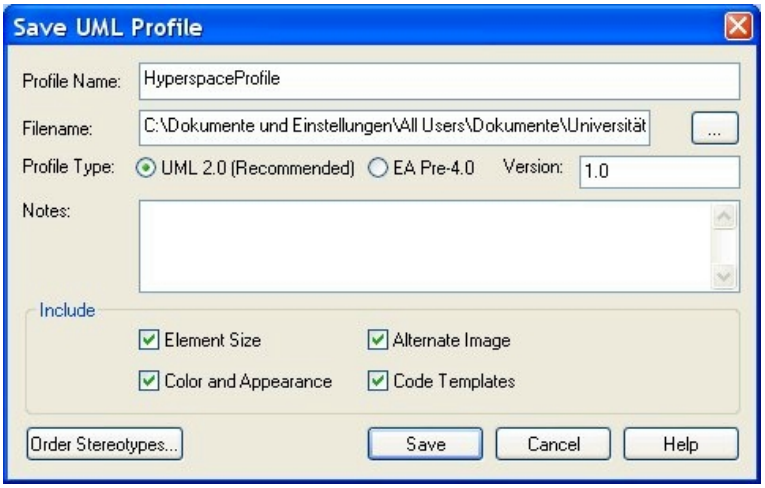

Abbildung 4.7: Dialog *Save UML Profile*

```
6           ellipse(0,0,100,75);
7           println("Concern:");
8           println("#name#");
9   }
```

Die so festgelegten Parameter definieren die Erscheinungsform eines Belangelements im Hyperspace-Diagramm. `editableField = ‘‘name‘‘` legt fest, dass der *Name* des Belangs veränderbar ist. Die beiden `println`-Befehle am Ende bewirken eine entsprechende Beschriftung des Belangs. `h_align` und `v_align` legen die horizontale und vertikale Ausrichtung des Belangelements als *zentriert* fest. Mit dem `setfillcolor`-Befehl definiert man die Farbgebung des Belangs im RGB-Modus, hier in *grün*. `ellipse` legt schließlich die Form eines Belangs als *Ellipse* mit *x- und y-Koordinate* 0, einer *Breite* von 100 und einer *Höhe* von 75 Pixeln fest.[1]

Die Erstellung des Shape Scripts wird durch Klicks auf die Buttons *[OK]* und *[Close]* abgeschlossen.

Die verschiedenen erzeugten und angepassten Stereotypen werden zum Schluss als UML-Profil exportiert. Dazu öffnet man durch einen Rechtsklick auf das Klassendiagramm das Kontextmenü und wählt die Option *Save as Profile...* aus. Im sich öffnenden Dialogfenster werden der Name (hier: `HyperspaceProfile`) und der Speicherort gewählt, für die übrigen Einstellungen können die Default-Einstellungen beibehalten werden. Ein Klick auf den *[Save]*-Button bewirkt schließlich die Erzeugung des Hyperspace-Profils (Abbildung 4.7).

Nutzung des UML-Profils

Das UML-Profil für die Umsetzung des Hyperspace-Modells lässt sich mit wenigen Schritten in ein bestehendes EA-Project einbinden: Man wählt das Fenster *Resources*

[1] Eine vollständige Referenz der möglichen Befehle in Shape Scripts sowie eine umfangreiche Erläuterung der Nutzung des Shape Editors finden sich in Kapitel 16.3 von [Spa08].

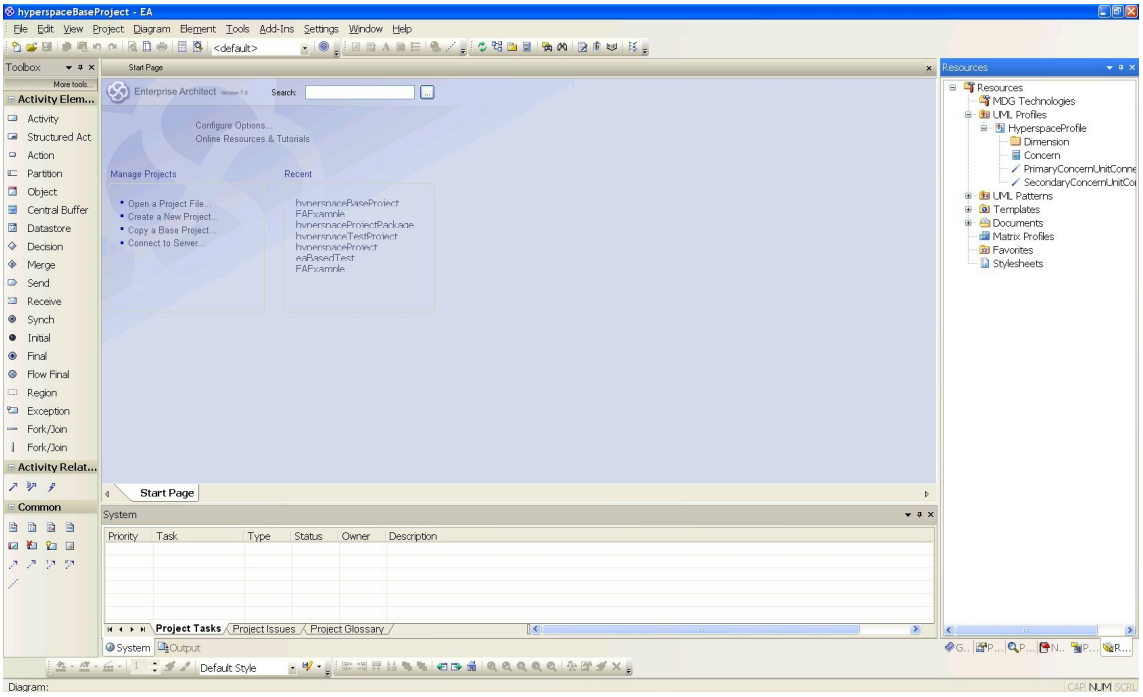

Abbildung 4.8: Enterprise Architect mit geöffnetem *Resources*-Fenster (rechts im Bild)

(gewöhnlich am rechten Bildschirmrand) aus. Ein Rechtsklick auf *UML Profiles* öffnet ein Kontextmenü, in dem die Option *Import Profile* gewählt wird. Es öffnet sich ein Dialogfenster *Import UML Profile*, in dem über das Feld *Filename* das zu importierende UML-Profil ausgewählt wird, hier `HyperspaceProfile.xml`. Ein Klick auf den Button *[Import]* macht die im Profil hinterlegten Stereotypen für das aktuelle Projekt verfügbar.

Die zur Verfügung stehenden Stereotypen können aus dem *Resources*-Fenster in diverse Diagramme gezogen werden. Für eine optimale Nutzung sollte man jedoch eine zusätzliche Sicht erzeugen, wie im Abschnitt 4.3.2 beschrieben.

4.3.2 Die Hyperspace-Sicht

Neben der Erzeugung eines passenden UML-Profils empfiehlt sich für die Umsetzung des erweiterten Hyperspace-Modells auch die Erstellung einer zusätzlichen Sicht in Enterprise Architect. Diese View wird das Diagramm des Hyperspaces sowie die zu jeder Dimension gehörenden Klassendiagramme enthalten.

Die Erstellung einer zusätzlichen Sicht gestaltet sich simpel: In einem geöffneten EA-Projekt öffnet man im *Project Browser* durch einen Rechtsklick auf *Model* ein Kontextmenü, aus dem die Option *New View* ausgewählt wird. Daraufhin öffnet sich das Dialogfenster *Create New View* (Abb. 4.9), in dem neben dem Namen der neuen View (`Hyperspace View`) auch die Art des View-Icons definiert wird (*Set View Icon Style*). Die Auswahl von *Simple View* bietet sich hier an. Durch

Abbildung 4.9: Dialog *Create New View*

Anklicken des Buttons *[OK]* wird die neue Sicht erzeugt und erscheint im Project Browser. In die neu erzeugte Sicht muss nun das Hyperspace-Diagramm eingefügt werden. Dazu öffnet man wiederum durch einen Rechtsklick auf den neu erzeugten Menüpunkt *Hyperspace View* im Project Browser ein Kontextmenü, aus dem man *Add/Add Diagram* auswählt. Im sich öffnenden Dialogfenster *New Diagram* definiert man einen Namen (`Hyperspace Diagram`) und wählt als Diagrammtyp *Package*. Ein Klick auf *[OK]* schließt den Dialog und erzeugt in der Hyperspace-Sicht das gewünschte Paketdiagramm.

4.3.3 Die Modellierung des Hyperspace

Nach dem Import des Hyperspace-Profils und der Erzeugung einer Hyperspace-Sicht mit dem dazugehörenden Hyperspace-Diagramm kann mit der Modellierung des Hyperspaces selbst begonnen werden. Hierzu werden das *Hyperspace Diagram* und das *Resources*-Fenster geöffnet. Aus dem Hyperspace-Profil in *Resources* kann man nun durch Ziehen des *Dimension*-Icons in das Hyperspace-Diagramm Dimensionen einfügen. Es öffnet sich jeweils ein Dialogfenster *New Package Name*, in dem man den Namen der Dimension festlegen und mit einem Klick auf *[OK]* bestätigen kann.

Um Belange zu einer Dimension hinzufügen zu können, muss man zunächst einmalig ein Klassendiagramm für die jeweilige Dimension erzeugen - ein Rechtsklick auf die Dimension im Project Browser und das Auswählen von *Add/Add Diagram* und des Diagrammtyps *Class* im sich öffnenden Dialogfenster genügen. Das erzeugte Klassendiagramm öffnet sich automatisch. Für das spätere Öffnen des Klassendiagramms einer Dimension genügt ein Doppelklick auf die Dimension im Hyperspace-Diagramm.

Wie das Einfügen von Dimensionen in das Hyperspace-Diagramm funktioniert auch das Einfügen von Belangen in eine Dimension: Aus dem *Resources*-Fenster

zieht man das Icon *Concern* in das Klassendiagramm der Dimension. Der neue Belang erscheint als grüne Ellipse im Diagramm, zudem öffnet sich ein Dialogfenster *Properties*, in dem man den Namen des Belangs definiert.

Um den Belangen einer Dimension Units zuzuordnen, wählt man deren Icons an entsprechender Stelle im Project Browser aus (Klassen befinden sich im *Class Model*, Anwendungsfälle im *Use Case Model* usw.) und zieht diese in das Klassendiagramm der Dimension. Daraufhin öffnet sich ein Dialogfenster *Paste Element*, in dem die Option *as Simple Link* ausgewählt wird. Nach der Bestätigung durch *[OK]* erscheint das Artefakt im Klassendiagramm. Bestehende Relationen zwischen Units werden so zwar ebenfalls in das Diagramm eingebracht, können aber nach einem Rechtsklick auf die Relation durch Auswahl der Option *Set Visibility/Hide Connector* im Kontextmenü versteckt werden.

Die Unit wird - abhängig davon, ob es sich um den Belang einer artefaktbasierenden oder einer selektierenden Belangdimension handelt - mittels eines *PrimaryConcernUnitConnector* oder eines *SecondaryConcernUnitConnector* mit dem jeweiligen Belang einer Dimension verbunden. Dieser wird ebenfalls aus dem *Resources*-Fenster in das Klassendiagramm und dort auf den jeweiligen Belang gezogen. Im erscheinenden Dialogfenster *Set link target* wählt man das Artefakt aus und bestätigt mit *[OK]*.

Kapitel 5 zeigt die Modellierung des Hyperspaces noch einmal anhand eines Anwendungsbeispiels.

4.4 Modifikationen am Überführungstool

Wie bereits in Kapitel 3 verdeutlicht wird, finden sowohl *JGraLab* als auch Elmar Brauchs *Überführungstool* sowie das *EA-API* in Hynalysis Verwendung. Sind die APIs von Enterprise Architect und JGraLab vergleichsweise einfach zu nutzen, treten im Hinblick auf die Nutzung des Überführungstools Probleme auf. Dadurch werden Modifikationen am Überführungstool nötig, die nachfolgend erläutert werden.

4.4.1 Identifikation von Diagrammelementen

Um Diagrammelemente im Enterprise Architect zweifelsfrei zu identifizieren, wird deren globale, eindeutige ID, die *GUID* benötigt. Diese kommt in Hynalysis zum Einsatz, wenn die Ergebniselemente im Original-Repository eingefärbt werden sollen: Die durch die Auswertung der GReQL-Anfrage ermittelten Lösungselemente müssen nach ihrer Überführung aus `JValue` in Instanzen der jeweiligen JGraLab-Klassen gegen das Original-EA-Repository abgeglichen werden. Stimmen die *GUID*s des EA-Diagrammelements und des JGraLab-Elements überein, so wird das Element im Original-Repository durch Aufruf der Methode `SetAppearance()` der Klasse `Element` aus dem EA-API durch eine Änderung der Rand- und Schriftfarbe sowie eine Änderung der Randdicke kenntlich gemacht.

Das dem Überführungstool zugrundeliegende Schema sieht die *GUID* als Attribut eines Elements oder auch eines Konnektors in der ursprünglichen Version jedoch nicht vor. Entsprechend muss im zugrundeliegenden Schema zur eindeutigen Identifikation von Diagrammelementen das Attribut `guid` den Knotenklassen `Pakkage` (und damit auch `Model`), `Diagram` und `Element` sowie der Kantenklasse `Connector` hinzugefügt werden[2]. Diese Ergänzungen werden in der Klasse `SchemaGenerator` des Überführungstools jeweils mit Hilfe der Methode `addAttribute()` vorgenommen.

Die in Abschnitt 3.3 vorgestellte und in Abbildung 3.6 gezeigte ursprüngliche Architektur des Überführungstools muss aber noch über diese Klasse hinaus verändert werden: Die Methoden `createModel()`, `createPackage()`, `createElement()` und `createConnector()` der Klasse `JGraLabWriter` werden so angepasst, dass beim Schreiben des JGraLab-Repositorys die jeweilige GUID des Modells, Pakets, Elements oder Konnektors gesetzt wird. Dazu werden die Parameterlisten der Methoden um die GUID erweitert und innerhalb der Methoden jeweils die zur GUID gehörende `set()`-Methode (`set_elementGuid()` etc.) eingefügt.

Die Klasse `JGraLabWriter` erbt von der Klasse `Writer`. Sie implementiert deren abstrakte `create()`-Methoden, die bereits im vorherigen Absatz vorgestellt wurden. Aus den Änderungen der Signaturen in `JGraLabWriter` ergeben sich notwendige Anpassungen der Klasse `Writer`: Auch hier werden die Signaturen um die GUID erweitert. Darüber hinaus wird `Writer` in ein Interface geändert, so dass die Modifier `abstract` entfallen.

Die Änderungen an `Writer` ziehen Änderungen an der davon erbenden Klasse `EAWriter` nach sich: Dort werden ebenfalls die Signaturen durch Erweiterung der Parameterlisten um die GUID angepasst. Im Gegensatz zu den `create()`-Methoden der Klasse `JGraLabWriter` müssen die `create()`-Methoden hier jedoch nicht um die `set()`-Methoden für die jeweilige GUID erweitert werden. Sie werden zur Erfüllung der Überführungsfunktionalität von JGraLab nach EA nicht benötigt.

Schließlich müssen auch die Klassen `TransformerJGraLabToEA` sowie `TransformerEAToJGraLab` angepasst werden. In erstgenannter Klasse werden lediglich die Aufrufe der Methoden aus `EAWriter` angepasst. Da der jeweils für die GUID übergebene Wert nicht verwendet wird, genügt ein `null` (ein beliebiger String ist ebenso möglich).

Die Anpassungen an der Klasse `TransformerEAToJGraLab` erfolgen quasi analog zu denen im vorhergehenden Absatz. Hier wird jedoch der als GUID zu übergebende Wert weiterverwendet. Zur Ermittlung desselben dienen die Methoden `GetDiagramGUID()`, `GetElementGUID()`, `GetPackageGUID()` bzw. `GetConnectorGUID()`.

4.4.2 Auswahl von Diagrammelementen

Ebenso wie die GUID findet auch die Selektion von Diagrammelementen im ursprünglichen Schema des Überführungstools keine Berücksichtigung. Lässt sich die

[2]Das Attribut `guid` ist für die genannten Knotenklassen als `packageGuid`, `diagramGuid`, `elementGuid` bzw. `connectorGuid` umgesetzt.

Frage, ob ein Diagrammelement im EA ausgewählt ist oder nicht, durch das Hinzufügen eines Attributs `isSelected` zu den Klassen `Element` und `Connector` analog zum Hinzufügen der GUID im Schema umsetzen, gestaltet sich das Setzen des jeweiligen Wertes theoretisch wesentlich umfangreicher und diffiziler.

Nachfolgend werden die Vorgehensweise und die daraus resultierenden notwendigen Änderungen am Überführungstool beschrieben, die im Falle einer tatsächlichen Auswählbarkeit von Diagrammelementen im Enterprise Architect erfolgen müssen. Da das EA-API in der im Rahmen der Studienarbeit verwendeten Version jedoch die entsprechenden Methoden zwar anbietet, diese aber in der Entwicklungs- und Testphase von Hynalysis ausschließlich Fehler und keine brauchbaren Ergebnisse lieferten, ist schließlich auf die Auswahl von Diagrammelementen verzichtet worden. Nähere Erläuterungen zu dieser Problematik finden sich in Kapitel 6.

