UNIVERSITÄT KOBLENZ · LANDAU

Fachbereich 4 - Informatik
Institut für Softwaretechnik

Überführung von UML-Modellen aus dem Enterprise Architect nach JGraLab

Studienarbeit
im Studiengang Informatik

vorgelegt von:
Elmar Brauch
Mat-Nr.: 203110017
am
27.03.2007

Betreuer:
Prof. Dr. Jürgen Ebert, Daniel Bildhauer, Institut für Softwaretechnik

Koblenz, im März 2007

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden. Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.

. . . . . .

(Ort, Datum) (Unterschrift)

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Ziel der Studienarbeit

Die Studienarbeit soll dem Java Graphenlabor (JGraLab) über das application programming interface (API) des Enterprise Architects einen Zugriff auf bzw. einen Import dessen Repository-Inhalt ermöglichen. Bei der Gelegenheit soll das API des Enterprise Architect analysiert werden.

Die Umgebung der Studienarbeit besteht also aus dem Enterprise Architect (EA) und JGraLab. Ein Tool, das für den Zugriff bzw. Import verantwortlich ist, soll in der Studienarbeit entwickelt werden.

Der Enterprise Architect [Spa06a] ist ein Modellierungswerkzeug, das Editoren für verschiedene visuelle Dokumente, insbesondere UML-Modelle, bietet. Der Enterprise Architect wird von Sparx Systems entwickelt und soll in Zukunft an Projekten der Universität Koblenz verwendet werden.
JGraLab [Kah06] ist ein universitätseigenes Produkt der Arbeitsgruppe Ebert. JGraLab wird zur Manipulation, zur algorithmischen Auswertung und zur Untersuchung von Graphen, die zu bestimmten Schemata passen, verwendet.

JGraLab bietet dem Anwender verschiedene Möglichkeiten Graphen zu verarbeiten. Jedoch keine Möglichkeit zur grafischen Modellierung von UML-Diagrammen. Dazu verwendet man ein externes UML-Modellierungswerkzeug wie den Enterprise Architect. JGraLab bietet dem Benutzer aber effiziente Möglichkeiten zur algorithmischen Auswertung und Untersuchung von Graphen. Dies kann der Enterprise Architect nicht leisten.
Da Modelle auch als Graphen interpretieren werden können, würde sich eine Kooperation beider Produkte anbieten, so dass die Vorteile von beiden genutzt werden können. Diese Kooperation könnte durch ein weiteres Tool ermöglicht werden. Dieses sollte die Modelle vom Enterprise Architect auslesen und sie in Graphen für JGraLab überführen. Nachdem JGraLab die Graphen verarbeitet hat, sollte das Tool diese auch wieder zurück in den Enterprise Architect schreiben können. Dieses Tool soll in dieser Studienarbeit entwickelt werden.

Um die Modelle des Enterprise Architects verwenden zu können, benötigt das Tool einen Zugriff auf das Repository des Enterprise Architects. Dieser Repository-Zugriff soll über das

API des Enterprise Architects erfolgen. Dazu muss das API zuerst analysiert und verstanden werden. Das Tool liest also den Inhalt des Repositorys aus. Das sind zum einen die Modelle und zum anderen die Metamodelle, die die Syntax der Modelle beschreiben. Dann bildet das Tool die Modelle auf TGraphen [Kah06] ab. Das ist die Graphenklasse, die in JGraLab verwendet wird. Diese Abbildung geschieht mit Hilfe der Metamodelle und der Schemata, die die Syntax von TGraphen beschreiben, analog zu Metamodellen. Nach verschiedenen Ergänzungen und Manipulationen, die eventuell zur Abbildung nötig sind, sollten die Modelle als TGraphen gespeichert werden, so dass sie anschließend mit den Methoden von JGraLab verarbeitet werden können. Die Überführung von Modellen in TGraphen ist der Kern der Studienarbeit.

Das Tool sollte anschließend auch noch den Rückweg ermöglichen, also das Umwandeln von TGraphen in Modelle, die dann zurück in das EA-Repository geschrieben werden. Durch das zu entwickelnde Tool soll also eine möglichst nahtlose Bearbeitung von Modellen mit Hilfe des Enterprise Architects und JGraLab ermöglicht werden. Das Zurückschreiben von Modellen in den Enterprise Architect ist allerdings nur ein optionaler Punkt der Studienarbeit.

Da der Enterprise Architect ein gekauftes Fremdprodukt ist, wird sich das zu entwickelnde Tool an JGraLab orientieren und wird daher, wie JGraLab, in Java entwickelt.

2 Enterprise Architect

2.1 Einführung

Für Programmierer stellt der Enterprise Architect ein API bereit. Dieses kann benutzt werden, um Projekte und deren Inhalt innerhalb des Enterprise Architect Repositorys zu bearbeiten. Bearbeiten heißt, dass zum Beispiel Projekte, Diagramme oder Klassen erzeugt und gelöscht werden können, aber auch, dass diese Elemente oder sonstige neu verknüpft, umbenannt oder erweitert werden können.

Das API bietet dem Programmierer also einen Repository-Zugriff, wobei die Objekte innerhalb des Repositorys die verschiedenen Elemente repräsentieren, die der Enterprise Architect verwendet, um seine UML-Diagramme aufzubauen, zu gliedern und zu unterteilen. Im Folgenden sollen die wichtigsten Funktionen des API erklärt werden. Wie man mit dem API arbeitet, wird anhand eines Beispiels gezeigt.

2.2 API-Klassen des Enterprise Architect

In diesem Abschnitt sollen die wichtigsten Klassen des EA-API, deren Beziehungen zueinander, ihre Attribute und ihre Funktionen vorgestellt werden. Momentan bietet das API des Enterprise Architect dem Programmierer zirka 65 verschiedene Klassen. Mit deren Hilfe kann der Programmierer die Projekte des Enterprise Architects verarbeiten. Um einen ersten Überblick zu gewinnen, ist es nicht sinnvoll ein vollständiges Klassendiagramm zu betrachten, daher zeigt Abb. 2.1 ein Klassendiagramm, das sich auf die wichtigsten Bestandteile beschränkt.

Repository. Die Klasse *Repository* beschreibt das Objekt, das alle weiteren Instanzen enthält. Es ist im Prinzip ein Hauptcontainer, der alle weiteren Instanzen der Klassen, die in Abb. 2.1 gezeigt sind, enthält. Eine *Repository* Instanz dient also als Container für EA-Projekte.
Der Enterprise Architect speichert seine Projekte in *eap*-Dateien. Um den Inhalt einer solchen *eap*-Datei in ein Repository zu laden, verwendet man die Methode *OpenFile* und über-