Die ausgewählten Diagrammelemente lassen sich nur über die Methoden `GetSelectedElements()` und `GetSelectedConnector()` der Klasse `Diagram` des EA-API ermitteln. `GetSelectedElements()` liefert eine `Collection` der ausgewählten Elemente, `GetSelectedConnector()` den ausgewählten Konnektor[3].

Die Klasse `TransformerEAToJGraLab` muss zunächst um eine Collection von `Collection`s für die ausgewählten Diagrammelemente sowie eine Collection von `Connector` für die in den verschiedenen Diagrammen ausgewählten Konnektoren erweitert werden.

In der Methode `transformPackageContent()` muss der erstgenannten Collection für jedes Diagramm, welches in der Iteration durch das Paket durchlaufen wird, eine Collection der ausgewählten Elemente hinzugefügt werden. Auf die so entstandene Collection von Collections muss dann in der Iteration durch die Elemente des Pakets zugegriffen werden: Ist ein Element in einer der Collections enthalten, so muss es auch im zu erzeugenden TGraphen als selektiert gekennzeichnet werden. Die Signatur der Methode `createElement()` aus `JGraLabWriter` muss dementsprechend neben der GUID um einen weiteren, boolschen Parameter `isSelected` erweitert werden, über den das gleichnamige Attribut des Elements gesetzt wird.

Daneben muss in der Iteration durch die Diagramme eines Pakets für jedes durchlaufene Diagramm der ausgewählte Konnektor bestimmt und zur letztgenannten Collection hinzugefügt werden. Die so entstandene Collection wird in der Methode `transformRelationships()` der Klasse `TransformerEAToJGraLab` benötigt: Hier werden die Konnektoren jedes Elements eines Modells durchlaufen. Dabei muss für jeden Konnektor geprüft werden, ob er in der Collection enthalten ist. Ist dies der Fall, muss das Attribut `isSelected()` des Konnektors im TGraph gesetzt werden. Entsprechend muss die Methode `createConnector()` der Klasse `JGraLabWriter` ebenfalls um den Parameter sowie das Setzen des Attributs erweitert werden.

Das Attribut `isSelected` wurde im Rahmen der Anpassungen am Überführungstool umgesetzt, hat jedoch aufgrund der zu Beginn dieses Abschnitts beschriebenen Problematik noch keine Bedeutung für die Funktionalität des Überführungstools.

[3]Es wird deutlich, dass man in einem Diagramm in EA zwar mehrere Elemente, jedoch nur *höchstens einen* Konnektor auswählen kann. Der Konnektor ist in der Collection, die von `GetSelectedElements()` zurückgegeben wird, nicht enthalten.

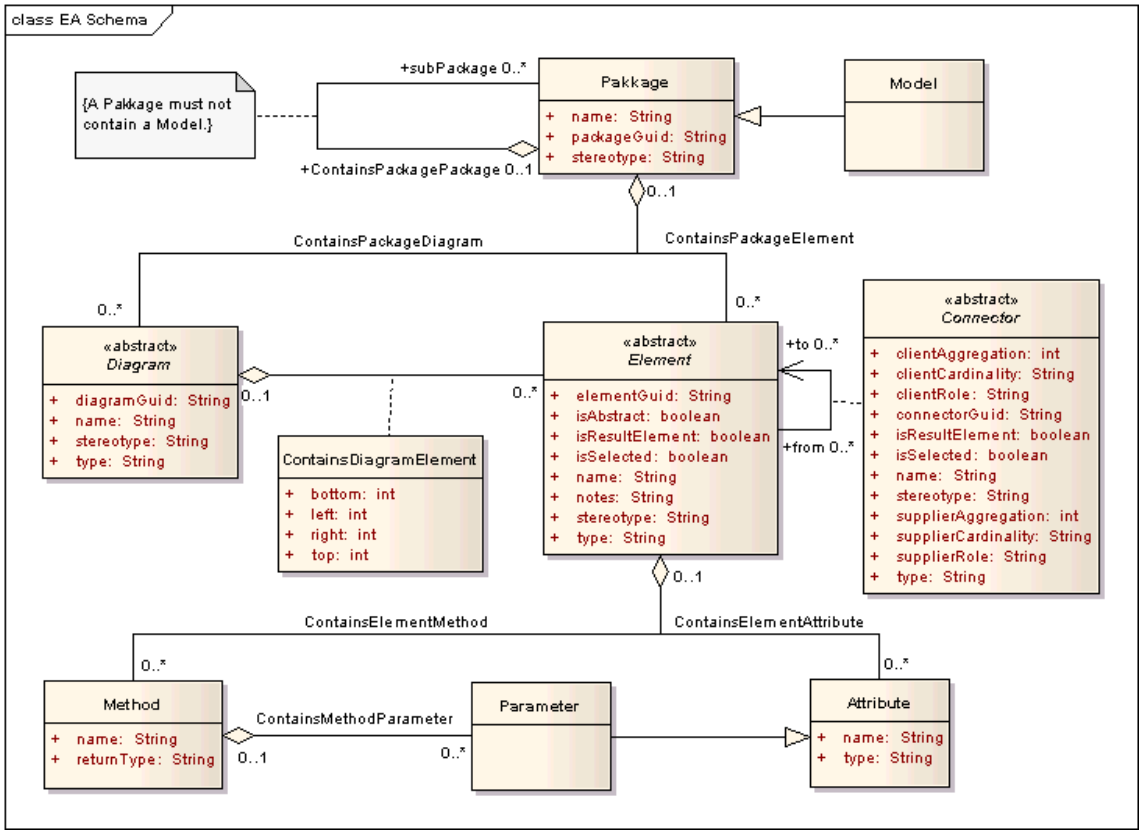

Abbildung 4.10: Überarbeitetes JGraLab-Schema für EA

4.4.3 Markierung von Diagrammelementen als Ergebniselemente

Die Markierung von Elementen als Ergebniselemente der Auswertung einer GReQL-Anfrage lässt sich im Enterprise Architect durch Änderungen der Darstellung derselben verwirklichen. In einem TGraphen besteht diese Möglichkeit hingegen nicht. Um auch in dem per Auswertung einer GReQL-Anfrage zu analysierenden Graphen die Elemente, welche Ergebniselemente der Anfrageauswertung sind, zu markieren, muss im Schema zur Knotenklasse `Element` sowie zur Kantenklasse `Connector` das boolsche Attribut `isResultElement` hinzugefügt werden. Diese Ergänzungen erfolgen analog zu denen von `guid` und `isSelected`.

`isResultElement` wird bei der Überführung von EA nach JGraLab standardmäßig für alle Instanzen von `Element` und `Connector` auf `false` gesetzt. Änderungen an diesem Wert werden erst bei der Weiterverarbeitung der Anfrageergebnisse durch Hynalysis notwendig.

Abbildung 4.10 zeigt das aus den vorangegangenen Überlegungen resultierende überarbeitete JGraLab-Schema für EA.

4.4.4 Anzahl laufender Instanzen des Enterprise Architect

Im Laufe der Tests von Hynalysis fiel auf, dass bei der Nutzung des Überführungstools Instanzen von Enterprise Architect zwar gestartet, aber nicht mehr geschlossen wurden. So waren nach einem Durchlauf von Hynalysis meist mehrere laufende Instanzen von EA zu finden.

Die Ursache liegt in einem fehlenden Funktionsaufruf im Überführungstool: Zwar wird die Methode `CloseFile()` der Klasse `Repository` aufgerufen und schließt so das geöffnete EA-Repository. Die laufende Instanz des Enterprise Architect wird aber nicht geschlossen. Ein Aufruf der Methode `Exit()` der Klasse `Repository`, welcher in der Klasse `TransformerEAToJGraLab` in der Methode `transform()` sowie in der Methode `closeEARepository()` der Klasse `EAWriter` erfolgt, behebt dieses Problem für beide Überführungsrichtungen.

Die sich auf der beiliegenden CD-ROM befindende Version des Überführungstools enthält alle zuvor vorgestellten umgesetzten Änderungen.

4.5 Die Architektur von Hynalysis

Hynalysis gliedert sich in zwei Komponenten: `GUI` und `Controller`. Während das GUI ausschließlich die grafischen Bedienungskomponenten enthält, stellt die Controller-Komponente die gesamte notwendige Funktionalität von Hynalysis zur Verfügung und ermöglicht zudem die Nutzung des Tools über die Kommandozeile. Dazu nutzt die Controller-Komponente die schon mehrfach erwähnten externen Komponenten EA-API, JGraLab und das Überführungstool. Abbildung 4.11 zeigt das Komponentendiagramm, Abbildung 4.12 (S. 66) ein stark vereinfachtes Klassendiagramm von Hynalysis.

Detailliertere Klassendiagramme zu den beiden Komponenten finden sich in den Abschnitten 4.5.1 und 4.5.2. Diese Abschnitte erläutern knapp die Architektur der Komponenten und die wichtigsten Merkmale der sie realisierenden Klassen, können aber nicht die auf der beiliegenden CD-ROM zu findende Javadoc-Dokumentation ersetzen.

4.5.1 Die Controller-Komponente

Die Controller-Komponente setzt sich aus den vier Klassen `Controller`, `GReQLManager`, `ResultManager` und `UserCommunicator` zusammen, welche sich im Paket `de.uni.koblenz.hynalysis.controller` befinden. Das Klassendiagramm in Abbildung 4.13 auf Seite 67 bietet eine Übersicht über die Klassen der Controller-Komponente.

Controller

Die Klasse `Controller` ist die Hauptklasse von Hynalysis. Sie ist als Singleton konzipiert. Neben dem privaten Konstruktor, der stattdessen nutzbaren Methode

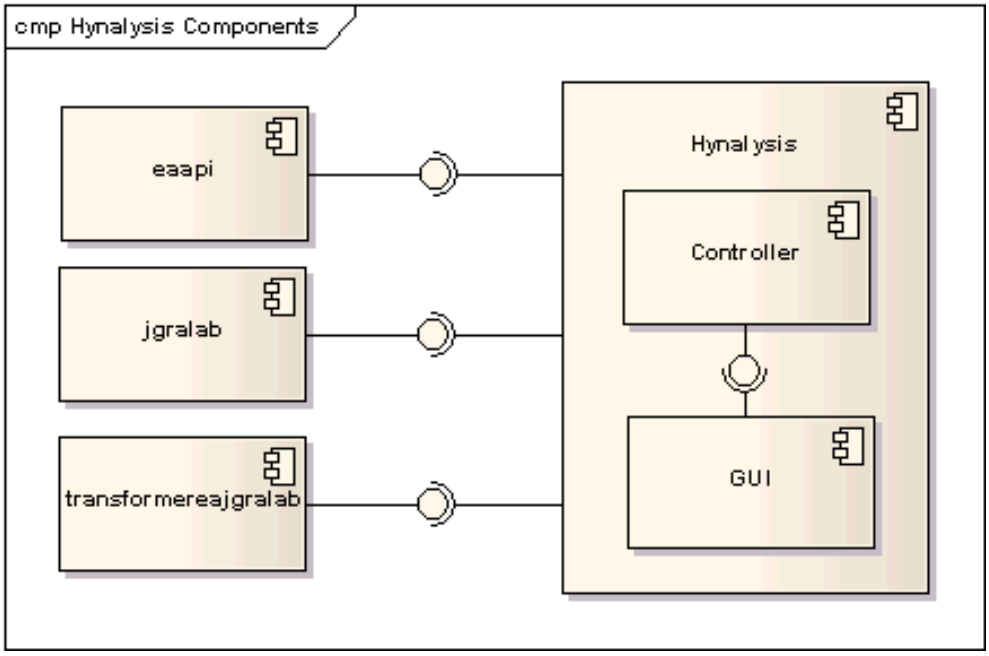

Abbildung 4.11: Komponentensicht auf Hynalysis

`createController()` sowie verschiedenen Gettern und Settern enthält `Controller` im Wesentlichen die folgenden drei Methoden:

`main()` ermöglicht die Nutzung von Hynalysis über die Kommandozeile. Nach einem Aufruf von `createController()` zur Erzeugung einer Instanz von `Controller` folgt ein Aufruf von `run()`, welche den weiteren Programmablauf in der Kommandozeile steuert.

`run()` entscheidet zunächst anhand des Flags `doAgainSelected`, ob vom Nutzer ein weiterer Durchlauf von Hynalysis gewünscht wurde. Davon abhängig wird innerhalb einer `while`-Schleife zunächst die Methode `getUserChoices()` der Klasse `UserCommunicator` aufgerufen. Danach wird der Status des Flags angepasst.

`analyzeHyperspace()` startet, sofern der Nutzer nicht die Nutzung eines schon bestehenden TGraphen gewählt hat, zuerst die Methode `transform()` der Klasse `Controller` aus dem Überführungstool. Damit wird, basierend auf den Nutzereingaben in der Kommandozeile bzw. im GUI, das EA-Repository in einen TGraphen überführt. Für den neu erzeugten oder den bereits bestehenden Graphen wird dann durch die Methode `evaluateGReQLQuery()` der Klasse `GReQLManager` die GReQL-Anfrage ausgewertet. Diese wird wahlweise aus einer externen Datei geladen oder kann über GUI oder Kommandozeile vom Nutzer eingegeben werden. Das Anfrageergebnis wird schließlich durch die Methoden der Klasse `ResultManager` aufbereitet und an die vom Nutzer gewählten Ausgabeparameter angepasst.

Von den hier bisher nicht genannten, kleineren Helfermethoden der Klasse `Controller` sind noch die boolschen Methoden `checkEAPath()`, `checkGReQLPath()` und `checkTGraphPath()` hervorzuheben: Sie prüfen die vom Nutzer angegebenen Pfade der Quell- und Zieldateien anhand ihrer Suffixe auf mögliche Richtigkeit.

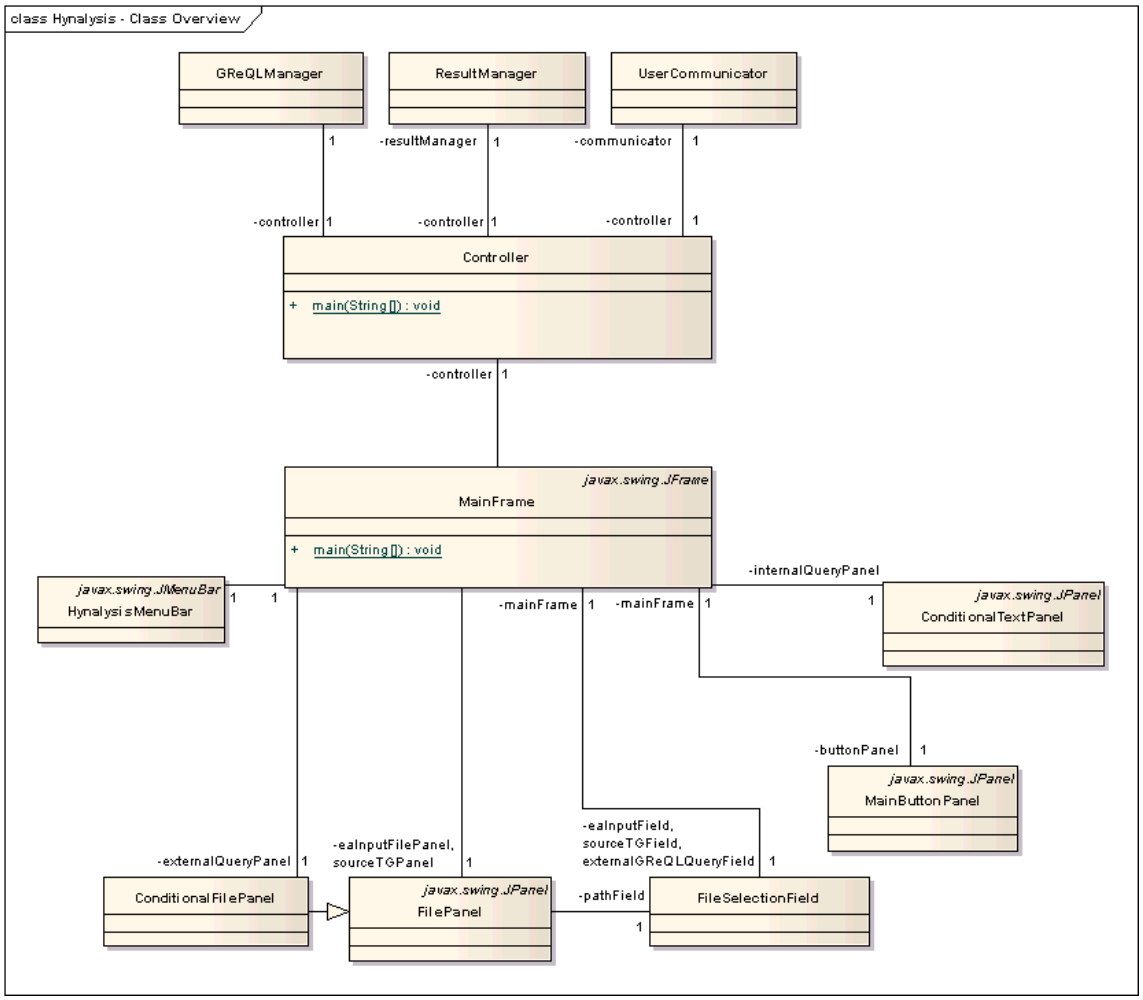

Abbildung 4.12: Übersicht über die Klassen von Hynalysis

Stimmt das Suffix eines Pfades für ein EA-Repository mit `.eap` bzw. `.EAP` überein, so ist der Rückgabewert von `checkEAPath()` `true`, andernfalls `false`. Analog führen die beiden anderen Methoden Prüfungen auf die Dateiendungen `.tg` und `.greql` aus.

GReQLManager

Die Klasse `GReQLManager` enthält eine Methode:

`evaluateGReQLQuery()` erzeugt eine Instanz der Klasse `GReQLEvaluator` aus JGraLab. Für die GReQL-Anfrage wird dann die Methode `startEvaluation()` aus `GReQLEvaluator` aufgerufen, die die Anfrage auswertet. Der Rückgabewert der Methode ist das Anfrageergebnis in Form einer Instanz von `JValue`.

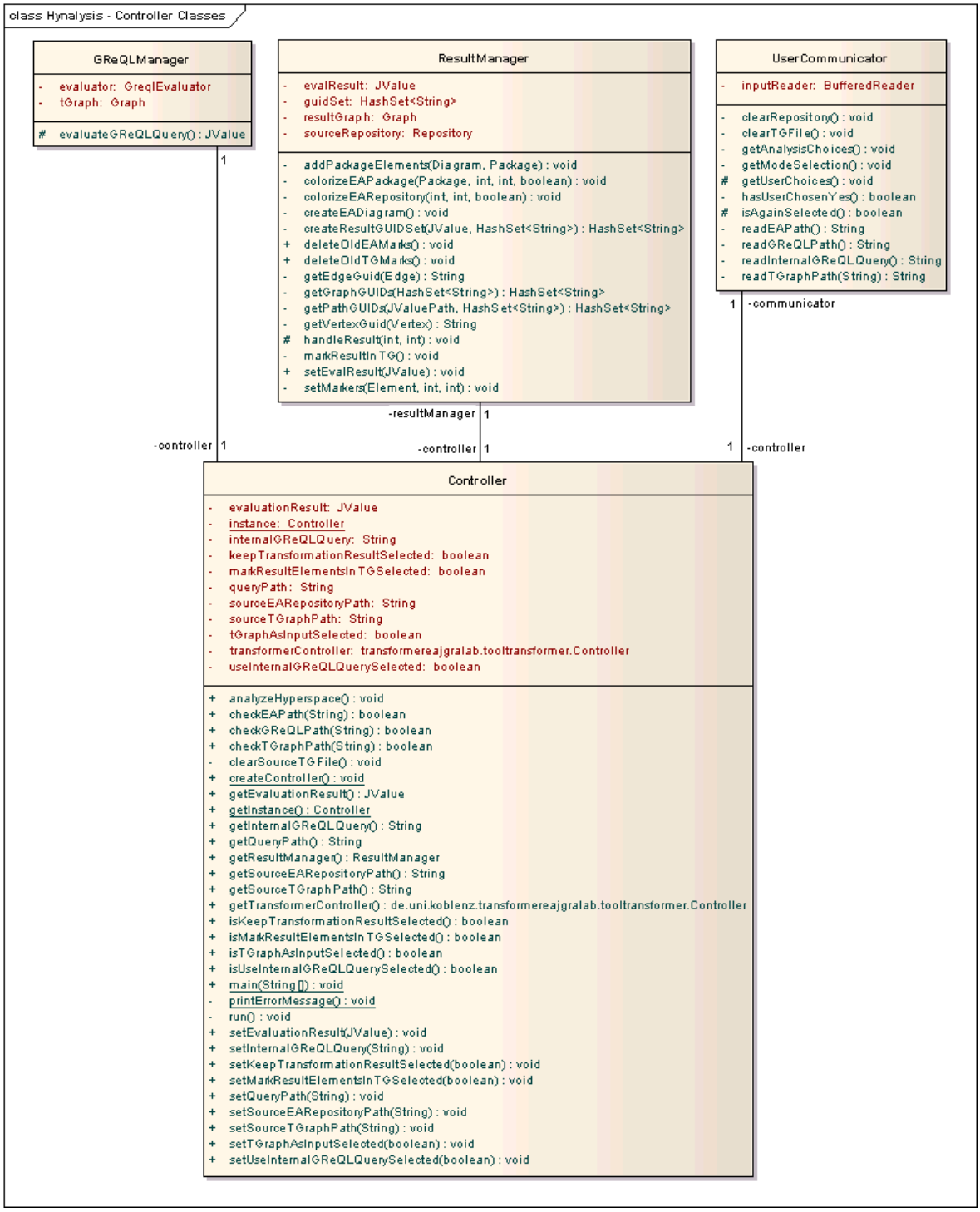

Abbildung 4.13: Klassendiagramm zu `de.uni.koblenz.hynalysis.controller`

ResultManager

Die Klasse `ResultManager` enthält sämtliche Methoden zur Verarbeitung des Ergebnisses der GReQL-Anfrage. Von den insgesamt 15 Methoden sollen hier die wichtigsten vorgestellt werden:

Die Methode `handleResult()` löscht zunächst evtl. bestehende Markierungen aus einer vorherigen Analyse im zu analysierenden EA-Repository sowie, falls erforderlich, im dazugehörenden TGraphen. Daraufhin werden zuerst die GUIDs der Elemente aus dem als JValue zur Verfügung stehenden Anfrageergebnis mit der Methode `createResultGUIDSet()` im HashSet `guidSet` gesammelt. Mit Hilfe der GUIDs erfolgt dann in der Methode `colorizeEARepository()` das Einfärben der Ergebniselemente im EA-Repository. Im Anschluss werden die Ergebniselemente durch Setzen des Flags `isResultElement` auch im TGraphen markiert. Schließlich wird im EA-Repository durch Aufruf der Methode `createEADiagram()` ein zusätzliches Diagramm erzeugt, welches ausschließlich die Ergebniselemente der GReQL-Anfrage enthält.

Die Methode `deleteOldTGMarks()` löscht bestehende Markierungen in einem TGraphen, indem sie diesen durchläuft und dabei dem Attribut `isResultElement` jedes Elements den Wert `false` zuweist.

`deleteOldEAMarks()` ruft im Gegensatz dazu lediglich die Methode `colorizeEARepository()` auf und übergibt dieser als Attribute die im Enterprise Architect ursprünglich vorgesehenen Darstellungsparameter für die Farbe und Dicke des Klassenrahmens.

Die Methode `colorizeEARepository()` unterscheidet zwischen zwei verschiedenen Modi: Sind lediglich alte Markierungen zu löschen, so werden die Ränder aller Elemente im EA-Repository entsprechend den übergebenen Parametern angepasst. Andernfalls werden die Ergebnisobjekte mit dem der Anfrage zugrundeliegenden Repository verglichen. Kann ein Ergebniselement mit einem Element des EA-Repositorys identifiziert werden, wird dieses Element im Repository durch Veränderung der Randfarbe und -dicke markiert.

UserCommunicator

Die Klasse `UserCommunicator` enthält die notwendige Funktionalität zur Interaktion zwischen Hynalysis und dem Nutzer über die Kommandozeile. Entsprechend den Nutzereingaben werden verschiedene Dialoge durchlaufen und die jeweiligen Werte über `Controller` geprüft und gesetzt. Von `UserCommunicator` aus werden zudem die aus den Nutzereingaben resultierenden Operationen angestoßen.

Die Funktionalität in `UserCommunicator` wird hauptsächlich durch drei Methoden realisiert: `getUserChoices()` fragt zunächst den Pfad des zu analysierenden Repositorys ab und setzt diesen im Controller. Im Anschluss wird mit `getModeSelection()` überprüft, ob der Nutzer lediglich bestehende Markierungen aus dem Repository oder einem TGraphen entfernen oder eine Anfrage auf dem Repository ausführen möchte, und die ausgewählte Aktion angestoßen. Hat der Nutzer sich für eine Aus-

wertung einer GReQL-Anfrage entschieden, so werden in `getAnalysisChoices()` die weiteren nötigen Einstellungen ermittelt und die Analyse gestartet.

4.5.2 Die GUI-Komponente

Die wesentliche Klasse der GUI-Komponente ist die Klasse `MainFrame`. Sie nutzt die sechs anderen Klassen `HynalysisMenuBar`, `FilePanel`, `ConditionalFilePanel`, `FileSelectionField`, `ConditionalTextPanel` und `MainButtonPanel`. Abbildung 4.14 auf Seite 70 zeigt ein Klassendiagramm des Pakets `de.uni.koblenz.hynalysis.gui`, in dem sich diese Klassen befinden.

MainFrame

Die Klasse `MainFrame` definiert das Hauptfenster des GUI. Dazu nutzt sie Instanzen der übrigen Klassen der GUI-Komponente. Eine eigene `main()`-Methode ermöglicht die Nutzung von Hynalysis über das GUI.

Neben `main()` ist in `MainFrame` vor allem `actionPerformed()` von Bedeutung. Diese Methode implementiert die gleichnamige Methode aus dem Interface `ActionListener` des Pakets `java.awt.event`. Basierend auf den Nutzereingaben, werden über diese Methode nicht nur bestimmte Werte im Controller gesetzt, sondern bei Anklicken eines Buttons durch den Nutzer auch die entsprechenden Operationen angestoßen.

Darüber hinaus definiert und erzeugt `MainFrame` Informations- und Fehlerdialogfenster, die den Nutzer über den Abschluss einer Operation in Hynalysis bzw. über aufgetretene Fehler bei der Ausführung einer Operation in Hynalysis informieren.

HynalysisMenuBar

Die Klasse `HynalysisMenuBar` definiert die Menüleiste, welche sich am oberen Rand des Hauptfensters befindet. Sie ist bewusst einfach gehalten und hält nur eine Möglichkeit zum Schließen des Hauptfensters und somit auch Hynalysis vor.

FileSelectionField

Die Klasse `FileSelectionField` definiert eine Kombination eines `JTextFields` zur manuellen Eingabe eines Dateipfads und eines Explorer-Fensters zur komfortablen Auswahl einer Datei. Das Explorer-Fenster wird realisiert durch einen `JFileChooser` aus dem Paket `javax.swing`. Es wird immer dann geöffnet, wenn der Button im dazugehörenden `FilePanel` (s. nächster Abschnitt) angeklickt wird. Der Dateipfad im Textfeld wird bei einer Dateiauswahl im Explorer-Fenster angepasst.

FilePanel

Die Klasse `FilePanel` definiert ein `JPanel` mit einer Kombination aus einem `FileSelectionField` und einem *[Select]*-Button, der das entsprechende Explorer-

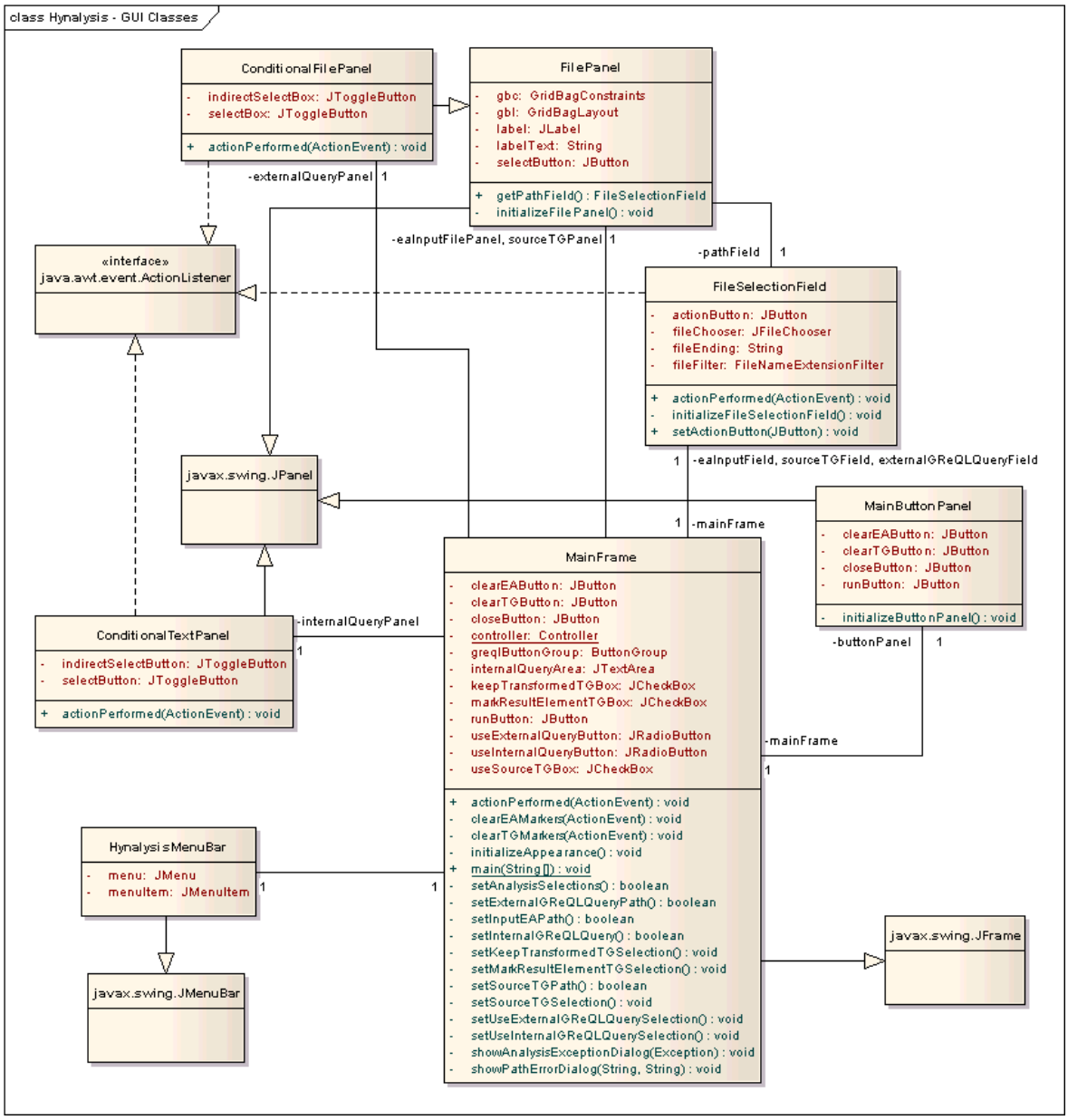

Abbildung 4.14: Klassendiagramm zu `de.uni.koblenz.hynalysis.gui`

Fenster öffnet. `FilePanel` kommt im GUI von Hynalysis für die Dateiauswahlfelder sowohl für das zu untersuchende EA-Repository als auch für den ausgewählten bzw. zu erzeugenden TGraphen zum Einsatz.

ConditionalFilePanel

Die Klasse `ConditionalFilePanel` erweitert die Klasse `FilePanel` durch Implementation des Interfaces `ActionListener`. Abhängig vom Wert eines JToggleButtons wird das gesamte Panel sichtbar oder unsichtbar gesetzt und entsprechend aktiviert oder deaktiviert. Im GUI von Hynalysis wird das Panel bei der Auswahl einer externen, d.h. zuvor in einer Datei gesicherten GReQL-Query eingesetzt: Wählt der Nutzer die Verwendung einer externen GReQL-Anfrage aus, so wird ein Dateiauswahlfeld für diese Anfrage eingeblendet.

ConditionalTextPanel

Die Klasse `ConditionalTextPanel` definiert ein TextPanel, dessen Sichtbarkeit, abhängig vom Wert eines `JToggleButtons`, gesetzt wird. Das Panel kommt im GUI bei der manuellen Eingabe einer GReQL-Anfrage durch den Nutzer zum Einsatz. Ist diese - im Gegensatz zur Nutzung einer externen Query - vom Nutzer gewünscht, verschwindet das Dateiauswahlfeld für die externe Anfrage und das Texteingabefeld erscheint.

MainButtonPanel

Die Klasse `MainButtonPanel` definiert die Button-Leiste am unteren Rand des GUI-Hauptfensters von Hynalysis. Diese Leiste enthält Buttons zum Entfernen von Markern aus dem gewählten EA-Repository, zum Entfernen von Markern aus dem gewählten TGraphen, zum Starten einer Anfrageauswertung sowie zum Schließen von Hynalysis.

5 Hynalysis im Einsatz

Nachdem im vorangegangenen Kapitel die verschiedenen Bereiche der Entwicklung von Hynalysis behandelt wurden, soll dieses Kapitel Hynalysis als Anwendung im Einsatz vorstellen.

Zunächst wird ein Beispielszenario vorgestellt, anhand dessen im weiteren Verlauf des Kapitels die Arbeit mit Hynalysis erläutert wird. Im Anschluss wird ein Überblick über die Installation der benötigten Komponenten sowie das GUI von Hynalysis geboten. Die Erläuterungen zur Auswertung einer GReQL-Anfrage mit Hilfe des GUI schließen das Kapitel ab.

5.1 Das Ausgangsbeispiel

Angenommen wird ein Softwareentwicklungsprojekt, in dessen Rahmen für ein Handelsunternehmen eine Software entwickelt werden soll, die die Personal-, die Auftrags- und die Kundenverwaltung ermöglicht sowie Funktionalitäten zur Buchführung bietet.