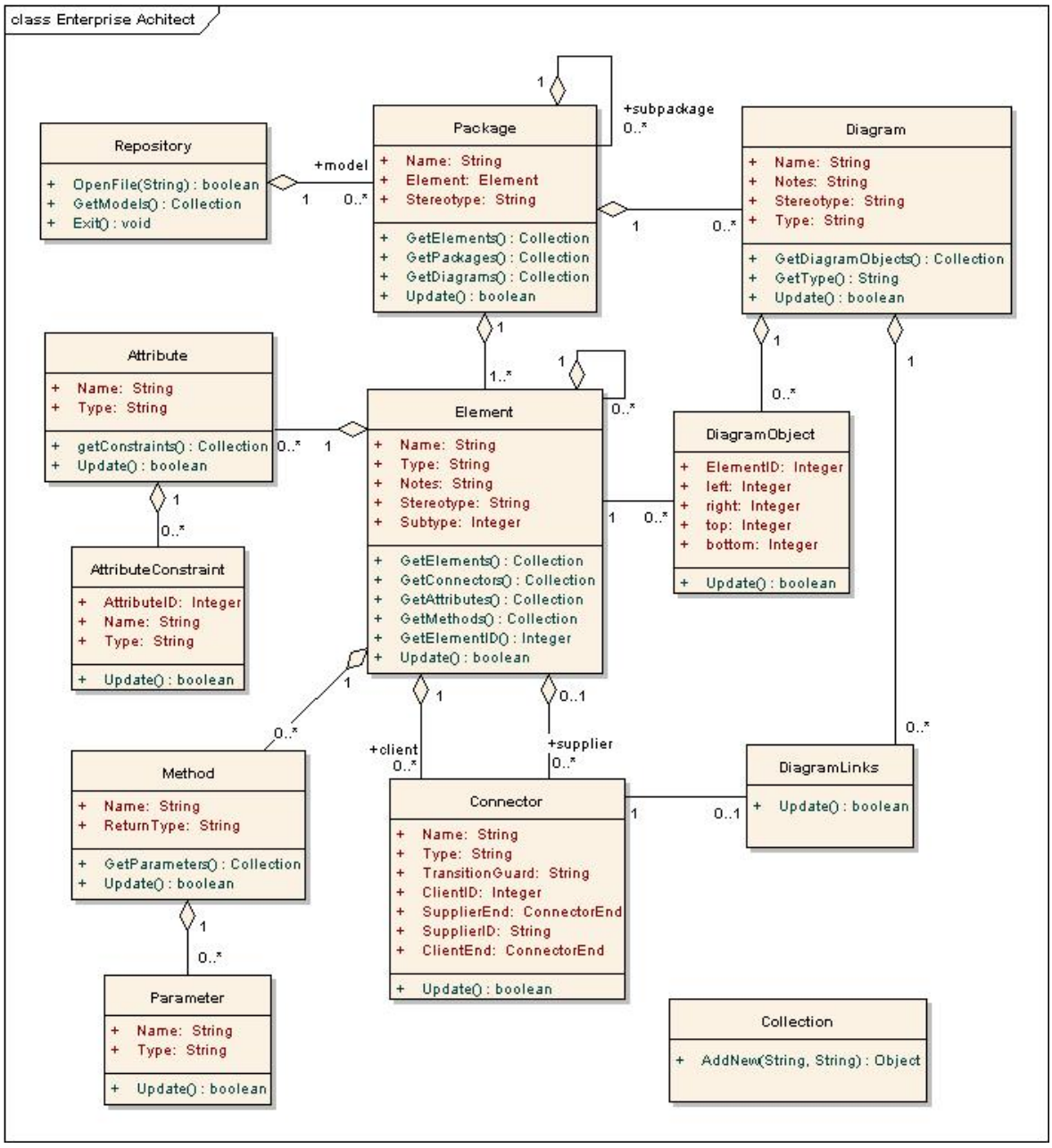

Abbildung 2.1: Die wichtigsten Klassen des EA-API

gibt ihr den Pfad zur *eap*-Datei.
Nach dem Ausführen der *OpenFile* Methode spiegelt das Objekt *Repository* den Inhalt des Projekts aus der *eap*-Datei wider.
Auffällig ist, dass immer nur eine *eap*-Datei geöffnet werden kann. Es führt sogar zu Fehlern, wenn versucht wird, eine weitere *eap*-Datei zu öffnen, auch wenn die Alte vorher geschlossen wurde. Vermutlich entsteht dieses Verhalten aufgrund eines Fehlers im API.
Die Tatsache, dass nur eine *eap*-Datei bearbeitet werden kann, weist auf eine Umsetzung des Repositorys als Singleton hin. Allerdings wurden bei der Umsetzung in der EA-API nicht die Richtlinien des Singleton-Patterns [GHJV95] umgesetzt, da zum Beispiel ein Konstruktor-Aufruf mit *new* ausgeführt werden kann.

Kollektion. Die Klasse *Collection* wird im Enterprise Architect immer dann verwendet, wenn ein Objekt viele Instanzen einer anderen Klasse enthalten soll. Die Aggregationen aus dem Klassendiagramm in Abb. 2.1 werden durch Kollektionen umgesetzt.
Die wichtigste Methode dieser Klasse ist *AddNew*. Durch *AddNew* wird ein neues Objekt erzeugt und in eine Kollektion des Enterprise Architects eingefügt. *AddNew* liefert als Rückgabewert die Instanz einer Klasse, welche in die Kollektion hinzugefügt werden soll. Daher kann *AddNew* auch nicht das neu hinzuzufügende Objekt als Parameter übergeben werden, sondern nur Strings, die dieses Objekt weiter beschreiben. Der Rückgabewert von *AddNew* muss also immer vom gleichen Typ sein, wie die Elemente innerhalb der Kollektion, auf die *AddNew* ausgeführt wird.
Neue Objekte werden mit *AddNew* immer durch zwei Strings erzeugt. Ein String steht für den Namen, der andere beschreibt den Typ des neuen Objektes. Der Typ entscheidet zum Beispiel über die Diagrammart, also zum Beispiel ob das neue Objekt ein Klassen- oder Anwendungsfalldiagramm werden soll.
Das durch *AddNew* neu erzeugte Objekt muss noch in das Repository geschrieben werden. Dazu besitzen alle Klassen, die durch *AddNew* in eine Kollektion eingefügt werden können, eine *Update*-Methode. Mit Hilfe der *Update*-Methode kann also ein Objekt in das Repository geschrieben werden.
Besonders bei den Kollektionen erkennt man die Verwendung des Analogieprinzips in dem EA-API. So werden Instanzen vieler Klassen immer mit *AddNew* erzeugt und in die Kollektion eingefügt. Dabei wird jede Instanz zuerst nur durch die beiden *AddNew* Parameter *Name* und *Type* festgelegt und schreibt sich durch die *Update*-Methode ins Repository. Wenn man an einer Instanz durch Setter-Methoden noch weitere Attribute setzt, so braucht man die *Update*-Methode nur einmal für jede Instanz aufzurufen, um sie ins Repository zu schreiben. Falls *Update* nicht ausgeführt wird, wird auch das Objekt nicht im Repository gespeichert und wird spätestens beim Beenden des Programms, das das EA-API verwendet, aus dem

Speicher gelöscht.

Allgemein kann über die verwendeten Attribute innerhalb des EA-API gesagt werden, dass sie entweder nur einen Lesezugriff oder einen Lese- und Schreibzugriff bieten. Im UML-Klassendiagramm aus Abb. 2.1 sind Attribute, die dem Programmierer sowohl Lese- als auch Schreibzugriff bieten, durch normale Attribute innerhalb einer Klasse dargestellt. Attribute innerhalb einer Klasse, die dem Programmierer nur einen Lesezugriff erlauben, diese sind zum Beispiel vom Typ *Collection*, sind im UML-Klassendiagramm nicht als Attribute gelistet sondern es werden nur die Getter-Methoden gezeigt, die diese Attribute als Rückgabewert liefern.
Es gibt auch Attribute, die als ID eines Objektes fungieren. Diese Attribute existieren, weil das Repository des Enterprise Architects auf einer JET Datenbank [Spa06b] aufsetzt. Innerhalb einer Datenbank haben alle Objekte IDs. Daher besitzen auch die Objekte innerhalb des API IDs, mit deren Hilfe sie auf die entsprechenden Objekte in der Datenbank abgebildet werden.
An dieser Stelle erkennt man auch, warum neue Instanzen immer mit *AddNew* angelegt werden müssen. Das hängt damit zusammen, dass an der entsprechenden Stelle in der Datenbank, auf die das Repository aufsetzt, ein Eintrag geschrieben werden muss. Dieser Eintrag ist dann abhängig von der Kollektion in der sich das neue Objekt befindet. Wenn man Objekte erzeugen möchte, die sich bereits in der Datenbank befinden, aber noch nicht innerhalb des Programms instanziert wurden, so kann dies mit verschiedenen Funktionen geschehen, die das Objekt aufgrund der ID aus der Datenbank erstellen. Dies wird später an Beispielen gezeigt.