Die Entwickler haben in Enterprise Architect bereits ein Anwendungsfall- und ein Klassendiagramm erstellt. Beide Diagramme, welche in den Abbildungen 5.1 auf Seite 74 und 5.2 auf Seite 75 zu finden sind, sind offensichtlich unvollständig im Hinblick auf eine brauchbare Unternehmenssoftware. Jedoch handelt es sich hier erstens um ein Anwendungsbeispiel, das nicht die reelle Komplexität einer solchen Software widerspiegeln soll. Zweitens zeigt das Beispiel zugleich, dass das erweiterte Hyperspace-Modell und Hynalysis auch noch nach Beginn des Entwicklungsprozesses und in den frühen Phasen des Softwareentwurfs eingesetzt werden können.

Die dargestellten Anwendungsfälle bilden drei typische Geschäftsabläufe in einem Handelsunternehmen ab: *Calculate Salaries* ist die monatliche Gehaltsabrechnung für alle Mitarbeiter eines Unternehmens, welche von einem Mitarbeiter der Personalabteilung, hier als *Human Resources Administrator* bezeichnet, durchgeführt wird. *Prepare Financial Statement* bildet die Bilanzerstellung ab, die von einem Buchhalter oder, wie in diesem Fall, einem *Manager* erstellt wird. *Execute Order* schließlich stellt einen Auftrag eines Kunden, eines *Customers*, an das Unternehmen und die entsprechende Abwicklung dar. Auf Unternehmensseite nimmt ein Verkäufer, ein *Sales Assistant*, den Auftrag an und wacht über die ordnungsgemäße Abwicklung. In allen genannten Anwendungsfällen kommt die Unternehmenssoftware zum Einsatz.

Das Klassendiagramm zeigt die Modellierung der Klassen der Unternehmenssoftware: `Human Resources Manager` stellt die Funktionalitäten zur Personalverwaltung, z.B. für die Gehaltsabrechnung bereit. Sie greift auf die Klasse `Staffer`, von

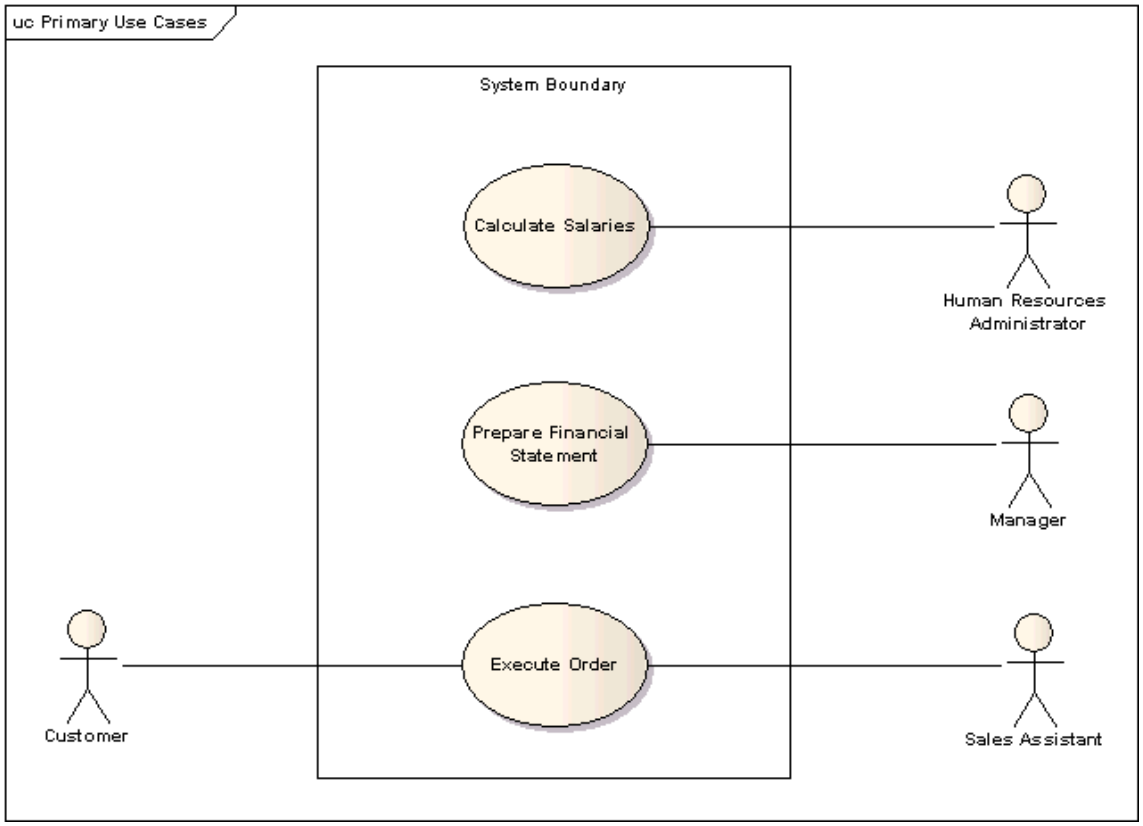

Abbildung 5.1: Anwendungsfalldiagramm des Anwendungsbeispiels

der die Klassen `Manager` und `Assistant` erben, zu. Instanzen dieser Klassen stellen verschiedene Mitarbeiter und deren relevante Stammdaten dar.

`CustomerManager` stellt Funktionalitäten zur Verwaltung von Kunden und deren Stammdaten, welche über Instanzen der Klasse `Customer` vorgehalten werden, bereit. `OrderManager` dient der Auftragserstellung und -abwicklung. Aufträge selbst werden über `Order` realisiert, die einzelnen bestellten Artikel oder Dienstleistungen über `OrderItem`.

`AccountingManager` schließlich stellt die Funktionalitäten zur Buchführung im Unternehmen bereit. Da hierunter Aufgaben wie die Bilanzerstellung fallen, muss `AccountingManager` auf diverse verschiedene Klassen zugreifen, damit z.B. die Personalkosten und die Einnahmen ermitteln werden können.

Die beiden hier vorgestellten Diagramme bestehen, wie eingangs erwähnt, bereits in einem EA-Projekt. In den folgenden Abschnitten wird dieses Projekt um ein Hyperspace-Diagramm ergänzt und dann mit Hilfe von Hynalysis untersucht.

5.2 Das Hyperspace-Diagramm

Damit in das bestehende EA-Projekt ein Hyperspace-Diagramm eingebunden werden kann, muss zunächst das UML-Profil importiert werden, welches die für

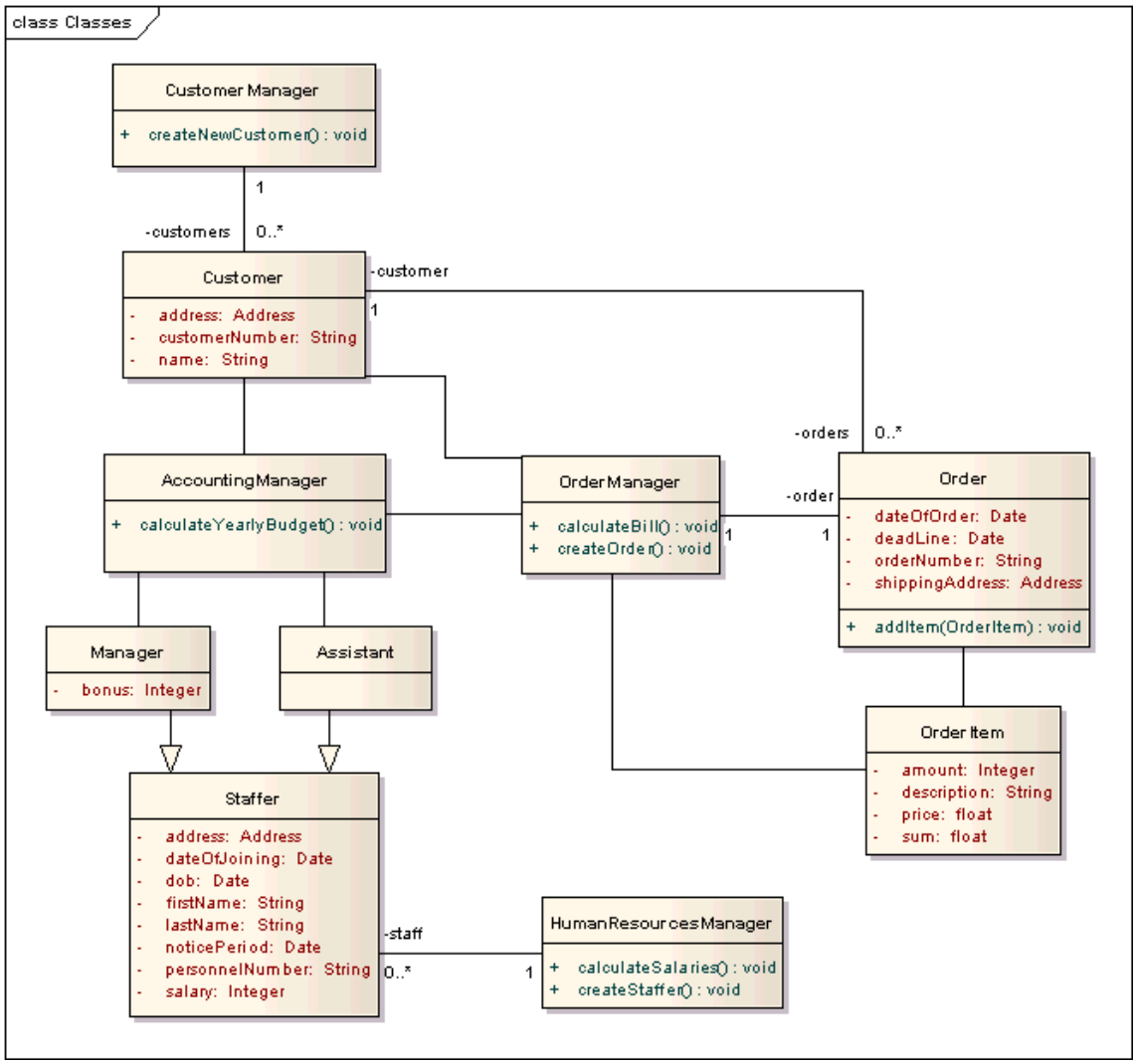

Abbildung 5.2: Klassendiagramm des Anwendungsbeispiels

Hyperspace-Diagramme benötigten Stereotypen bereitstellt. Die Datei `hyperspaceProfile.xml`, welche dieses Profil enthält, befindet sich auf der der Studienarbeit beiliegenden CD-ROM.

5.2.1 Import des Hyperspace-Profils

Hat man das Projekt im Enterprise Architect geöffnet, so befindet sich am rechten Rand des Fensters ein Feld mit verschiedenen Karteireitern am unteren Rand. Darin enthalten sind u.a. der *Project Browser*, der einen Überblick über die verschiedenen Objekte im Projekt bietet, und das Karteiblatt *Resources*, welcher für den Import des UML-Pakets ausgewählt werden muss. Abbildung 5.3 auf Seite 76 zeigt das geöffnete Fenster des Enterprise Architect. Das Karteiblatt *Resources* ist bereits geöffnet und in der Abbildung mit einem Rahmen versehen.

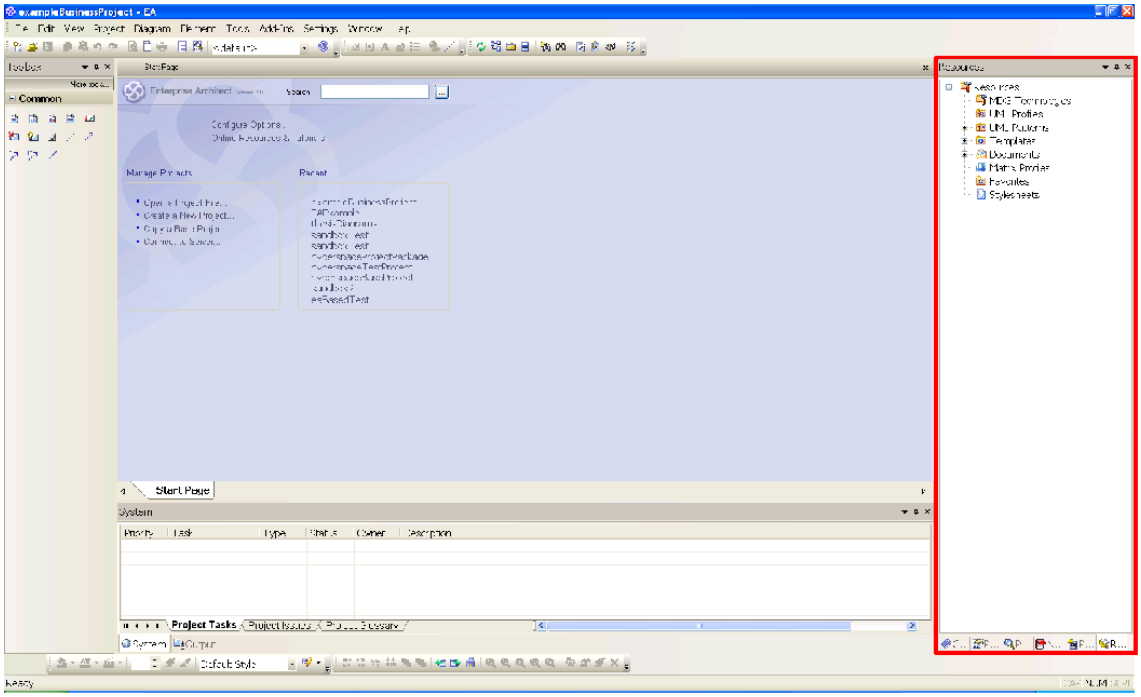

Abbildung 5.3: Enterprise Architect mit geöffnetem Fenster *Resources*

Innerhalb von *Resources*, in Abbildung 5.4 auf Seite 76 noch einmal als Ausschnitt vergrößert dargestellt, öffnet sich durch einen Rechtsklick auf *UML Profiles* ein Menü, aus dem die Option *Import Profile* ausgewählt werden muss. Im sich daraufhin öffnenden Fenster *Import UML Profile* öffnet man durch einen Klick auf den Button *[...]* das Datei-Auswahlfenster, in dem man die Datei `hyperspaceProfile.xml` auswählt. Die Optionen *Element Size*, *Color and Appearance*, *Alternate Image* und *Code Templates* unter *Import*, welche als Default-Werte gesetzt sind, können beibehalten werden. Ein Klick auf den Button *[Import]* importiert das Hyperspace-Profil und macht die darin enthaltenen Stereotypen für die Erstellung eines Hyperspace-Diagramms anwendbar.

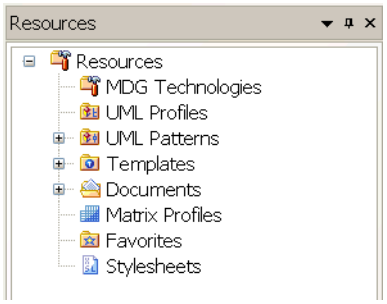

Abbildung 5.4: Vergrößerte Darstellung eines Ausschnitts von *Resources*

Abbildung 5.5: Dialog *Create New View*

5.2.2 Einrichtung der Hyperspace-Sicht

Um die durch den Import des UML-Profils gewonnen Möglichkeiten auch sinnvoll nutzen zu können, wird im Projekt eine zusätzliche Sicht *Hyperspace View* erzeugt, in der dann das Hyperspace-Diagramm angelegt wird.

Dazu muss im schon erwähnten Feld am rechten Rand des Fensters des Enterprise Architects der *Project Browser* durch einen Klick auf den entsprechenden Karteireiter geöffnet werden. Darin wird mit einem Rechtsklick auf *Model* ein Kontextmenü geöffnet, aus dem die Option *New View* ausgewählt wird. Es öffnet sich das Dialogfenster *Create New View*, in dem der Name der zu erzeugenden Sicht (`Hyperspace View`) sowie unter *Set View Icon Style* deren Typ (*Simple*) bestimmt wird. Ein Klick auf den Button *[OK]* schließt die Erzeugung der Sicht ab. Abbildung 5.5 zeigt das Dialogfenster mit den gesetzten Optionen.

5.2.3 Einfügen des Diagramms

Innerhalb der Hyperspace-Sicht wird nun das Hyperspace-Diagramm eingefügt. Im Project Browser wird durch einen Rechtsklick auf die neu erzeugte Sicht *Hyperspace View* ein Kontextmenü geöffnet, aus dem der Menüpunkt *Add* und im sich öffnenden Submenü *Add Diagram* ausgewählt werden.

Daraufhin öffnet sich das Dialogfenster *New Diagram*, in dem der Name (`Hyperspace Diagram`) sowie der Typ (*Select From: UML Structural*, *Diagram Types: Package*) des zu erzeugenden Diagramms bestimmt werden. Ein Klick auf den Button *[OK]* schließt das Dialogfenster; das Hyperspace-Diagramm wird erzeugt. Abbildung 5.6 auf Seite 78 zeigt das Dialogfenster *New Diagram* mit den gewählten Einstellungen.