Paket und Modell. Das Repository enthält beliebig viele Instanzen der Klasse *Package*. Diese *Package*-Objekte auf der höchsten Ebene werden *Modell* genannt, beachte dazu die Rollennamen in Abb. 2.1. Das Repository besitzt also eine Kollektion, die Instanzen der Klasse *Package* beinhaltet, die als *Models* bezeichnet werden. Um Zugriff auf die Modelle innerhalb des Repositorys zu bekommen, gibt es die Methode *GetModels*. Diese Methode liefert als Rückgabewert eine *Collection* von *Packages*, die dann die weiteren Daten des Projektes enthalten.
Die Modelle innerhalb des Repositorys werden oft als Wurzeln bezeichnet, obwohl ein Repository beliebig viele Modelle bzw. Wurzeln haben kann. Unter einer Wurzel können sich dann weitere Pakete befinden, diese werden dann aber nicht mehr Modell sondern Unterpaket bzw. Paket genannt.
Wie man in Abb. 2.1 erkennen kann, besitzt ein Paket Kollektionen für Unterpakete, Elemente und Diagramme. Die entsprechende Getter-Methoden liefert diese Kollektionen, also

wird zum Beispiel *GetPackages* verwendet, um alle Unterpakete als *Collection* zu erhalten. Eine Besonderheit der Klasse *Package* ist, dass sie ein Attribut *Element* der Klasse *Element* besitzt. Durch dieses Attribut kann das Paket alle Eigenschaften eines Elements zusätzlich annehmen. Auffällig ist, dass diese Eigenschaft nicht durch Vererbung, sondern durch Delegation umgesetzt wurde.

Element. Im Allgemeinen werden Instanzen der Klasse *Element* am häufigsten verwendet. Ein Element kann als Klasse, als Anwendungsfall, als Knoten etc. fungieren. Um dem Element diese Rollen zuzuweisen, muss man beim Erzeugen des Elements das Attribut *Type* setzen. Will man zum Beispiel ein Element, das eine Klasse repräsentieren soll, erzeugen, so muss man *Type* den String-Wert „Class“ zuweisen. Da ein Objekt *Element*, wie jedes andere Objekt innerhalb einer Kollektion, mit *AddNew* erzeugt wird, hat es auch noch das Attribut *Name*.
Wie in Abb. 2.1 zu erkennen ist, besitzt die Klasse *Element* Aggregationen zu den Klassen *Attribute*, *Method*, *Connector* und *Element*. Das bedeutet, dass ein Element beliebig viele Attribute, Methoden, Verbindungen und Unterelemente haben kann, diese werden dann nach dem Analogieprinzip innerhalb von Kollektionen gesammelt, auf die per *GetAttributes*, per *GetMethods*, per *GetConnectors* bzw. per *GetElements* zugegriffen werden kann.

Attribut. Die Klasse *Attribute* wird genutzt, um eine Instanz eines *Elements* mit Attributen zu erweitern. Beim Erzeugen eines Attributs muss *Name* und *Type* gesetzt werden. Dies wird aber durch die Verwendung von *AddNew* erzwungen, da sich neue Attribute immer in einer Kollektion eines Elementes befinden müssen.
Die Klasse *Attribute* besitzt eine Aggregation zur Klasse *AttributeConstraint*, diese wird, wie gewohnt, mit Hilfe einer Kollektion innerhalb der Klasse *Attribute* realisiert.
AttributeConstraints erweitern Attribute mit Bedingungen. Beim Erzeugen eines solchen *AttributeConstraints* muss *Name* und *Type* angegeben werden, dabei steht *Name* für die Bedingung an sich, zum Beispiel „> 17“, und der *Type* beschreibt den Typ der Bedingung, dies könnte zum Beispiel „Precision“ sein. Da jede Bedingung genau einem Attribut zugeordnet ist, gibt es auch noch das Attribut *AttributeID* innerhalb der Klasse *AttributeConstraint*. Dieses verweist dann auf die ID des *Attribute*-Objektes, in dessen Kollektion sich die Bedingung befindet.

Methode. Methoden werden ähnlich wie Attribute verwendet. Sie werden mit Hilfe der Klasse *Method* instanziert. Wie jedes Objekt, dass in einer Kollektion gesammelt wird, kann es nur per *AddNew* neu erzeugt werden. Die beiden Parameter von *AddNew* legen dann den Namen und den Rückgabetyp der neuen Methode fest.

Eine neu erzeugte Methode hat zuerst noch keine Parameter. Jede Methode besitzt eine Kollektion *Parameters*, auf die mit *GetParameters* zugegriffen werden kann. In dieser Kollektion werden dann die Parameter einer Methode gesammelt. Parameter sind Instanzen der Klasse *Parameter*, ihre wichtigsten Attribute sind *Name* und *Type*, die den Namen und den Attributtyp, zum Beispiel „int“ festlegen.

Beziehung. Die Klasse *Connector* verbindet zwei nicht notwendig verschiedene Elemente. Ein Element besitzt eine *Collection*, in der Beziehungen gesammelt werden. Wenn eine Beziehung zwei Elemente verbindet, so befindet sie sich in beiden Kollektionen der jeweiligen Elemente. Die Beziehung steht zu dem Element, auf dessen Kollektion *AddNew* angewendet wurde, in der Rolle des *Clients* bzw. dem Startpunkt der Beziehung. Die Rolle des *Suppliers* wird anschließend durch eine ID festgelegt. Dazu besitzt jeder Connector ein Attribut *SupplierID*, in diesem Attribut wird dann die ID des Elementes eingetragen das als *Supplier* bzw. Ziel der Beziehung fungiert. Es gibt aber auch noch das Attribut *ClientID*, in diesem wird die ID, die auf das Element in der Client-Rolle verweist, gespeichert. Sowohl auf die *SupplierID* als auch auf die *ClientID* besteht Lese- und Schreibzugriff, so dass man Start- und Endpunkt einer Beziehungen beliebig verändern kann. Weitere Details der Beziehung beinhalten die Attribute *SupplierEnd* und *ClientEnd*. Beide sind vom Typ *ConnectorEnd*. Eine Instanz der Klasse *ConnectorEnd* speichert zum Beispiel den Rollennamen oder die Multiplizität am jeweiligen Ende der Beziehung.

Diagramm. Bisher wurde noch nicht gezeigt, wie Diagramme erzeugt werden können. Es wurden zwar die wichtigsten Bestandteile der Projekte vorgestellt, aber noch nicht, was man tun muss, um diese mit Hilfe des EA-API grafisch darzustellen.
Zur grafischen Darstellung von Projekten gibt es die Klasse *Diagram*. *Diagram* Instanzen werden in einer Kollektion innerhalb der Klasse *Package* gesammelt. Daher werden sie auch mit *AddNew* erzeugt, dabei werden wie gewohnt *Name* und *Type* gesetzt. Durch *Type* wird festgelegt welche Diagrammart erzeugt werden soll. Beispielwerte für *Type* sind: „Analysis“ für vereinfachte Aktivitätsdiagramme, „Component“ für Komponenten-Diagramme, „Sequence“ für Sequenz-Diagramme, „Use Case“ für Anwendungsfall-Diagramme.
Um das Diagramm zu befüllen, muss man Instanzen der Klasse *DiagramObject* in die *Collection DiagramObjects* des Diagramms einfügen. Der Zugriff auf die *DiagramObjects*-Kollektion wird durch *GetDiagramObjects* ermöglicht.

Diagramm-Objekt. Ein Objekt der Klasse *DiagramObject* hat die folgenden Attribute vom Typ Integer zur Positionsbestimmung im Diagramm: *left*, *right*, *top* und *bottom*. Diese Werte definieren die bounding box des Elementes im Diagramm.

Allerdings ist eine Instanz eines *DiagramObjects* nur ein Platzhalter für ein Objekt der Klasse *Element*. Daher besitzt die Klasse *DiagramObject* das Attribut *ElementID*, diese ID referenziert das Element, das im Diagramm an der festgelegten Position erscheinen soll.
Dies wird in Abb. 2.1 durch eine Assoziation dargestellt. Jedem Diagramm-Objekt ist ein Element zugewiesen, ein Element kann jedoch von beliebig vielen Diagramm-Objekten referenziert werden.

Diagramm-Beziehungen. So wie es für die Elemente Diagramm-Objekte gibt, gibt es für die Beziehungen Diagramm-Beziehungen. Diagramm-Beziehungen werden durch die Klasse *DiagramLinks* realisiert. Eine Diagramm-Beziehung ist ein Platzhalter im Diagramm für eine Instanz der Klasse *Connector*
Wenn eine Beziehung zwischen zwei Elementen, die durch Diagramm-Objekte in einem Diagramm vertreten werden, instanziert wird, so wird automatisch eine Instanz der Klasse *DiagramLinks* angelegt.

Mit Hilfe der bis hierhin vorgestellten Klassen ist es möglich, im Enterprise Architect Projekte anzulegen, zu manipulieren oder auszulesen. Die vorgestellten Klassen haben noch viele andere Attribute, die benutzt werden können, um weitere Feinheiten zu modellieren. Wenn man sich in diese Möglichkeiten einarbeiten möchte, so sollte man den **EAUserGuide** [Spa06b] benutzen.

Beim Analysieren des EA-API fallen einige unschöne Dinge auf, zum Beispiel wird in dem API für Java kein JavaDoc verwendet. Die Dokumentation der einzelnen Klassen befindet sich extern im EAUserGuide.
Ein weiteres Manko des EA-API ist die Tatsache, dass es keine Vererbung gibt. Alle Klassen befinden sich in der gleichen Hierarchie-Ebene. Das erschwert die Verwendung des API unnötig, was auch an später folgenden Code-Beispielen zu erkennen ist.
Wie man schon an diesen Punkten erkennen kann, verletzt das EA-API die Richtlinien des Java Styleguides [Sun99]. Eine weitere Styleguide Verletzung ist die Tatsache, dass sämtliche Methoden- und viele Attributnamen mit einem Großbuchstaben beginnen.

2.3 Einbinden des EA-API in ein Projekt

Wenn man das API des Enterprise Architects im eigenen Java Code verwenden möchte, so muss man zuerst den Windows Pfad um den Pfad zur Datei *SSJavaCOM.dll* erweitern. Außerdem muss die *eaapi.jar* in den Java CLASSPATH eingebunden werden. Diese beiden

Dateien gehören zum Enterprise Architect, werden aber nur mit der **Corporate Edition** ausgeliefert[1]. Der Programmierer findet anschließend alle Klassen des EA-API im Java-package *org.sparx*.

2.4 Erzeugen eines Beispiel-Projektes

Als nächstes soll anhand eines Beispiels gezeigt werden, wie ein einfaches Projekt mit Hilfe des EA-API erstellt wird. Das visuelle Ergebnis soll ein Diagramm mit zwei Klassen „*Man*“ und „*Woman*“ sein, die mit einer Assoziation „*isMarriedWith*“ verbunden sind. Abb. 2.2 zeigt dieses Diagramm.

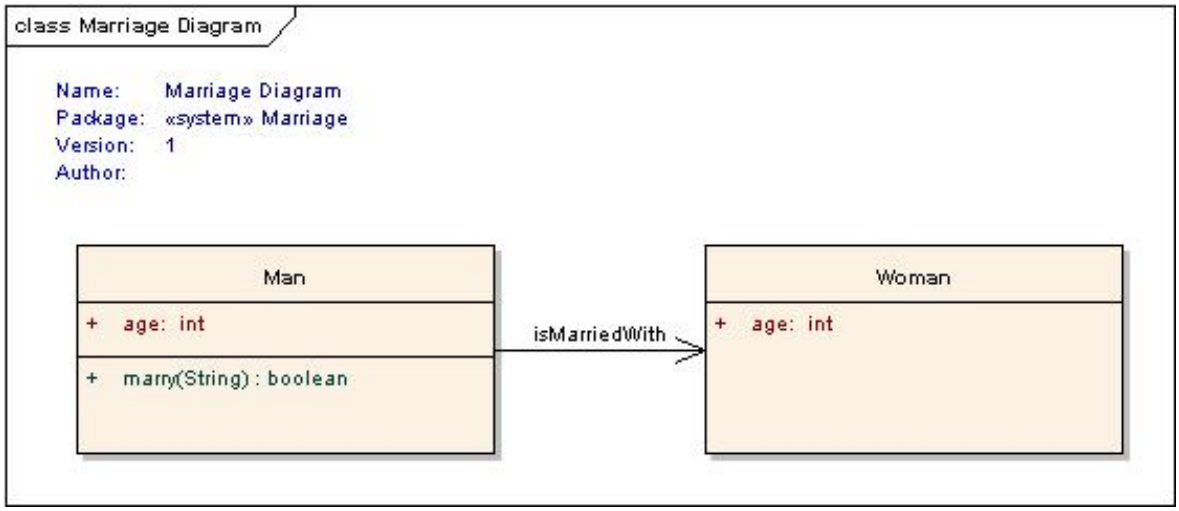

Abbildung 2.2: Beispiel Diagramm erzeugt mit Hilfe des EA-API

1. Repository Die Grundvoraussetzung zum Anlegen eines Projektes ist, dass man ein **Repository-Objekt** hat und dass in diesem Repository eine *eap*-Datei geöffnet ist. Das Repository-Objekt wird mit *new* angelegt. Zum Öffnen einer Datei wird die *OpenFile*-Methode des Repository-Objektes verwendet. Dieser muss man den Pfad zu einer *eap*-Datei als String übergeben. Der Rückgabewert von *OpenFile* ist „true“, wenn die Datei ins Repository geladen werden konnte. In Listing 2.1 ist dies gezeigt. Fehlerbehandlungen werden in allen Listings nur durch einen Kommentar angedeutet.

Listing 2.1: Repository-Objekt und laden einer eap-Datei

```
1 Repository repositoryEA = new Repository();
2 boolean isFileOpen = repositoryEA.OpenFile("C:/test.eap");
```

[1] Die Dateien *SSJavaCOM.dll* und *eaapi.jar* befinden sich ebenfalls in der freien Testversion der **Corporate Edition**

```
3 if (!isFileOpen){
4   // eap-file could not be opened.
5 }
```

2. Modell Mit Hilfe des Repository-Objektes kann nun mit dem Anlegen des Diagramms begonnen werden. Das Diagramm aus Abb. 2.2 befindet sich allerdings in einem Paket, so dass vor dem Anlegen des Diagramms ein Paket erzeugt werden muss. Dieses Paket soll aber kein Modell sein, sondern sich innerhalb eines Modells befinden. Demnach ist der nächste Schritt das Anlegen eines Modells.
Dazu nutzt man die *GetModels*-Methode des *Repository*-Objektes (hier *repositoryEA*) und erzeugt mit *AddNew* ein neues **Modell**. Der erste Parameter von *AddNew* legt den Namen des Modells fest, der zweite wird nicht benötigt, so dass ein leerer String übergeben wird. *packageNewModel* referenziert das neu erzeugte Modell. Nachdem auf *packageNewModel* ein erfolgreiches *Update* (Rückgabewert entspricht „true“) ausgeführt wurde, befindet sich das neu erzeugte Modell im Repository. Der Quellcode dazu ist in Listing 2.2 gezeigt.

Listing 2.2: Erzeugen eines Modells

```
1 Package packageNewModel =
2   repositoryEA.GetModels().AddNew("NewModel", "");
3 if (!(packageNewModel.Update())) {
4   // Model could not be written in the repository.
5 }
```

3. Paket Nun kann im neuen Modell ein **Paket**, welches das Diagramm enthalten wird, angelegt werden. Dazu verwendet man die *GetPackages*-Methode, um die Kollektion der Pakete innerhalb des Modells zu bekommen. Auf diese Kollektion wird *AddNew* angewendet, dabei wird der Paketname durch den ersten Parameter von *AddNew* gesetzt.
Wie in Abb. 2.2 zu erkennen ist, steht vor dem Paket-Namen noch der Stereotyp „*«system»*“. Dieser kann erst gesetzt werden, nachdem das Paket durch *Update* ins Repository geschrieben wurde. Dazu benutzt man die Methode *SetStereotype*, diese muss man allerdings auf das Element anwenden, welches das Paket näher beschreibt. Anschließend muss ein zweites *Update* ausgeführt werden. Diese beiden *Updates* sind nötig, da vor dem ersten *Update* noch nicht auf das Element, das Teil des Pakets ist, zugegriffen werden konnte. Listing 2.3 zeigt diese Schritte.