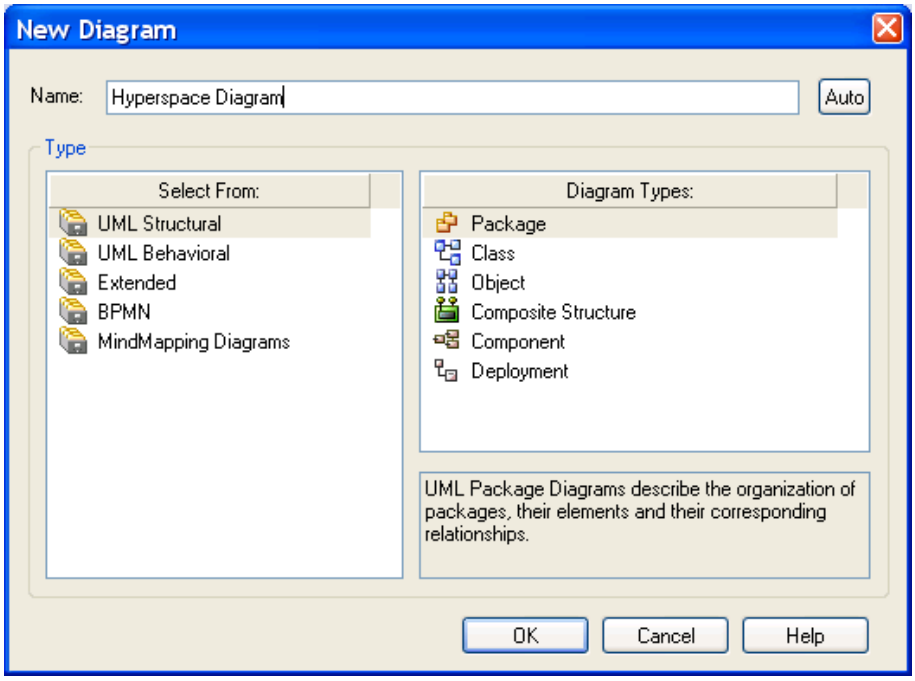

Abbildung 5.6: Dialog *New Diagram*

Der Typ des Hyperspace-Diagramms (*Package*) lässt bereits erahnen, dass die Überschrift *„Das Hyperspace-Diagramm“* nur die halbe Wahrheit darstellt. Tatsächlich verteilt sich die Modellierung des Hyperspaces auf mehrere Diagramme, deren Erzeugung im Anschluss erläutert wird.

Das *Hyperspace Diagram* enthält ausschließlich die Dimensionen. Deren Stereotyp ist als Paket modelliert, so dass zum Abschluss der Modellierung des Hyperspaces für jede Dimension ein eigenes Diagramm existieren wird.

Modellierung des Hyperspaces

Die Entwickler im Anwendungsbeispiel sehen für die Modellierung des Hyperspaces drei Dimensionen vor: *Classes* für die Klassen und *Use Cases* für die Anwendungsfälle als artefaktbasierende Belangdimensionen sowie *Functionalities* für die Funktionalitäten der Unternehmenssoftware als selektierende Belangdimension.

Um diese Dimensionen in das Hyperspace-Diagramm einzufügen, muss im Enterprise Architect zunächst das Fenster *Resources* geöffnet werden[1]. Dort befindet sich in *UML Profiles* das zuvor importierte *HyperspaceProfile*. Darin befinden sich die für die Modellierung des Hyperspaces benötigten Stereotypen. Abbildung 5.7 zeigt den Ausschnitt des Fensters *Resources* mit dem aufgeklappten UML-Profil.

Durch Ziehen von *Dimension* von *Resources* in das *Hyperspace Diagram* wird dort eine Dimension eingefügt. Das Dialogfenster *New Package Name* erscheint, in das

[1] *Hyperspace Diagram* wird bei der Erzeugung in EA automatisch als aktuell ausgewähltes Diagramm geöffnet.

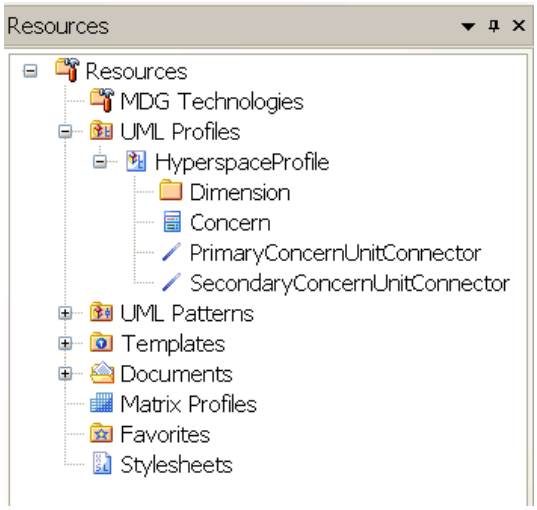

Abbildung 5.7: Ausschnitt des Fensters *Resources* mit vollständig eingeblendetem Hyperspace-Profil

der Name der zu erzeugenden Dimension - im Beispiel also `Classes`, `Use Cases` oder `Functionalities` - eingegeben wird. Bei Klick auf den Button *[OK]* wird das Dialogfenster geschlossen und die Dimension in das Diagramm eingefügt. Sie kann dort nach Bedarf verschoben werden. Abbildung 5.8 auf Seite 80 zeigt das fertige *Hyperspace Diagram* zum Anwendungsbeispiel.

Im Anschluss wird der Hyperspace für die einzelnen Dimensionen ausmodelliert. Dazu wird im Project Browser für jede Dimension zunächst ein Klassendiagramm erzeugt. Der Ablauf ist ähnlich dem der Erzeugung des Hyperspace-Diagramms:

Ein Rechtsklick auf die jeweilige Dimension öffnet das Kontextmenü, aus dem *Add* ausgewählt wird. Aus dem sich öffnenden Sub-Menü wird dann *Add Diagram* ausgewählt, so dass sich das Dialogfenster *New Diagram* öffnet. Hier wird dem Diagramm derselbe Name wie der jeweiligen Dimension zugewiesen; aus *Diagram Types* wird *Class* ausgewählt. Mit einem Klick auf den Button *[OK]* wird auch hier die Diagrammerstellung abgeschlossen und das Diagramm als aktuell ausgewähltes Diagramm geöffnet.

Einfügen von Belangen

Innerhalb eines Diagramms einer Dimension werden nun die verschiedenen Belange eingefügt. In der Dimension *Classes* gibt es so für jede Klasse einen Belang und in *Use Cases* für jeden Anwendungsfall. Von den Entwicklern im Anwendungsbeispiel werden für die Dimension *Functionalities* die Belange *Human Resources Administration* (Personalverwaltung), *Accounting* (Buchführung), *Customer Relationship Management* (Kundenverwaltung und -betreuung) und *Order Management* (Auftragsverwaltung) erzeugt.

Um die Belange in eine Dimension einfügen zu können, müssen das Fenster *Resources* und das Diagramm der jeweiligen Dimension geöffnet sein. Aus dem

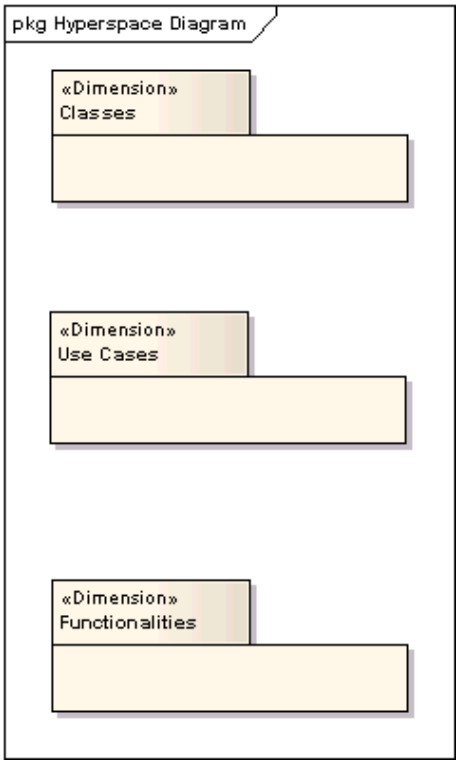

Abbildung 5.8: Hyperspace-Diagramm zum Anwendungsbeispiel

Hyperspace-Profil wird zur Erzeugung eines Belangs das Item *Concern* in das Diagramm gezogen. Das Dialogfenster zur Bestimmung der Eigenschaften des Elements öffnet sich. Hier wird unter *Name* der Name des Belangs eingegeben. Der Belang erscheint zeitgleich mit dem Öffnen des Dialogfensters im Diagramm und kann nach Bedarf beliebig verschoben werden.

Nach Erzeugung der Belange einer Dimension können diesen die Units zugeordnet werden. Dazu wird das Item einer Unit aus dem Project Browser in das Diagramm gezogen. Das Dialogfenster *Paste Element* erscheint. Unter *Paste Element into Diagram* muss die Option *as Simple Link* gewählt werden. Die Unit erscheint nach einem Klick auf *[OK]* im Diagramm.

Wenn eine Unit in das Diagramm eingefügt worden ist, muss diese noch über einen Konnektor mit dem entsprechenden Belang verbunden werden. Dazu wird wieder das Fenster *Resources* benötigt, wo aus dem Hyperspace-Profil ein *PrimaryConcernUnitConnector* oder ein *SecondaryConcernUnitConnector* in das Diagramm gezogen wird - je nachdem, ob es sich um eine artefaktbasierende oder um eine selektierende Belangdimension handelt. Der Konnektor muss auf eines der Diagrammelemente gezogen werden, welches mit einem anderen verbunden werden soll. Über das erscheinende Dialogfenster wird das Diagrammelement am anderen Ende des Konnektors bestimmt.

Sofern zwischen Units Verbindungen (z.B. Generalisierungen, Assoziationen) bestehen, werden diese in das Diagramm der Dimension übernommen. Dies kann die

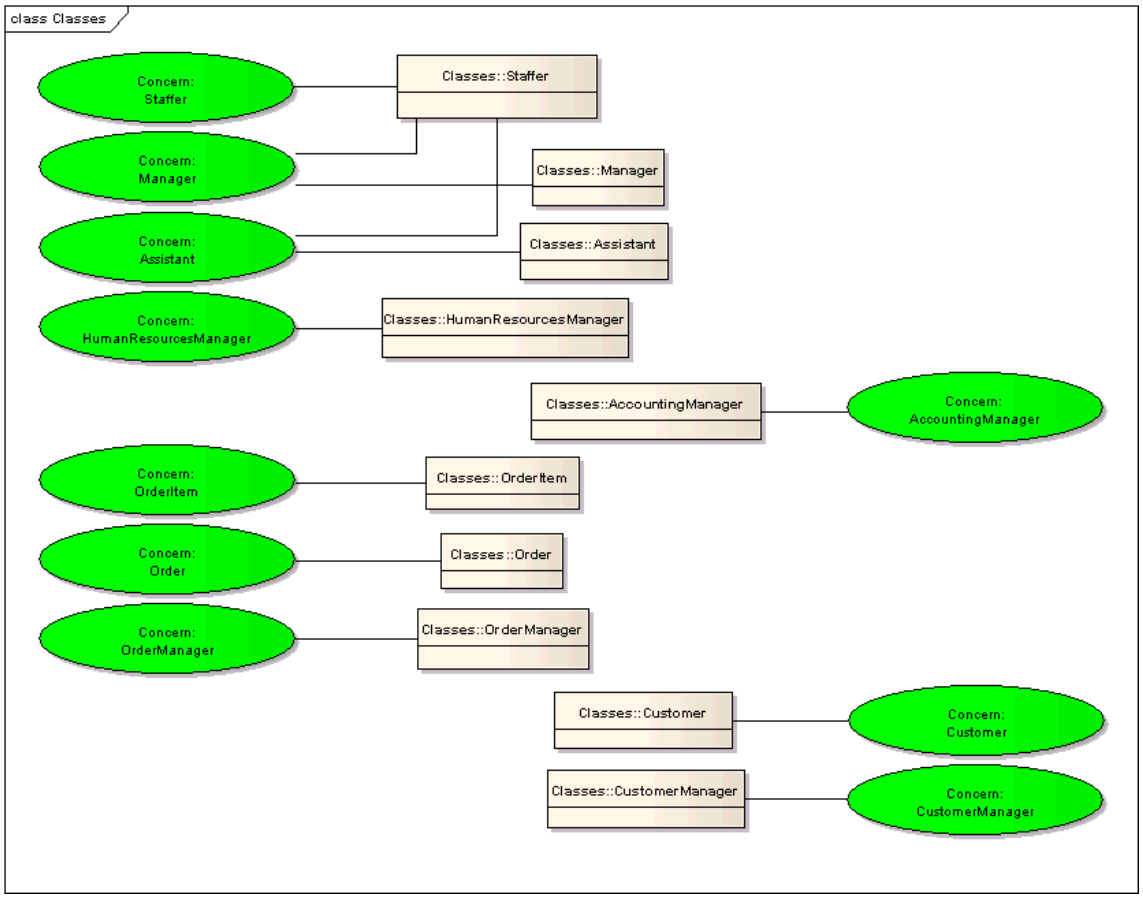

Abbildung 5.9: Beispieldiagramm zur Dimension *Classes*

Übersichtlichkeit stark einschränken. Ein Rechtsklick auf die Verbindung öffnet ein Kontextmenü, über das unter *Set Visibility* die Sichtbarkeit des Konnektors eingestellt werden kann. *Hide Connector* versteckt die Verbindung. Damit kann dem Problem der eingeschränkten Übersichtlichkeit entgegengewirkt werden.

Auch kann unter *Set Visibility* mit *Hide All Labels* eingestellt werden, dass Labels am Konnektor nicht angezeigt werden. Dies ist insofern interessant, dass der Stereotyp eines Konnektors zwischen Belang und Unit im Normalfall eingeblendet wird, was bei vielen Konnektoren die Übersichtlichkeit ebenfalls nicht gerade verbessert.

Ebenso ist es sinnvoll, die Attribute und Methoden eines Elements im Diagramm einer Dimension zu verstecken. Diese lassen sich in dem jeweiligen Diagramm einsehen, aus dem die Unit in das Diagramm der Dimension gezogen wurde. Darüber hinaus sind sie für die Modellierung des Hyperspaces nutzlos, solange es im Enterprise Architect nicht möglich ist, Belangen einzelne Methoden oder Attribute zuzuordnen.

Abbildung 5.9 zeigt das Diagramm der Dimension *Classes* aus dem Anwendungsbeispiel. Die Abbildungen 5.10 auf Seite 82 und 5.11 auf Seite 83 zeigen die Diagramme zu den Dimensionen *Use Cases* und *Functionalities* aus dem Anwendungsbeispiel.