Listing 2.3: Erzeugen eines Paketes

```
1 Package packageMarriage =
```

```
  packageNewModel.GetPackages().AddNew("Marriage", "");
if (packageMarriage.Update()){
  packageMarriage.GetElement().SetStereotype("system");
  if (!packageMarriage.Update()) {
    // Stereotype could not be added.
  }
} else {
  // Package could not be written in the repository.
}
```

4. Element In dem soeben erzeugten Paket können nun Elemente und Diagramme erzeugt werden. Für das Beispiel aus Abb. 2.2 werden zwei **Elemente**, die als die Klassen „*Man*“ und “*Woman*“ fungieren sollen, benötigt.
Durch die *GetElements*-Methode des Paketes und die *AddNew*-Methode kann man ein neues Element erzeugen. Wichtig ist hierbei, dass über den zweiten Parameter von *AddNew* der Typ des Elementes festgelegt wird. Um eine Klasse zu erhalten, verwendet man *Class*. Nach dem erfolgreichen Ausführen von *Update*, befindet sich die Klasse „*Man*“ im Paket „Marriage“. Das Erzeugen der Klasse „*Woman*“ funktioniert analog, siehe dazu Listing 2.4.

Listing 2.4: Erzeugen von zwei Elementen

```
Element classMan =
  packageMarriage.GetElements().AddNew("Man", "Class");
if(!(classMan.Update())){
  // Element could not be written in the repository.
}

Element classWoman =
  packageMarriage.GetElements().AddNew("Woman", "Class");
if(!(classWoman.Update())){
  // Element could not be written in the repository.
}
```

5. Diagramm In Listing 2.5 wird gezeigt, wie ein neues **Diagramm** in einem Paket erzeugt wird. Dabei ist zu beachten, dass der zweite Parameter von *AddNew* den Diagrammtyp bestimmt, hier wurde „Logical“ gewählt.

Listing 2.5: Erzeugen eines Diagramms

```
Diagram logicalMarriage = packageMarriage.GetDiagrams()
```

```
2    .AddNew("Marriage␣Diagram", "Logical");
3  if(!logicalMarriage.Update()){
4    // Diagram could not be written in the repository.
5  }
```

6. Diagramm-Objekt Damit die beiden Klassen „*Man*“ und „*Woman*“ innerhalb des Diagramms angezeigt werden, muss für jede Klasse eine Instanz von *DiagramObject* erzeugt werden. Man benutzt die *GetDiagramObjects*-Methode des gewünschten Diagramms, um dann mit *AddNew* neue **Diagramm-Objekte** anzulegen. Dabei wird im ersten Parameter von *AddNew* die Position innerhalb des Diagramms durch einen String festgelegt, der zweite Parameter ist unwichtig. Bevor das neue Diagramm-Objekt mit *Update* ins Repository geschrieben wird, sollte ihm noch das Element, das es vertreten soll, zugewiesen werden, dies geschieht mit *SetElementID*. Die ID eines Elements kann man mit *GetElementID* herausfinden. Listing 2.6 beschreibt, wie man zwei Diagramm-Objekte für die Elemente *classMan* und *classWoman* anlegt.

Listing 2.6: Erzeugen von zwei Diagramm-Objekten

```
1  DiagramObject diagramobjectMan = logicalMarriage
2    .GetDiagramObjects().AddNew("l=50;r=250;t=100;b=200;", "");
3  diagramobjectMan.SetElementID(classMan.GetElementID());
4  if(!diagramobjectMan.Update()){
5    // DiagramObject could not be written in the repository.
6  }
7
8  DiagramObject diagramobjectWoman = logicalMarriage
9    .GetDiagramObjects().AddNew("l=350;r=550;t=100;b=200;","");
10 diagramobjectWoman.SetElementID(classWoman.GetElementID());
11 if(!diagramobjectWoman.Update()){
12   // DiagramObject could not be written in the repository.
13 }
```

7. Beziehung Um die beiden Klassen „*Man*“ und „*Woman*“ durch eine **Beziehung** zu verbinden, wird ein neuer *Connector* instanziert. Da diese Beziehung in Abb. 2.2 vom „*Man*“ zur „*Woman*“ verläuft, muss auf die Connector-Kollektion des Elementes „*Man*“ ein *AddNew* ausgeführt werden. *AddNew* bekommt als Name „isMarriedWith“ und als Typ „Association“ zugewiesen, so dass die Beziehung die Rolle einer Assoziation annimmt.

Die Rolle des Clients wird dabei automatisch an die Klasse „*Man*“ vergeben, da auf deren Kollektion *AddNew* ausgeführt wurde. Der Supplier muss mittels *SetSupplierID* explizit gesetzt werden. Die ID der Klasse „*Woman*“ bekommt man durch *GetElementID*. Listing 2.7 zeigt die dazu nötigen Schritte.

Listing 2.7: Erzeugen einer Beziehung

```
Connector associationIsMarriedWith = classMan.GetConnectors()
  .AddNew("isMarriedWith", "Association");
associationIsMarriedWith
  .SetSupplierID(classWoman.GetElementID());
if(!(associationIsMarriedWith.Update())){
  // Connector could not be written in the repository.
}
```

8. Diagramm-Beziehung Die Instanz der Klasse *DiagramLinks* wird automatisch erzeugt, da sich die Beziehung aus Listing 2.7 zwischen zwei Elementen mit zugehörigen Diagramm-Objekten befindet. Aufgrund dieser automatischen Erzeugung gibt es hierzu kein Listing.

9. Attribut In Abb. 2.2 erkennt man noch, dass beide Klassen das Attribut „age“ besitzen. Um ein solches **Attribut** anzulegen, wird *AddNew* auf die *Attribute*-Kollektion eines Elementes angewendet. Dabei übergibt man „age“ und „int“ als Werte für die Parameter *name* und *type*. Die nötigen Befehle für die beiden Klassen „*Man*“ und „*Woman*“ sind in Listing 2.8 notiert.

Listing 2.8: Erzeugen von zwei Attributen

```
Attribute intAgeMan =
  classMan.GetAttributes().AddNew("age", "int");
if(!(intAgeMan.Update())){
  // Attribute could not be written in the repository.
}

Attribute intAgeWoman =
  classWoman.GetAttributes().AddNew("age", "int");
if(!(intAgeWoman.Update())){
  // Attribute could not be written in the repository.
}
```

10. Methode Als letztes fehlt nur noch die **Methode** „marry“. Diese wird ähnlich wie ein Attribut angelegt, jedoch wird *AddNew* auf die *Method*-Kollektion ausgeführt und der zweite Parameter entspricht dem Rückgabewert (hier: „boolean“).
Erst nachdem *Update* für die neue Methode ausgeführt wurden, kann der noch fehlende **Parameter** erzeugt werden. Dazu wird auf die *Parameter*-Kollektion der Methode ein *AddNew* ausgeführt. Die Parameter von *AddNew* entsprechen denen bei der Instanz eines Attributs. Das Erzeugen einer Methode mit Parameter wird in Listing 2.9 dargestellt.

Listing 2.9: Erzeugen einer Methode mit Parameter

```
1 Method booleanMarry =
2   classMan.GetMethods().AddNew("marry", "boolean");
3 if(booleanMarry.Update()){
4   Parameter stringNameWoman = booleanMarry.GetParameters()
5     .AddNew("nameWoman", "String");
6   if(!(stringNameWoman.Update())){
7     // Parameter could not be written in the repository.
8   }
9 } else {
10  // Method could not be written in the repository.
11 }
```

2.5 Lesen im EA-Repository

In den Beispielen bisher wurde nur gezeigt, wie ins Repository geschrieben wird. Das Auslesen wurde nicht gezeigt. Wenn man eine beliebige Instanz innerhalb des Repositorys hat, so kann man deren Attribute mit Hilfe von **Getter-Methoden** auslesen. Möchte man zum Beispiel den Stereotypen des Objektes *packageMarriage* von Listing 2.3 auslesen, so würde man die Methode *GetStereotype* verwenden.
Wenn alle Objekte innerhalb einer Kollektion ausgelesen werden sollen, so kann dies mit Hilfe eines Iterators geschehen. Dazu sind die Methoden des Interfaces **Iterator** der Java-Standard-Bibliothek in der Kollektion des Enterprise Architects implementiert. Um über die Kollektion zu interieren, kann die For-Each Schleife verwendet werden. Sie wird in Listing 2.5 gezeigt und iteriert in diesem Beispiel über alle Objekte innerhalb der Modell-Kollektion.

```
1 for (Package modelToRead : repositoryEA.GetModels()){
2   // Method shows the content of a package or a model.
3   showPackageDetails(modelToRead);
4 }
```

2.6 Bewertung

Ursprünglich bestand die Hoffnung, dass der Enterprise Architect über das API ein Metamodell bereitstellt, welches dann für Transformationen genutzt werden kann. Der Enterprise Architect besitzt jedoch nur ein sehr schwaches Metamodell. So ist es zum Beispiel möglich in einem Aktivitätsdiagramm UseCases, Lebenslinien, Klassen, Objekte, Akteure usw. zu erzeugen. Diese Tatsache ist im Hinblick auf die angedachten Funktionen des Überführungstools sehr enttäuschend, da man nun von Hand ein Metamodell erzeugen muss, das die Basis der Überführung ist. Dieses Metamodell bzw. Schema wird in Abschnitt 5.1 definiert.

Wie bereits an verschiedenen Stellen in diesem Kapitel erwähnt, besitzt das EA-API noch weitere Mängel, diese werden nun abschließend gelistet.

- Es kann immer nur eine *eap*-Datei geöffnet werden. Das Öffnen eines anderen EA Projektes führt zu Fehlern.
- Der Quellcode des EA-API ist nicht kommentiert.
- Innerhalb des EA-API wird keine Vererbung verwendet - alle Klassen befinden sich auf gleicher Hierarchie-Ebene.
- Der Quellcode des EA-API richtet sich nicht nach dem Java Styleguide [Sun99].
- Zum Anzeigen von Ausnahmen verwendet das EA-API keine Exceptions, sondern liefert „false“ als Rückgabewert der entsprechenden Methode.

3 Java Graphenlabor

In diesem Kapitel soll das Java Graphenlabor genauer vorgestellt werden. Da es auf TGraphen operiert, ist es wichtig den Begriff TGraph vorab zu klären. Anschließend wird JGraLab genauer erklärt und gezeigt, wie man es verwenden kann.

3.1 TGraphen

Ein Graph besteht aus Knoten und Kanten, wobei eine Kante zwei Knoten verbindet. Weiterführendes zu Graphen findet der Leser in [Cha85].
TGraphen sind eine spezielle Art von Graphen, sie werden durch die folgenden Eigenschaften definiert:

- Typisiert: Sämtliche Elemente des TGraphen sind typisiert.
- Attributiert: Alle Elemente des Graphen dürfen Attribut-Werte-Paare besitzen.
- Gerichtet: TGraphen sind immer gerichtet. Allerdings können sie ohne Änderung der Darstellung auch als ungerichtete Graphen betrachtet werden, so dass zum Beispiel gleichzeitig eine gerichtete und eine ungerichtete Graphen-Traversierung durchgeführt werden kann.
- Angeordnet: Die Graphenelemente innerhalb eines Graphen befinden sich prinzipiell in einigen Anordnungen untereinander. So gibt es Anordnungen der Knoten, Anordnungen der Kanten und Anordnungen der Inzidenzen.

Eine formale Definition von TGraphen kann in [Ste06] gefunden werden. Weiterführende Erklärungen zu TGraphen besonders im Bezug zu JGraLab befinden sich in [Kah06].

3.2 Überblick JGraLab

Ursprünglich wurde das Graphenlabor [DW03] in C entwickelt. Diese C Versionen und spätere C++ Versionen waren in erster Linie auf Geschwindigkeit bezüglich der Traversierung von TGraphen ausgerichtet.

Auf Grund des hohen Zuwachses an Rechenleistung in den letzten Jahren ist das Graphenlabor allerdings nicht mehr auf eine Umsetzung in C++ oder ähnlich performanten Sprachen angewiesen. Daher wurde im Zuge einer Diplomarbeit an der Universität Koblenz [Kah06] JGraLab entwickelt. Dies ist eine Umsetzung des C++ Graphenlabors in Java. Dabei sollten vor allem die Vorteile der Objektorientierung und der Plattformunabhängigkeit in das Graphenlabor einfliessen.

JGraLab wird zum Erstellen von Graphen verwendet. Das Erstellen geschieht in zwei Schritten. Zuerst muss das Schema des Graphen definiert und generiert werden. Generieren heißt, dass JGraLab anhand des definierten Schemas Java Klassen generiert, die dann zur Instanzierung des eigentlichen Graphen genutzt werden können. Die Instanzierung mit Hilfe der generierten Klassen ist dann der zweite Schritt.

Um JGraLab innerhalb eines Projekts nutzen zu können, muss man lediglich die Datei *jgralab.jar* [Kah06] einbinden. Diese *jar*-Datei fungiert als Bibliothek, welche die nötigen Klassen zur Nutzung von JGraLab bereitstellt.

3.3 Erstellen eines Schemas

Die folgenden Kapitel sollen die Nutzung von JGraLab anhand eines Beispiels demonstrieren. Zuerst soll erklärt werden, wie ein Schema in JGraLab erstellt werden kann. Dann soll im nächsten Kapitel gezeigt werden, wie eine Instanzierung eines TGraphen anhand dieses konkreten Schemas durchgeführt werden kann. Die Nutzung des modellierten Graphen ist nicht Thema dieser Studienarbeit, weitere Information dazu findet der Leser in [SBR06].

Wenn man nun zum Beispiel ein UML-Diagramm in JGraLab erstellen möchte, so benötigt man zuerst ein **Schema**. Dieses Schema erfüllt für TGraphen eine ähnliche Rolle, wie Metamodelle für Modelle. Nachdem das Schema erstellt wurde, kann das eigentliche UML-Diagramm mit Hilfe des Schemas in JGraLab instanziert werden.
Im folgenden Beispiel soll ein stark vereinfachtes UML-Metamodell [OMG05] mit Hilfe von JGraLab erzeugt werden. Diese UML-Metamodell ist in Abb. 3.1 dargestellt. Es ist kein vollständiges und kein korrektes UML-Metamodell, dennoch vermittelt es den Eindruck, dass es möglich ist, ein komplexeres Metamodell mit JGraLab zu definieren. Mit Hilfe eines solchen Metamodells ist es anschließend möglich, konkrete Ausprägungen bzw. UML-Diagramme in JGraLab zu instanzieren. Dieses Instanzieren ist dann einer der Schritte, die nötig sind, um die Überführung des Repository-Inhaltes vom Enterprise Architect nach JGraLab zu realisieren.