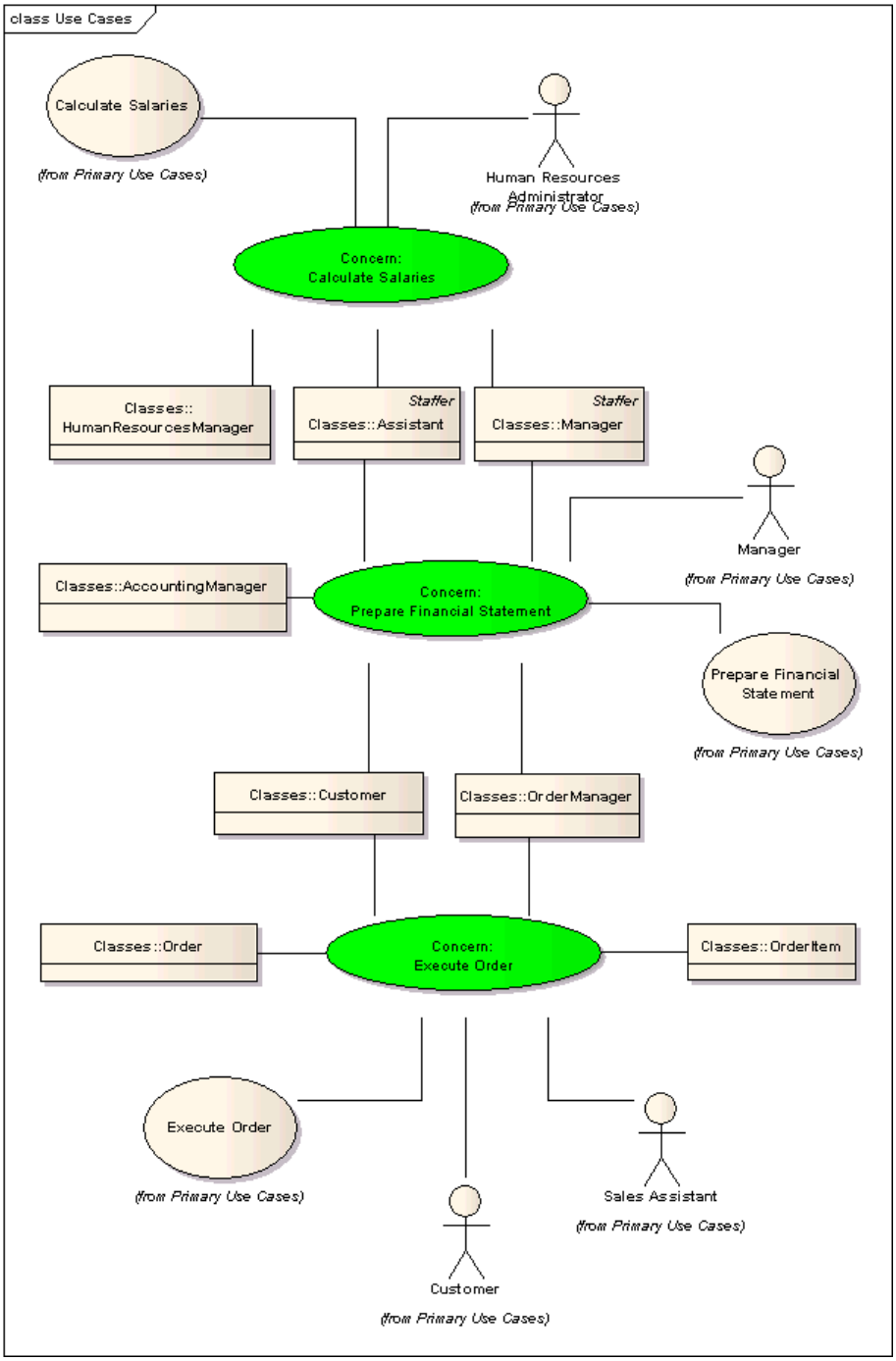

Abbildung 5.10: Beispieldiagramm zur Dimension *Use Cases*

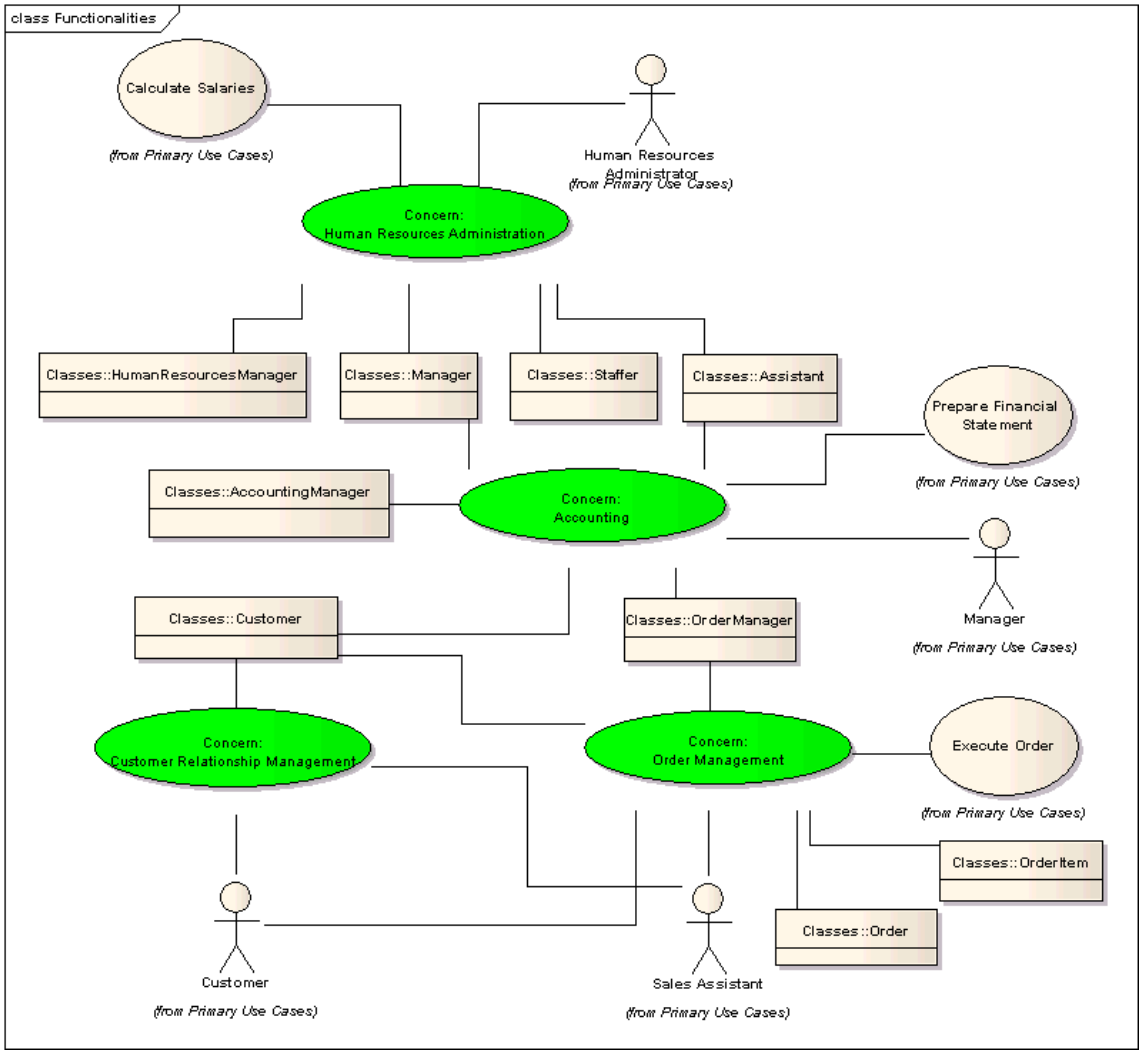

Abbildung 5.11: Beispieldiagramm zur Dimension *Functionalities*

5.3 Die Nutzung von Hynalysis

Ist die Modellierung des Hyperspaces, wie im Anwendungsbeispiel, zumindest teilweise abgeschlossen, so wird der Einsatz von Hynalysis interessant. Dieser Abschnitt führt das Anwendungsbeispiel fort und erläutert dabei die Installation von Hynalysis, das GUI und dessen Nutzung[2].

5.3.1 Die Installation

Hynalysis ist auf der beiliegenden CD-ROM in zwei Versionen verfügbar: Während der Ordner `Source` im Ordner `Hynalysis` den Quellcode von Hynalysis enthält, befindet sich im Ordner `Application` die `.jar`-Datei `hynalysis.jar`, welche den Einsatz von Hynalysis als GUI-Anwendung ermöglicht.

Um Hynalysis nutzen zu können, sind zunächst einige Vorbereitungen zu treffen. Hynalysis benötigt das API des Enterprise Architect, `eaapi.jar`. Zur Nutzung des EA-API wird wiederum die Bibliothek `SSJavaCOM.dll` benötigt. Diese muss sich innerhalb des Windows-Pfads, das EA-API im Java-Klassenpfad befinden. Aus lizenzrechtlichen Gründen befinden sich weder das EA-API noch die DLL auf der CD-ROM. Beide wurden in Version 7 des Enterprise Architect jedoch sowohl mit der Corporate Edition als auch mit der Testversion geliefert.

Neben diesen beiden Komponenten benötigt Hynalysis auch noch das JGraLab-Paket `jgralab.jar` sowie das Überführungstool in `transformer.jar`. Diese beiden Pakete befinden sich im Ordner `External JARs` auf der CD-ROM. Zudem wird das GNU-getopt-Paket `java-getopt-1.0.13.jar` benötigt, welches im Internet kostenfrei heruntergeladen werden kann[3]. Befinden sich auch diese drei JAR-Pakete im Java-Klassenpfad, kann Hynalysis gestartet werden.

5.3.2 Das Hauptfenster

Startet man Hynalysis als GUI, so öffnet sich das Hauptfenster, welches in Abbildung 5.12 dargestellt ist. Dessen Eigenschaften werden nachfolgend beschrieben.

Die *Menüleiste* am oberen Rand des Fensters dient ausschließlich dem sog. *„Look and Feel“*. Unter *File* findet sich lediglich die Option zum Schließen von Hynalysis.

Unterhalb der Menüleiste findet sich das Dateiauswahlfeld *Input repository* für das zu untersuchende EA-Projekt. Hier kann manuell der Pfad des Repositorys eingegeben werden. Alternativ kann durch einen Klick auf den Button *[Select]* ein Explorerfenster geöffnet werden, in dem das Repository ausgewählt werden kann.

Die beiden Optionen *Use existing TGraph* und *Keep TGraph resulting from the transformation* sind Alternativen. Erstgenannte Option kommt dann zum Einsatz, wenn das zu untersuchende EA-Repository bereits zuvor in einen TGraphen überführt wurde. So lässt sich die Zeit, welche für die Überführung benötigt wird, ein-

[2] Auf die Vorstellung der Nutzung von Hynalysis als Kommandozeilentool wird, da sich die gewählten Dialoge stark an die Interaktion im GUI anlehnen, bewusst verzichtet.

[3] z.B. auf `http://www.urbanophile.com/~arenn/hacking/download.html`

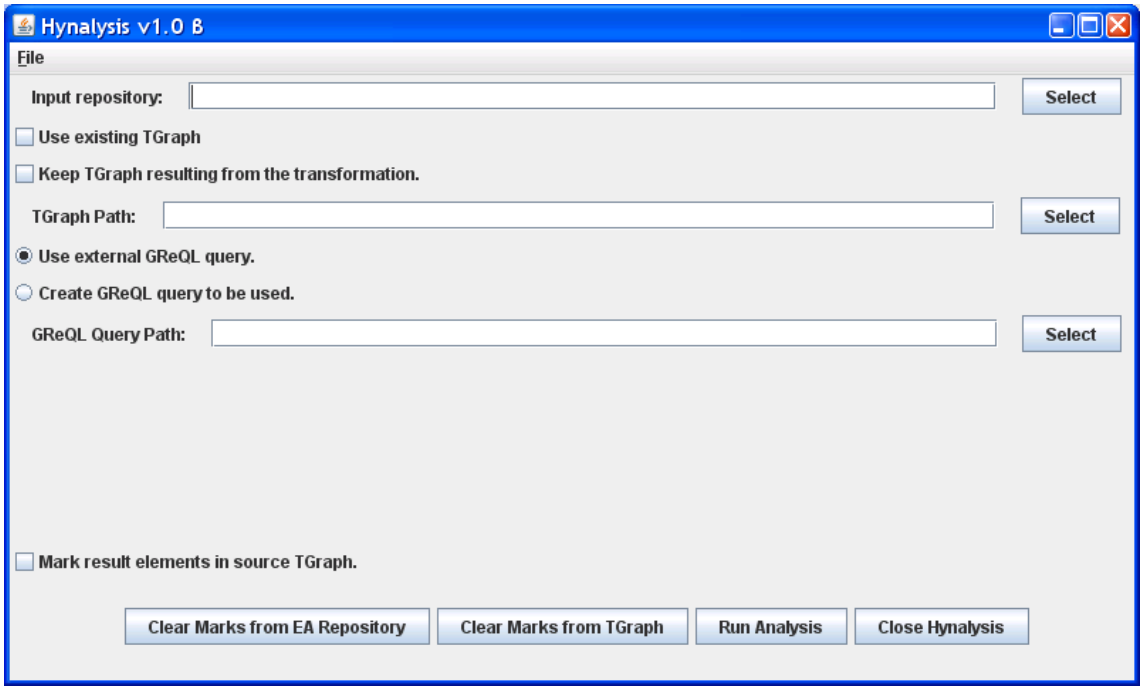

Abbildung 5.12: Das Hauptfenster von Hynalysis

sparen. Letztgenannte Option ist hingegen dann interessant, wenn das Ergebnis der Überführung gespeichert werden soll, um z.B. bei einer weiteren Anfrage darauf zurückgreifen zu können.

Wird keine der beiden Optionen gewählt, so erfolgt eine Überführung des EA-Repositorys in einen TGraphen, wobei das Ergebnis der Überführung nach der Untersuchung des Repositorys verworfen wird. In allen drei Fällen wird der Pfad des TGraphen im Feld *TGraph Path* eingegeben. Auch hier besteht die Möglichkeit zur Nutzung eines Explorerfensters über den Button *[Select]*.

Die beiden Radio Buttons *Use external GReQL Query* und *Create GReQL Query to be used* geben dem Nutzer die Wahlmöglichkeit, eine GReQL-Anfrage zur Untersuchung des Graphen aus einer Datei zu laden oder in einem Textfeld in Hynalysis eine GReQL-Anfrage zu erstellen. Bei Auswahl der ersten Option ist das Dateiauswahlfeld *GReQL Query Path* aktiv, in das der Pfad der externen `.greql`-Datei eingegeben werden muss. Über *[Select]* ist hier wieder die Möglichkeit der Nutzung eines Explorerfensters gegeben. Wählt der Nutzer die zweite Option, verschwindet das Dateiauswahlfeld und das Textfeld *GReQL Query* öffnet sich, in das dann manuell die Anfrage eingegeben wird.

Die Option *Mark result elements in source TGraph* bietet die Möglichkeit, die Ergebniselemente der Anfrageauswertung nicht nur im EA-Repository sondern auch im zugrundeliegenden TGraphen zu markieren. Ist die Option gewählt, wird für die Ergebniselemente das Flag `isResultElement` gesetzt.

Die Button-Leiste am unteren Rand des Fensters bietet schließlich vier Möglichkeiten: Klickt man auf *[Clear Marks from EA Repository]*, so werden alle eventuell

schon bestehenden Markierungen im gewählten EA-Repository gelöscht. Ein bestehendes Diagramm mit Ergebniselementen mitsamt der es enthaltenden Sicht muss hingegen manuell im EA gelöscht werden!

Ein Klick auf den Button *[Clear Marks from TGraph]* bewirkt das Löschen von eventuell bestehenden Markierungen im TGraphen. Hier wird für alle Elemente das Flag `isResultElement` auf `false` gesetzt.

Der Button *[Run Analysis]* startet die Anfrageauswertung. *[Close Hynalysis]* schließt Hynalysis.

Dialogfenster informieren den Nutzer über beendete Operationen und aufgetretene Fehler.

5.3.3 Die Nutzung

Im Anwendungsbeispiel soll nun eine GReQL-Anfrage auf dem Softwareprojekt ausgewertet werden. Diese wird zunächst in einem beliebigen Editor erzeugt und in Form einer `.greql`-Datei abgespeichert.

Die verwendete Beispielanfrage bestimmt all die Klassen und Anwendungsfälle, von denen die Klasse `Order` über eine ausgehende oder mehrere ausgehende, d.h. in Richtung `Order` verlaufende, Kanten des Typs `Connector` erreichbar ist. Sie ist wie folgt formuliert:

```
from v:V{EClass}, w:V{EClass,EUseCase}
with v.name="Order" and v(<--{Connector}+)w
report v,w
end
```

In der `from`-Klausel weist `v:VEClass` der Variable `v` die Menge der Knoten vom Typ `EClass` zu. Dies dient der Ermittlung der Klasse `Order`. Entsprechend wird `w` die Menge der Knoten vom Typ `EClass` oder `EUseCase` zugewiesen, da nur Klassen und Anwendungsfälle berücksichtigt werden sollen.

Die `with`-Klausel schränkt die Menge der zur Verfügung stehenden Knoten `v` auf diejenigen ein, deren Attribut `name` den Wert `Order` hat. Da im Klassendiagramm nur eine Klasse `Order` existiert und diese in den übrigen Diagrammen lediglich verlinkt wurde, ist hierfür die Identität gewährleistet, d.h. alle gefundenen Klassenknoten mit dem Namen `Order` repräsentieren dieselbe Klasse.

Der Pfadausdruck `v(<-Connector+)w` in der `with`-Klausel kann zerlegt werden[4]: `v` ist der Knoten der Klasse `Order`, `w` der jeweilige Knoten, von dem aus `v` erreichbar ist. `<-` legt fest, dass die zu berücksichtigenden Kanten in `v` einlaufende Kanten sind. `Connector` schränkt den Typ der Kanten auf `Connector` und davon abgeleitete Typen ein. Damit werden nur die Kanten berücksichtigt, die jeweils innerhalb eines Diagramms vorkommen. Das Kleene-Plus `+` schließlich bestimmt, dass `Order` von `w` über beliebig viele, aber mindestens eine Kante erreicht wird.

[4]Die runden Klammern `()` im Pfadausdruck sind hier nicht zwingend nötig, erhöhen aber die Übersichtlichkeit.

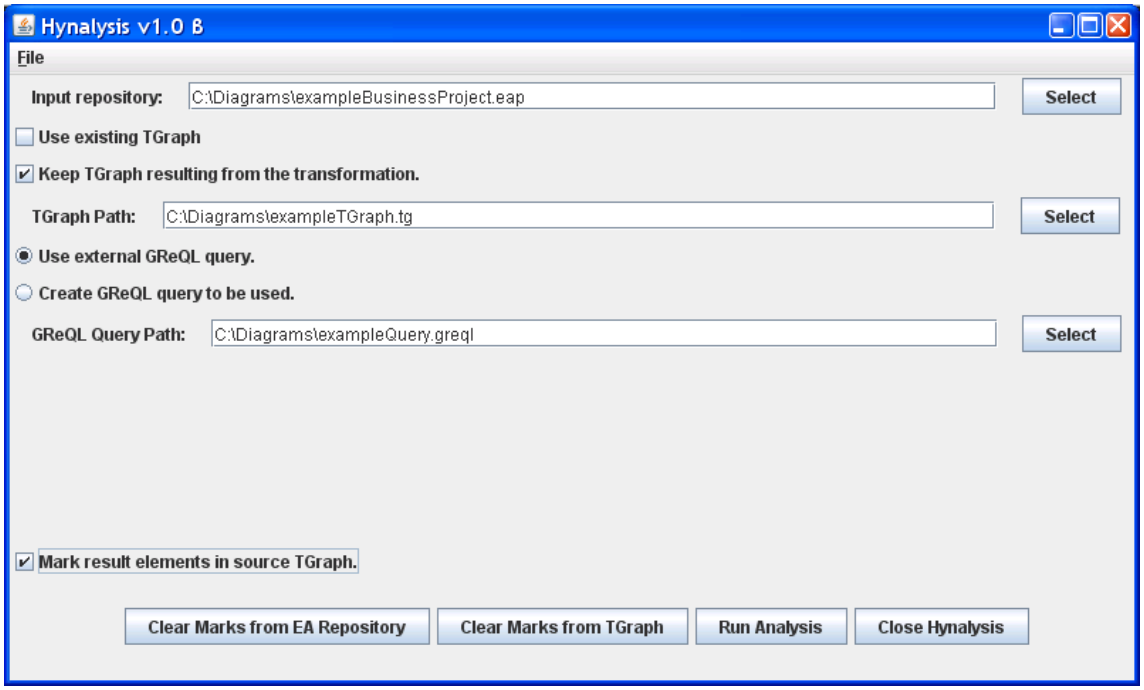

Abbildung 5.13: Hynalysis mit gesetzten Werten aus dem Anwendungsbeispiel

Die Anfrageauswertung mit Hynalysis

Damit Hynalysis diese Anfrage auswerten kann, muss das Repository, auf dem die Anfrage ausgewertet werden soll, zunächst im Enterprise Architect geschlossen werden. Ansonsten kann es zu Zugriffskonflikten auf das Repository kommen, weil Hynalysis nicht als Plug-In, sondern als externe Anwendung konzipiert ist. Auch der EA selbst sollte geschlossen werden, da Hynalysis die geöffnete Instanz von EA nicht nutzt und stattdessen im Laufe des Betriebs mehrfach Instanzen von EA öffnet und schließt.

Ist die Anfrage formuliert, Enterprise Architect geschlossen und Hynalysis geöffnet, bleibt nicht mehr viel zu tun: Im Feld *Input repository* wird der Pfad des EA-Repositorys gesetzt. Im Beispiel ist dies `C:\Diagrams\exampleBusinessProject.eap`, wie in der Abbildung 5.13 zu sehen ist.

Da die erste Anfrageauswertung bevorsteht und die Entwickler sicher noch weitere Anfragen stellen wollen, wird die Option *Keep TGraph resulting from the transformation* gewählt. Im Feld *TGraph Path* wird der Zielpfad für das Transformationsergebnis eingegeben (hier: `C:\Diagrams\exampleTGraph.tg`). Damit die Ergebniselemente der Anfrageauswertung auch im Tgraphen markiert werden, wählen die Entwickler gleich die Option *Mark result elements in source TGraph* mit aus.

Für die auszuwertende Anfrage wird die Option *Use external GReQL Query* ausgewählt, und der Pfad der `.greql`-Datei im Feld *GReQL Query Path* eingegeben. Im Beispiel ist dies `C:\Diagrams\exampleQuery.greql`.

Ein Klick auf den Button *[Run Analysis]* startet die Auswertung der GReQL-Anfrage. Der erfolgreiche Abschluss der Anfrageauswertung wird durch das Dialogfenster *Analysis completed* angezeigt. Je nach Umfang und Komplexität des EA-Repositorys und der Anfrage, und je nachdem, ob ein bereits existierender TGraph genutzt werden kann oder erst ein solcher erzeugt werden muss, kann die Zeitspanne vom Anklicken des Buttons bis zum Erscheinen des Dialogfensters deutlich variieren. Ein wenig Geduld kann hier also bisweilen nützlich sein.

Betrachtung des Anfrageergebnisses

Nach der Anfrageauswertung öffnet man das EA-Repository im Enterprise Architect. Im Project Browser findet man eine neue Sicht *Result View*, welche das Diagramm *Result Diagram* mit den Ergebniselementen enthält, die man nach Bedarf anordnen kann. Daneben sind die Ergebniselemente im gesamten Repository durch eine veränderte Rahmenfarbe und -dicke markiert.

Hynalysis bleibt nach dem Abschluss einer Anfrageauswertung geöffnet, so dass mit den gegebenen Parametern gleich eine weitere Anfrage ausgewertet werden kann. Um Zeit einzusparen empfiehlt es sich, in einer solchen Situation den zuvor erzeugten TGraphen zu nutzen, indem man dessen Pfad unverändert lässt und die Option *Use existing TGraph* wählt. Es sei jedoch noch einmal darauf hingewiesen, dass man zuvor im EA die *Result View* sowie das *Result Diagram* manuell löschen muss!

Abbildung 5.14 zeigt das aus der Beispielanfrage hervorgehende Diagramm der Ergebniselemente. Die enthaltenen Elemente sind bereits manuell angeordnet. Attribute und Methoden in Klassen sind ebenso ausgeblendet wie sämtliche Labels an Konnektoren.

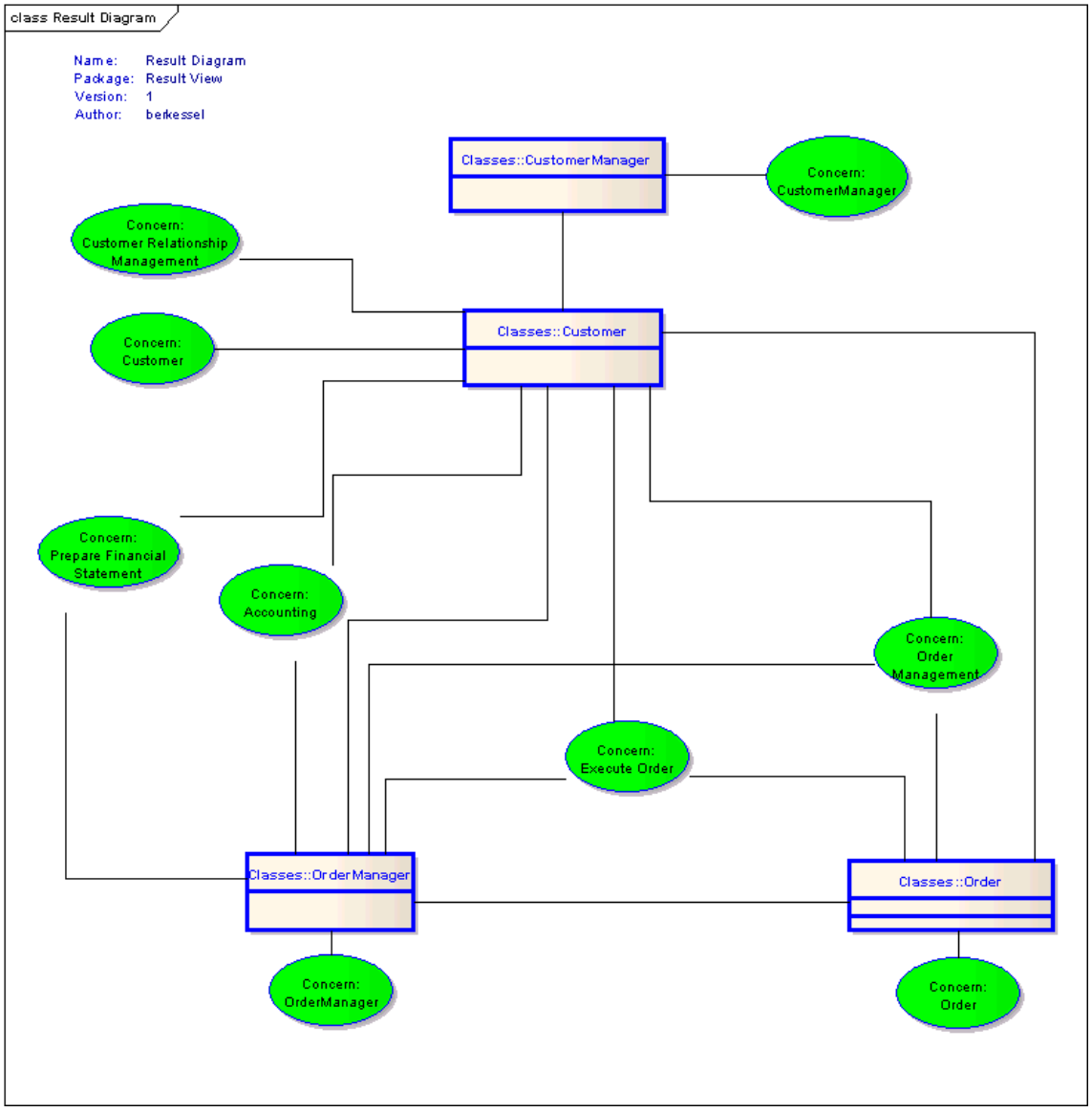

Abbildung 5.14: Ergebnisdiagramm zur Beispielanfrage

6 Ergebnisse

Zum Abschluss dieser Studienarbeit soll nun in 6.1 zunächst der umgesetzte Funktionsumfang von Hynalysis vorgestellt und im Hinblick auf die in Abschnitt 4.2 vorgestellten Anforderungen bewertet werden.

Anschließend wird in 6.2 ein Ausblick auf Chancen und Einsatzmöglichkeiten für sowie mögliche Änderungen an Hynalysis geboten, ehe in 6.3 eine kurze Zusammenfassung der Arbeit folgt.

6.1 Funktionsumfang und Bewertung

Zur Bewertung des Erfolgs oder auch Misserfolgs eines Projekts bietet es sich an, das tatsächlich Erreichte mit den zu Beginn gestellten Anforderungen zu vergleichen. Ebendies wird hier geboten: Nach einer Vorstellung des Funktionsumfangs von Hynalysis erfolgt die Bewertung anhand der Anforderungsliste aus 4.2.

6.1.1 Der Funktionsumfang von Hynalysis

Hynalysis stellt sowohl bei der Nutzung als Anwendung in der Kommandozeile als auch bei der Nutzung des GUI den gleichen Funktionsumfang zur Verfügung:

- Für ein Projekt im Enterprise Architect kann eine GReQL-Anfrage ausgewertet werden.
- Der Nutzer kann entscheiden, ob der zur Auswertung der GReQL-Anfrage erstellte TGraph gespeichert wird.
- Damit nicht jedesmal eine Überführung des EA-Repositorys in einen TGraphen erfolgen muss, besteht die Möglichkeit, die GReQL-Anfrage auf einem bereits aus dem Repository erstellten TGraphen auszuwerten.
- Die GReQL-Anfrage stammt im Standardfall aus einer externen Datei, welche der Nutzer zuvor in einem beliebigen Editor erstellt hat. Alternativ kann der Nutzer die Anfrage direkt in Hynalysis eingeben.
- Die Ergebniselemente werden im EA-Repository, für das die Anfrage ausgewertet wird, durch Änderung der Randfarbe und -dicke sowie eine Änderung der Schriftfarbe markiert. Im zugrundeliegenden TGraphen erfolgt - sofern vom Nutzer gewünscht - die Markierung durch Setzen des boolschen Attributs `isResultElement`.

- Zusätzlich werden nach der Anfrageauswertung im EA-Repository eine zusätzliche Sicht und innerhalb dieser Sicht ein Diagramm, welches die Ergebniselemente enthält, eingefügt.
- Der Nutzer kann bestehende Markierungen sowohl im EA-Repository als auch im TGraphen löschen. Markierungen werden standardmäßig vor Ausführung einer neuen Anfrage gelöscht. Die im Projekt erstellte Sicht mitsamt des erstellten Diagramms der Ergebniselemente muss allerdings vom Nutzer vor der Auswertung einer weiteren GReQL-Anfrage manuell in EA gelöscht werden.

Zudem wird durch den Import des auf der beiliegenden CD-ROM zu findenden UML-Profils `hyperspaceProfile.xml` in ein Projekt im Enterprise Architect das erweiterte Hyperspace-Modell in diesem Projekt in EA anwendbar gemacht. Belangdimensionen und Belange können somit in EA erstellt und Units von der Größe bis hinunter zu Klassen oder auch Anwendungsfällen können den Belangen zugeordnet werden. Eine Zuordnung kleinerer Units, z.B. von Methoden, zu Belangen ist nicht möglich.

6.1.2 Erfüllung der Anforderungen

Bevor die funktionalen Anforderungen untersucht werden, soll kurz auf die technischen und sonstigen Anforderungen sowie auf die Qualitätsanforderungen eingegangen werden: Hynalysis ist kompatibel zu *Enterprise Architect, Version 7.0*. Es basiert auf *Java 6* und nutzt *JGraLab*, *GReQL* sowie das *EA JGraLab - Transformertool*.

Die Programmierung von Hynalysis erfolgte in *Eclipse Ganymed*. Der Quellcode von Hynalysis ist vollständig kommentiert. Neben der Dokumentation der Architektur in 4.5 findet sich auf der beiliegenden CD-ROM eine *Javadoc*-Dokumentation. Die im Quellcode, in den Kommentaren, in der Javadoc-Dokumentation und für die Interaktion mit dem Nutzer verwendete Sprache ist englisch.

Hynalysis verfügt über eine Fehlerbehandlung, die im Falle von Störungen dem Nutzer entsprechende Meldungen anzeigt. Größere Störungen der Programmstabilität konnten im Testbetrieb nicht festgestellt werden.

Die technischen und sonstigen Anforderungen sind damit ebenso wie die Qualitätsanforderungen vollständig erfüllt.

Funktionale Anforderungen

Bevor einige Punkte im Hinblick auf die Erfüllung der funktionalen Anforderungen detaillierter erläutert werden, soll unter Berücksichtigung des in 6.1.1 vorgestellten Funktionsumfangs von Hynalysis festgestellt werden, welche Anforderungen erfüllt sind. Zugunsten der Übersichtlichkeit werden diese Feststellungen in der gleichen Reihenfolge getroffen, in der auch die entsprechenden Anforderungen in 4.2.1 aufgelistet sind.

1. Durch das UML-Profil `hyperspaceProfile.xml` wird das erweiterte Hyperspace-Modell von Lohmann im Enterprise Architect anwendbar gemacht.

2. Der Nutzer kann im Enterprise Architect Belangdimensionen und Belange erstellen.
3. Die Units der in Enterprise Architect erstellten Artefakte können den Belangen zugeordnet werden. Die Ausnahme bilden hier Methoden und Variablen innerhalb von Klassen, die ebenfalls Units darstellen. Klassen selbst können Belangen zugeordnet werden.
4. Die Software ermöglicht zwar die Bestimmung von Slices, jedoch können im Enterprise Architect *keine* dazu zu berücksichtigenden Diagrammelemente ausgewählt werden.
5. Hynalysis ermöglicht die Auswertung von GReQL-Anfragen.
6. Auszuwertende GReQL-Anfragen können aus externen `.greql`-Dateien importiert werden.
7. Die für die Bestimmung von Slices zu berücksichtigenden Knoten- und Kantentypen können über die GReQL-Anfrage ausgewählt werden.
8. Das Ergebnis der Auswertung der GReQL-Anfrage wird durch Einfärben der Ergebniselemente im Original-Repository im EA angezeigt.
9. Das Ergebnis der Auswertung der GReQL-Anfrage kann zwar nicht als eigene `.tg`-Datei exportiert werden. Jedoch besteht die Möglichkeit, den aus der Transformation des EA-Repositorys hervorgehenden TGraphen zu speichern und innerhalb dieses Graphen die Ergebniselemente durch Setzen des boolschen Attributs `isResultElement` markieren zu lassen.
10. Das Ergebniselemente der Auswertung der GReQL-Anfrage werden in einem zusätzlichen Diagramm im Original-Repository im Enterprise Architect angezeigt.
11. Das Ergebnis der Auswertung der GReQL-Anfrage kann *nicht* in Form eines zusätzlichen EA-Repositorys exportiert werden.
12. Die Funktionalität zur Bestimmung von Slices und zur Auswertung von GReQL-Anfragen kann *nicht* aus dem Enterprise Architect heraus angestoßen werden.

Die gestellten Anforderungen werden somit zwar größtenteils von Hynalysis und dem UML-Profil erfüllt. Jedoch sollen die Gründe für die Nichterfüllung mancher Anforderungen nicht unerwähnt bleiben:

Die Funktionalität zur Bestimmung von Slices und zur Auswertung von GReQL-Anfragen kann nicht aus dem EA heraus angestoßen werden, weil Hynalysis als eigenständige Anwendung und nicht als sog. Add-In konstruiert ist. Dies wiederum liegt im EA-API begründet, welches für Version 7 des Enterprise Architect lediglich als Beta-Version vorliegt. In der dem API beigefügten Readme-Datei heißt es: *„[...]You cannot currently use thie API to write plug-ins for EA. It is only suitable for accessing the automation server API. Plug-in capability is currently being planned.“*[1] Die Erstellung von Plug-Ins im Rahmen des Enterprise Architect Add-In Models ist also - zumindest für Version 7.0 - nicht möglich. Da aber nur die Funk-

[1] Da es sich um ein Zitat handelt, ist auch das fehlerhafte Wort *thie* übernommen worden. Korrekt muss es *this* heißen.

tionalität von Add-Ins aus dem EA heraus angestoßen werden kann, muss Hynalysis als eigenständige Anwendung betrieben werden.

Damit lässt sich auch erklären, warum es nicht möglich ist, innerhalb des EA Diagrammelemente auszuwählen, die für die Bestimmung von Slices zu berücksichtigen sind: Die Abfrage von selektierten Elementen und Konnektoren mit den Methoden `GetSelectedElements()` und `GetSelectedConnector()` der Klasse `Diagram`, wie in 4.4.2 erläutert, ist nur mit einem Add-In nutzbar.[2] Jedoch ergibt sich hier, ebenso wie für die Auswahl von für die Bestimmung von Slices zu berücksichtigenden Knoten- und Kantentypen, die Möglichkeit, über die GReQL-Anfrage gezielt einzelne Diagrammelemente auszuwählen.