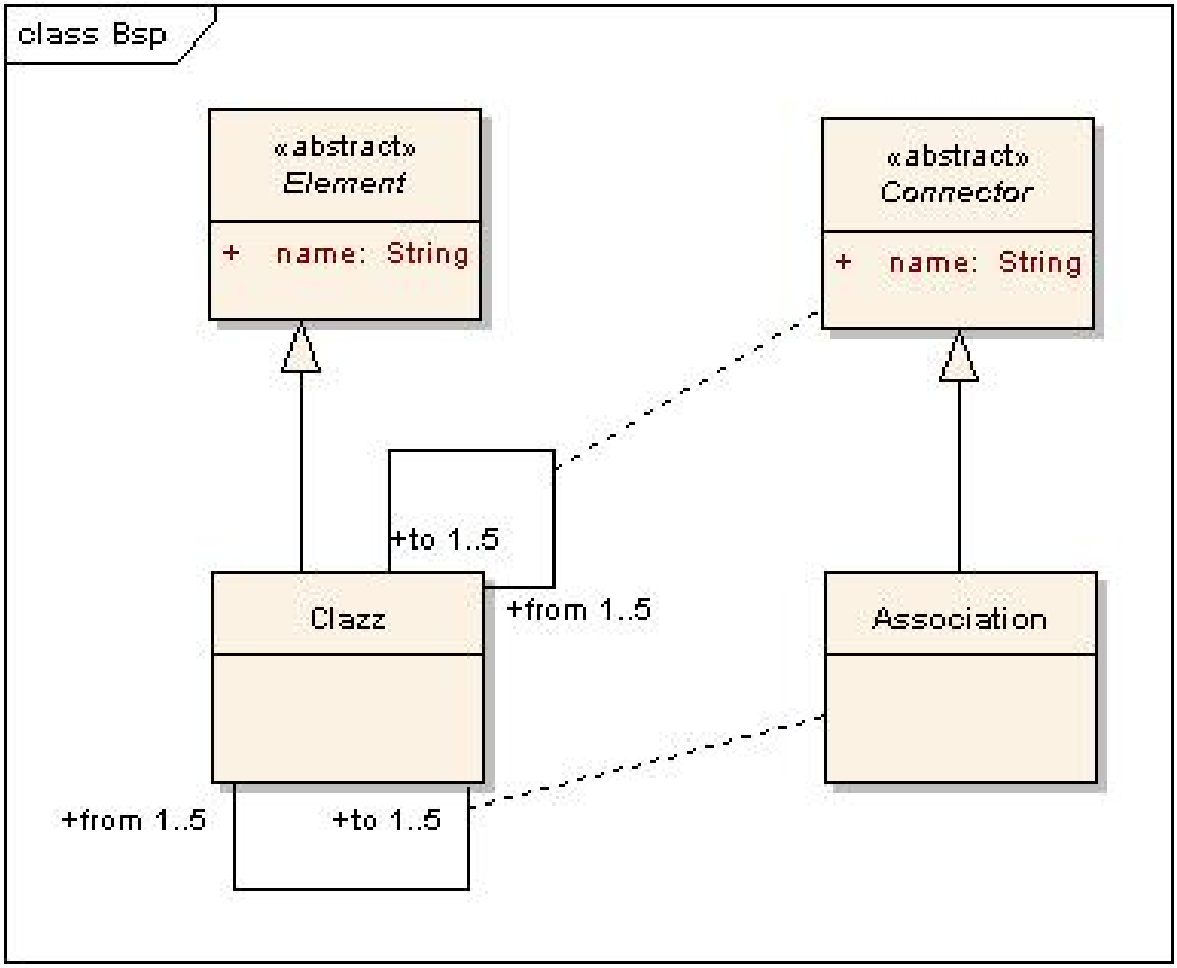

Abbildung 3.1: Klassendiagramm zum Schema aus Listing 3.1

Zum Erstellen eines Schemas gibt es zwei Möglichkeiten, Listing 3.1 zeigt, wie man mit Hilfe der Java-Klassen aus der JGraLab-Bibliothek das Diagramm aus Abb. 3.1 erzeugt. Die andere Möglichkeit verwendet eine Text-Datei, dies wird in [SBR06] genauer vorgestellt.

Im Rahmen der Studienarbeit sind nicht alle Klassen und Möglichkeiten, die JGraLab bietet, nötig. Daher werden in dem folgenden Beispiel und Klassendiagramm zu JGraLab auch nur die für die Thematik der Studienarbeit wesentlichen Klassen und Methoden vorgestellt und beschrieben.
Abb. 3.2 zeigt ein Klassendiagramm, das alle Klassen und Methoden des Beispiels aus Listing 3.1 enthält. Ein vollständiges Klassendiagramm kann man in [Kah06] finden.

Listing 3.1: Erzeugen des JGraLab-Schemas zu Abb. 3.2

```
1  Schema schemaClazzDiagram = new SchemaImpl
2    ("ClazzdiagramSchema", "classdiagram.schema");
3
4  GraphClass graphClazzDiagram =
5    schemaClazzDiagram.createGraphClass("Clazzdiagram");
6
7  VertexClass vertexElement =
8    graphClazzDiagram.createVertexClass("Element");
9  vertexElement.setAbstract(true);
10 vertexElement.addAttribute
11   ("name", schemaClazzDiagram.getDomain("String"));
12
13 VertexClass vertexClazz =
14   graphClazzDiagram.createVertexClass("Clazz");
15 vertexClazz.addSuperClass(vertexElement);
16
17 EdgeClass edgeConnector = graphClazzDiagram.createEdgeClass
18   ("Connector", vertexClazz, 1, 5, "from",
19     vertexClazz, 1, 5, "to");
20 edgeConnector.setAbstract(true);
21 edgeConnector.addAttribute
22   ("name", schemaClazzDiagram.getDomain("String"));
23
24 EdgeClass edgeAssociation = graphClazzDiagram.createEdgeClass
25   ("Association", vertexClazz, 1, 5, "from",
26     vertexClazz, 1, 5, "to");
```

```
27 edgeAssociation.addSuperClass(edgeConnector);
28
29 schemaClazzDiagram.commit("./generated");
```

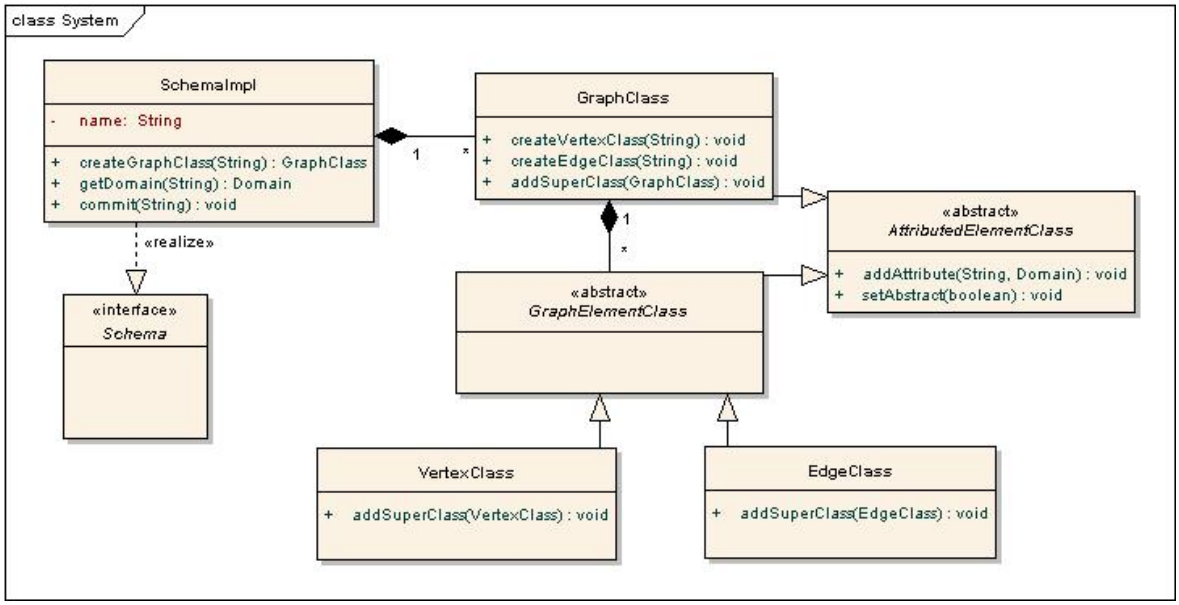

Abbildung 3.2: Vereinfachtes Klassendiagramm zu JGraLab

Schema. Zuerst benötigt man ein *Schema*-Objekt. Da JGraLab Interfaces und Implementation trennt (siehe Abb. 3.2) muss der Konstruktor *SchemaImpl* verwendet werden, um ein Objekt, welches das Interface *Schema* implementiert, zu erstellen. Die beiden Parameter im Konstruktor sind der Name des Schemas und die Java-package-Struktur, jeweils vom Typ String. Dies ist in Zeile 1 von Listing 3.1 zu sehen.

GraphClass. Eine *GraphClass* Instanz ist Bestandteil eines *Schema*-Objekts. Daher besitzt die Klasse *Schema* die Methode *createGraphClass*, mit deren Hilfe eine neue Instanz von *GraphClass* innerhalb eines Schemas angelegt werden kann, siehe Zeile 4.
Die Klasse *GraphClass* dient als Container für Spezialisierungen der abstrakten Klasse *GraphElementClass*. Außerdem besitzt sie die beiden Methoden *createVertexClass* und *createEdgeClass*, um neue Knoten- und Kanten-Klassen zu erzeugen.

AttributedElementClass. Es gibt zwei Spezialisierungen der Klasse *AttributedElementClass*. Diese sind *GraphElementClass* und *GraphClass*, also Graphen und deren Elemente. Um die Eigenschaft „attributiert“ des TGraphen umzusetzen, besitzt diese Klasse die Methode *addAttribute*. Die Methode *addAttribute* besitzt zwei Parameter, der erste ist vom

Typ String und legt den Namen des Attributs fest, der zweite ist eine Instanz der Klasse *Domain*. Die Klasse *Domain* wird zur Vereinfachung nicht in Abb. 3.2 gezeigt, im Kontext dieser Methode wird durch sie die Domäne des Attributs festgelegt.
Eine weitere Methode der Klasse *AttributedElementClass* ist die Methode *setAbstract*. Sie wird verwendet, um Graph- oder Element-Klasse als abstrakt zu deklarieren.