Ein Export des Ergebnisses der Auswertung der GReQL-Anfrage in einem zusätzlichen EA-Repository erschien im Hinblick auf die ohnehin erfolgende Markierung der Ergebniselemente im Original-Repository und das Hinzufügen eines Diagramms, welches die Ergebniselemente enthält, nicht mehr sinnvoll und wurde dementsprechend nicht umgesetzt. Ähnliche Überlegungen führten dazu, die Ergebniselemente nicht in einer eigenen `.tg`-Datei zu speichern, sondern mit Hilfe des Attributs `isResultElement` im originalen TGraphen zu markieren.

6.1.3 Bewertung

Hynalysis und das UML-Profil `hyperspaceProfile.xml` erfüllen gemeinsam fast alle gestellten Anforderungen. Die nicht erfüllten, funktionalen Anforderungen sind allesamt nicht erfüllbar oder im Nachhinein als nicht sinnvoll zu betrachten. Zumindest für die Auswahl von Diagrammelementen zur Bestimmung von Slices bietet sich aber der Umweg über eine entsprechende Formulierung der GReQL-Anfrage.

6.2 Ausblick

Obwohl die an Hynalysis gestellten Anforderungen größtenteils erfüllt sind, verbleiben einige Ansätze zur Verbesserung des Programms. Diese finden sich in erster Linie in den Erläuterungen zur Nichterfüllung mancher Anforderungen und sollen hier dargestellt werden. Im Anschluss werden die Perspektiven von Hynalysis im praktischen Einsatz und die sich daraus ergebenden Chancen und Möglichkeiten erläutert.

6.2.1 Anregungen

Mögliche Weiterentwicklungen von Hynalysis sollten die *Nutzung als Add-In für Enterprise Architect* ermöglichen. Günstigstenfalls wird dies bereits durch eine neuere Version von Enterprise Architect und damit auch ein überarbeitetes API für Java ermöglicht.[3]

[2] Es stellt sich in diesem Zusammenhang die Frage, warum das EA-API diese Methoden überhaupt vorsieht, wenn sie noch nicht brauchbar sind.

[3] Enterprise Architect ist zur Zeit in Version *7.5* verfügbar.

Eine mögliche Alternative ist derzeit lediglich in der Entwicklung einer *ActiveX COM*-Komponente, z.B. mit Visual Basic, zu sehen. Mit Hilfe einer *Java-ActiveX-Brücke* wie beispielsweise *EZ JCom*[4] könnte eine Verbindung zwischen der Komponente und Hynalysis erzeugt werden, so dass Hynalysis über eine „Umleitung“ der Zugriff auf den Enterprise Architect ermöglicht wäre. Jedoch besteht hier ein Risiko von Mehrfachzugriffen auf das EA-Repository. Ob die Umsetzung eines solchen Ansatzes möglich ist, bleibt fraglich.

Sobald der Zugriff auf das geöffnete Repository und die vollständige Interaktion zwischen Hynalysis und dem Enterprise Architect möglich sind, sollte die Funktionalität zur Auswahl von Diagrammelementen im EA und die darauf basierende Erzeugung eines Slices implementiert werden. Dazu ist eine Anpassung im Überführungstool aus [Bra07] notwendig, so dass das im Schema bereits vorgesehene Attribut `isSelected` gesetzt wird (s. 4.4.2). Auf dieses lässt sich dann im Rahmen einer GReQL-Anfrage für die Slice-Bestimmung zugreifen. Daneben sollte auch der Aufruf der Funktionalitäten von Hynalysis aus dem EA heraus umgesetzt werden.

Denkbar ist zudem, die Architektur von Hynalysis mit der des Überführungstools zu „verheiraten“. Dies liegt zum einen in Komfortaspekten begründet, da ein Nutzer sowohl die Überführungs- als auch die Analysefunktionalitäten in einem Tool finden würde. Zudem liegt die Verwandtschaft der beiden Programme auf der Hand. Schließlich sollte sich weniger häufiges Öffnen und Schließen des EA-Repositories auch in einer deutlichen Steigerung der Performanz von Hynalysis niederschlagen.

Als letzte im Hinblick auf die möglichst vollständige Umsetzung des erweiterten Hyperspace-Modells wünschenswerte Weiterentwicklung von Hynalysis bleibt noch die Möglichkeit der Zuordnung *aller* Units zu Belangen zu nennen. Könnten z.B. auch Variablen und Methoden aus Klassen in EA als eigenständige Units gehandhabt werden, wäre eine noch bessere Trennung der Belange möglich.

6.2.2 Zukunftschancen

Mit Hynalysis in Verbindung mit dem für Enterprise Architect erstellten UML-Profil bietet sich bereits jetzt die Möglichkeit, das erweiterte Hyperspace-Modell im Softwareentwicklungsprozess mit EA anzuwenden. Damit lässt sich eine deutliche Verbesserung der Trennung der Belange erzielen, was zu einer übersichtlicheren, besser strukturierten, leichter erweiterbaren und einfacher wartbaren Software führen kann.

Sobald es möglich ist, ausnahmslos alle Units in EA Belangen zuzuordnen, ist das erweiterte Hyperspace-Modell vollständig umgesetzt. Damit lassen sich die in 2.3 vorgestellten *Ilities* - *comprehensibility*, *limited impact of change*, *traceability* und *evolvability* - im Softwareprozess nahezu optimal verwirklichen.

Die Auswertung von GReQL-Anfragen bietet darüber hinaus die Möglichkeit, gezielte Anfragen auf einem Softwareprojekt auszuführen und so detaillierte Informationen über nicht sofort sichbare Zusammenhänge im Projekt zu erhalten. Dies fördert auch die Erstellung von Program Slices, wie sie in 2.5.1 vorgestellt werden, und ermöglicht so ein effizienteres Testen und Warten einer Software.

[4]Internet: `http://javaactivex.com/`

Mit einer durch die in 6.2.1 vorgeschlagenen Verbesserungen zu erreichenden höheren Effizienz von Hynalysis (und ggf. des Überführungstools) dürften sich - zumindest für die Softwareentwicklung mit Enterprise Architect - erhebliche Effizienzsteigerungen im gesamten Software-Lebenszyklus erreichen lassen, was insbesondere für Unternehmen von Interesse ist. Diese Effizienzsteigerungen lassen sich zwar nicht in Zahlen messen - immerhin fehlen stets die Vergleichswerte zu einem auf „herkömmlichem“ Wege entwickelten Softwaresystem mit identischem Funktionsumfang. Jedoch sollten die geringere Wahrscheinlichkeit von „Bananensoftware“ und die fast schon zu versprechenden Kosteneinsparungen Grund genug sein, dem hier vorgestellten Konzept mittelfristig auch im Praxiseinsatz eine Chance zu geben.

6.3 Fazit

Zum Ende dieser Studienarbeit soll ein abschließender Überlick über die behandelten Themen gegeben werden. Die wesentlichen Punkte werden zusammengefasst und mit den Perspektiven für Hynalysis in Zusammenhang gestellt.

6.3.1 Theoretische Hintergründe

In der Softwareentwicklung spielt das Prinzip der *Trennung der Belange*, der Zerlegung eines Softwaresystems in einzelne Module, eine sehr wichtige Rolle. Zur Trennung werden mehrere Dekompositionstechniken angewendet, mit denen das System jeweils entlang einer Sicht auf das System, einer *Dimension*, zerlegt wird. Da meist eine der angewendeten Dekompositionstechniken dominant ist, finden manche Belange sich in verschiedenen Dimensionen wieder, so dass keine saubere Trennung der Belange erreicht werden kann.

Um diesem Problem zu begegnen, fordern Harold Ossher und Peri Tarr eine simultane mehrdimensionale Trennung der Belange, die sie in ihrem Hyperspace-Modell umzusetzen versuchen. Der *Hyperspace* ist dabei ein mehrdimensionaler Raum, dessen Achsen verschiedene Dimensionen darstellen, an denen die Belange der jeweiligen Dimension abgetragen werden. Innerhalb des Belangraums werden die *Units* - die Grundelemente von Artefakten, aus denen sich eine Software zusammensetzt - eindeutig genau einem Belang jeder Dimension zugeordnet. Ist für eine Dimension kein Belang passend, so wird eine Unit für diese Dimension dem sog. Nullbelang zugeordnet.

Daniel Lohmann stellt im Konzept von Ossher und Parr einige Schwächen fest, die er mit seinem *erweiterten Hyperspace-Modell* zu umgehen versucht. So müssen Units in den frühen Phasen der Softwareentwicklung häufig mehreren Belangen einer Dimension zugeordnet werden - man denke nur an Akteure in Use Cases. Lohmann gibt deshalb für sein Modell die Raumeigenschaft auf und führt eine Unterscheidung zweier verschiedener Arten von Dimensionen, der artefaktbasierenden und der selektierenden Belangdimensionen, ein. Während erstgenannte direkt aus einer Artefaktsprache hervorgehen, z.B. Klassen aus Klassendiagrammen oder Use Cases aus

Anwendungsfalldiagrammen, bieten letztgenannte lediglich eine zusätzliche Sicht auf das Softwaresystem und beleuchten z.B. die Funktionalitäten des Systems.

6.3.2 Hynalysis

Während für die Umsetzung von Lohmanns Modell im Enterprise Architect eine zusätzliche Sicht sowie einige Anpassungen in einem UML-Profil genügen, sind für die Umsetzung der Bestimmung von *Slices* weitergehende Maßnahmen erforderlich, die in der im Rahmen der Studienarbeit entwickelten Software Hynalysis münden. Als Slice werden dabei all die Belange und Units definiert, die direkt oder transitiv mit einem Slicing-Kriterium, welches selbst wieder ein Belang oder eine Unit sein kann, verbunden sind.

Zur Bestimmung des Slices eines *EA-Repositorys* ist die Auswertung einer speziellen *GReQL-Anfrage* nötig. Da Anfragen mit dieser Graphenanfragesprache nur auf *TGraphen* funktionieren, muss zunächst eine Überführung des Repositorys in einen TGraphen erfolgen. Diese Funktionalität wird von Elmar Brauchs *Überführungstool* bereitgestellt, welches seinerseits zu diesem Zweck auf das mit dem Enterprise Architect gelieferte API sowie das API *JGraLab* zurückgreift. Erstgenanntes API ermöglicht die Manipulation von EA-Repositorys, letztgenanntes die Manipulation von TGraphen mit Java.

Die erwünschte Markierung von Anfrageergebnissen sowohl im EA-Repository als auch im TGraphen macht Änderungen am bereits im Überführungstool bestehenden Schema für EA-Repositorys nötig: Ein zusätzliches Attribut `isResultElement` macht die Markierung im TGraphen möglich; gemeinsam mit einem Attribut für die ein Element im EA-Repository eindeutig identifizierende *GUID* wird auch die Markierung in EA ermöglicht. Mit diesen zusätzlichen Attributen einhergehend sind Anpassungen am Überführungstool, damit die neuen Attribute im Rahmen der Überführung von EA in einen TGraph mit den entsprechenden Werten belegt werden können.

Die beiden vom Überführungstool genutzten APIs finden auch in Hynalysis selbst Anwendung: JGraLab stellt die Funktionalitäten zur Auswertung einer GReQL-Anfrage und zur Weiterverarbeitung der Anfrageergebnisse bereit. Darüber hinaus setzt und liest Hynalysis damit Attribute im TGraphen. Mit Hilfe des EA-APIs werden die Ergebniselemente der Anfrageauswertung im EA-Repository identifiziert und markiert.

6.3.3 Möglichkeiten und Chancen

In seiner derzeitigen Version bietet Hynalysis dem Nutzer die Möglichkeit, beliebige GReQL-Anfragen auf einem EA-Repository auszuwerten. Die Ergebniselemente der Anfrageauswertung werden im Enterprise Architect farblich und im TGraphen durch Setzen eines Flags markiert. Daneben können bestehende Markierungen gelöscht werden.

Obwohl Hynalysis noch als separates Tool und nicht als Plug-In im Enterprise Architect nutzbar ist, und obwohl noch Verbesserungspotential sowohl für Hynalysis selbst als auch für die Interaktion mit EA und dem Überführungstool besteht, sind im Rahmen des Softwareentwicklungsprozesses die durch den Einsatz von Hynalysis in Kombination mit Enterprise Architect und dem dafür erstellten UML-Profil erzielbaren Vorteile beachtlich: Es kann nicht nur eine wesentlich exaktere Trennung der Belange erzielt werden. Die Auswertung von GReQL-Anfragen ermöglicht zudem, gezielte Anfragen an ein Softwareentwicklungsprojekt zu stellen und so nicht sofort sichtbare Zusammenhänge im Projekt zu erkennen.

Daraus resultiert mit hoher Wahrscheinlichkeit eine insgesamt bessere Software: Eine schlankere, besser strukturierte Architektur erhöht nicht nur die Übersichtlichkeit für die Entwickler. Sie gewährleistet zudem eine bessere Nachvollziehbarkeit bei Änderungen, hält deren Einfluss auf das Gesamtsystem so gering wie möglich und verbessert die Erweiterbarkeit und Wartbarkeit der Software.

Der Einsatz von Hynalysis und Enterprise Architect mit der konsequenten Anwendung des erweiterten Hyperspace-Modells würde die Effizienz im Software-Lebenszyklus deutlich erhöhen. Damit böten sich auch und insbesondere außerhalb des universitären Rahmens in der Softwareindustrie erhebliche Potentiale zur Kosteneinsparung, die eine ganze Branche revolutionieren könnten!

Literaturverzeichnis

[BERS08] Bildhauer, Daniel ; Ebert, Jürgen ; Riediger, Volker ; Schwarz, Hannes: Using the TGraph Approach for Model Fact Repositories. In: *Proceedings of the Second International Workshop MoRSe 2008: Model Reuse Strategies – Can requirements drive reuse of software models?*, 2008, S. 9–18

[BHR+09] Bildhauer, Daniel ; Horn, Tassilo ; Riediger, Volker ; Schwarz, Hannes ; Strauss, Sascha: *grUML - A UML based modelling language for TGraphs.* Koblenz, 2009

[Bil06] Bildhauer, Daniel: *Ein Interpreter für GReQL 2. Entwurf und prototypische Implementation.* Koblenz, Universität Koblenz-Landau, Diplomarbeit, 2006

[Bra07] Brauch, Elmar: Überführung von UML-Modellen aus dem Enterprise Architect nach JGraLab / Universität Koblenz-Landau. Koblenz, 2007. – Studienarbeit

[BS07] Bildhauer, Daniel ; Schwarz, Hannes: JGraLab Citymap Tutorial / Universität Koblenz-Landau. Koblenz, 2007. – Forschungsbericht

[Cro99] Crowther, Jonathan (Hrsg.): *Oxford Advanced Learner's Dictionary.* 5th Edition. Oxford : Cornelsen & Oxford, 1999

[GHJV95] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns. Elements of Reusable Object-Oriented Software.* Reading, MA, USA : Addison-Wesley, 1995

[Hil99] Hilliard, Rich: *Aspects, Concerns, Subjects, Views, ...* URL: `http://www.cs.ubc.ca/~murphy/multid-workshop-oopsla99/position-papers/ws08-hilliard.pdf`, 1999. – Positionspapier, Einsendung zum *OOPSLA'99 Workshop on Multi-Dimensional Separation of Concerns in Object-Oriented Systems*

[KW99] Kullbach, Bernt ; Winter, Andreas: Querying as an Enabling Technology in Software Reengineering. In: Verhoef, C. (Hrsg.) ; Nesi, P. (Hrsg.): *Proceedings of the 3rd Euromicro Conference on Software Maintenance & Reengineering.* Los Alamitos : IEEE Computer Society, 1999, 42-50

[LE03] Lohmann, Daniel ; Ebert, Jürgen: A Generalization of the Hyperspace Approach Using Meta-Models / Universität Koblenz-Landau. Koblenz, 2003. – Forschungsbericht

[Loh02] Lohmann, Daniel: *Multidimensionales Trennen der Belange im Softwareentwurf.* Koblenz, Universität Koblenz-Landau, Diplomarbeit, 2002

[Mar06] Marchewka, Katrin: *Entwurf und Definition der Graphanfragesprache GReQL 2.* Koblenz, Universität Koblenz-Landau, Diplomarbeit, 2006

[OT99] Ossher, Harold ; Tarr, Peri: Multi-Dimensional Separation of Concerns in Hyperspace / IBM Research Division. IBM T.J. Watson Research Center, Yorktown Heights, NY, USA, 1999 (RC 21452(96717)16APR99). – Forschungsbericht

[OT00] Ossher, Harold ; Tarr, Peri: Multi-dimensional separation of concerns and the hyperspace approach. In: *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*, Kluwer, 2000

[Sch06] Schwarz, Hannes: *Entwicklung eines Dienstmodells für das Konzept des Program Slicing.* Koblenz, Universität Koblenz-Landau, Diplomarbeit, 2006

[Spa07] Sparx Systems Pty Ltd: *Company Profile: Sparx Systems.* Creswick, VIC, Australien : URL: `http://www.sparxsystems.com.au/downloads/pdf/CompanyProfile.pdf`, 2007

[Spa08] Sparx Systems Pty Ltd: *Enterprise Architect User Guide.* Creswick, VIC, Australien, 2008

[Wei81] Weiser, Mark: Program slicing. In: *ICSE '81: Proceedings of the 5th international conference on Software engineering.* Piscataway, NJ, USA : IEEE Press, 1981. – ISBN 0–89791–146–6, S. 439–449

[Zha97] Zhao, Jianjun: Using Dependence Analysis to Support Software Architecture Understanding. In: *New Technologies on Computer Software* (1997), S. 135–142