GraphElementClass. Die *GraphElementClass*-Klasse hat zwei Spezialisierung. Diese sind die Knoten- und Kanten-Klasse, also die beiden wesentlichen Bestandteile eines Graphen. Außerdem ist sie von der Klasse *AttributedElementClass* abgeleitet, so dass sie die Methoden *addAttribute* und *setAbstract* erbt.
In den Zeilen 10 und 21 wird gezeigt, wie man zu Knoten- und Kanten-Klassen Attribute hinzufügt. Dabei fällt auf, dass die Domäne mit Hilfe der Methode *getDomain* bestimmt wird. Diese Methode muss auf eine Instanz der Klasse *Schema* angewendet werden und bekommt als String-Parameter den Typ der gewünschten Domäne übergeben. Ihr Rückgabewert ist dann das entsprechende *Domain*-Objekt.

VertexClass. Klassen von Knoten werden mit Hilfe der Klasse *VertexClass* umgesetzt. In Listing 3.1 werden zwei Knoten-Klassen erzeugt. Zum Anlegen einer Knoten-Klasse wird die Methode *createVertexClass* verwendet, ihr String-Parameter ist der Name der Knoten-Klasse.
In Zeile 7 wird die Knoten-Klasse mit dem Namen „*Element*“ erzeugt. Diese wird als abstrakt deklariert und besitzt das Attribut „name“. Im Moment ist es noch nicht möglich eine Knoten-Instanz zu erzeugen, dazu wird noch eine nicht abstrakte Knoten-Klasse benötigt. Diese wird in Zeile 13 angelegt.
Um eine Vererbungshierarchie zu erzeugen, erbt die Knoten-Klasse „*Clazz*“ von der abstrakten Knoten-Klasse „*Element*“. Dies wird mit Hilfe der Methode *addSuperClass* realisiert, siehe dazu Zeile 15. Der Parameter dieser Methode ist eine Referenz auf die Superklasse.

EdgeClass. Die Kanten-Klassen werden weitgehend analog zu den Knoten-Klassen angelegt. In Listing 3.1 wird in Zeile 17 die abstrakte Kanten-Klasse „*Connector*“ mit dem Attribut „name“ erzeugt und in Zeile 24 wird die Spezialisierung „*Association*“ von der abstrakten Kanten-Klasse „*Connector*“ angelegt.
Die Methode *createEdgeClass* zum Anlegen von Kanten-Klassen unterscheidet sich allerdings von der Methode *createVertexClass*. Die Methode *createEdgeClass* besitzt insgesamt neun Parameter. Der erste Parameter ist vom Typ String und legt den Namen der Kanten-Klasse fest. Die nächsten vier Parameter gehören zusammen und definieren die Startklasse,

die minimale Multiplizität, die maximale Multiplizität und den Rollennamen. Analog definieren die Parameter sechs bis neun die Endklasse, deren Multiplizität und Rollennamen.

commit. Ein JGraLab-Schema nutzt man zur Erzeugung von Graphen, indem man die Instanzen der, anhand des Schemas, generierten Klassen verwendet. Diese Klassen werden aufgrund des Schema generiert, so dass all diese Klassen das komplette Schema beschreiben. Um Klassen zu generieren, wird die *commit*-Methode der Klasse *Schema* verwendet. Nach dem Ausführen von commit befinden sich alle Klassen, die das Schema beschreiben, in dem Pfad, der durch den String-Parameter von *commit* beschrieben wird. Dies wird in Zeile 29 von Listing 3.1 gezeigt.

3.4 Erstellen eines Graphen

Nachdem ein Schema erzeugt wurde, können die generierten Java Klassen verwenden werden, um einen konkreten Graphen zu instanzieren. Diese Instanzierung soll in diesem Kapitel an einem Beispiel gezeigt werden. Dazu wird das Schema aus dem vorherigen Kapitel verwendet. Da im Zuge der Studienarbeit UML-Diagramme des Enterprise Architects in TGraphen überführt werden sollen, wird im Beispiel das UML-Diagramm aus Abb. 2.2 verwendet.
Dieses UML-Diagramm besteht aus zwei Objekten „Man“ und „Woman“ und einer Assoziation „isMarriedWith“. Dies kann mit dem Schema aus Abb. 3.1 modelliert werden. Die beiden Attribute „age“, die Methode „marry“ und die Paketstruktur aus dem Beispiel-Diagramm können nicht mit Hilfe dieses einfachen Schemas erzeugt werden und werden daher weggelassen.
In Listing 3.2 wird gezeigt, wie man das Beispiel-Diagramm mit Hilfe der generierten Klassen instanziert. Zur Veranschaulichung gibt es das Objektdiagramm in Abb. 3.3. Hier sind die vier verwendeten Objekte und ihre Beziehungen zueinander gezeigt.

Listing 3.2: Modellieren von Abb. 2.2 in JGraLab

```
1 Clazzdiagram clazzdiagramLogical = ClazzdiagramSchema
2   .instance().createClazzdiagram("Logical", 100, 100);
3
4 Clazz clazzMan =
5   clazzdiagramLogical.createClazzWithAttributes("Man");
6 Clazz clazzWoman =
7   clazzdiagramLogical.createClazzWithAttributes("Woman");
8
```

```
9  Association associationIsMarriedWith =
10   clazzdiagramLogical.createAssociationWithAttributes
11   (clazzMan, clazzWoman, "isMarriedWith");
12
13 GraphIO.saveGraphToTG
14   ("./marry.tg", clazzdiagramLogical, null);
```

Graph. Zuerst muss eine Instanz der generierten Klasse *Clazzdiagram* erzeugt werden, dies geschieht mit der Methode *createClazzdiagram*, siehe Zeile 1 in Listing 3.2. Diese Methode besitzt drei Parameter, der erste definiert den Namen des Graphen, hier wurde „Logical“ gewählt, da das Diagramm im Enterprise Architect von diesem Typ ist. Die beiden anderen Parameter bereiten den Graphen auf eine Anzahl an Knoten und Kanten vor. Diese Grenzen sind nicht fest und werden nur genutzt, um den internen Speicher effektiver nutzen zu können. Sollte die Anzahl der Knoten oder Kanten im Graphen überschritten werden, so wird der Speicher automatisch relativ zeitintensiv neu strukturiert.

Knoten. Anschließend werden zwei Instanzen der Klasse „*Clazz*“ erzeugt, diese repräsentieren die Objekte „Man“ und „Woman“. Zur Instanzierung wird die Methode *createClazzWithAttributes* verwendet, ihr Parameter beschreibt den Namen des Attributs. Diese *create*-Methode wird auf dem Objekt „clazzdiagramLogical“ ausgeführt, so dass die Objekte „Man“ und „Woman“ sich anschließend im richtigen Graphen befinden. Diese Schritte sieht man in den Zeilen 4 und 6.

Kante. Zuletzt wird noch die Assoziation „isMarriedWith“ zwischen den Objekten „Man“ und „Woman“ instanziert. Dies geschieht mit der Methode *createAssociationWithAttributes*. Diese Methode besitzt drei Parameter, dabei referiert der erste den Startknoten und der zweite den Endknoten. Der dritte Parameter definiert für die Assoziation den Wert des Attributs „name“, in diesem Fall „isMarriedWith“. Dies wird in Zeile 9 gezeigt.

TG-Datei. Nachdem alle Knoten und Kanten im Graphen erzeugt wurden, kann der Graph in einer *tg*-Datei gespeichert werden. Dazu wird die statische Methode *saveGraphToTG* der Klasse *GraphIO* verwendet. Sie besitzt drei Parameter. Der erste Parameter legt den Pfad zur *tg*-Datei fest, der zweite referenziert den zu speichernden Graphen und der dritte Parameter referenziert eine *ProgressFunction* [Kah06], diese wird aber im Kontext des Überführungstool nicht benötigt und kann daher auf *null* gesetzt werden. Zeile 13 zeigt das Speichern in einer *tg*-Datei.

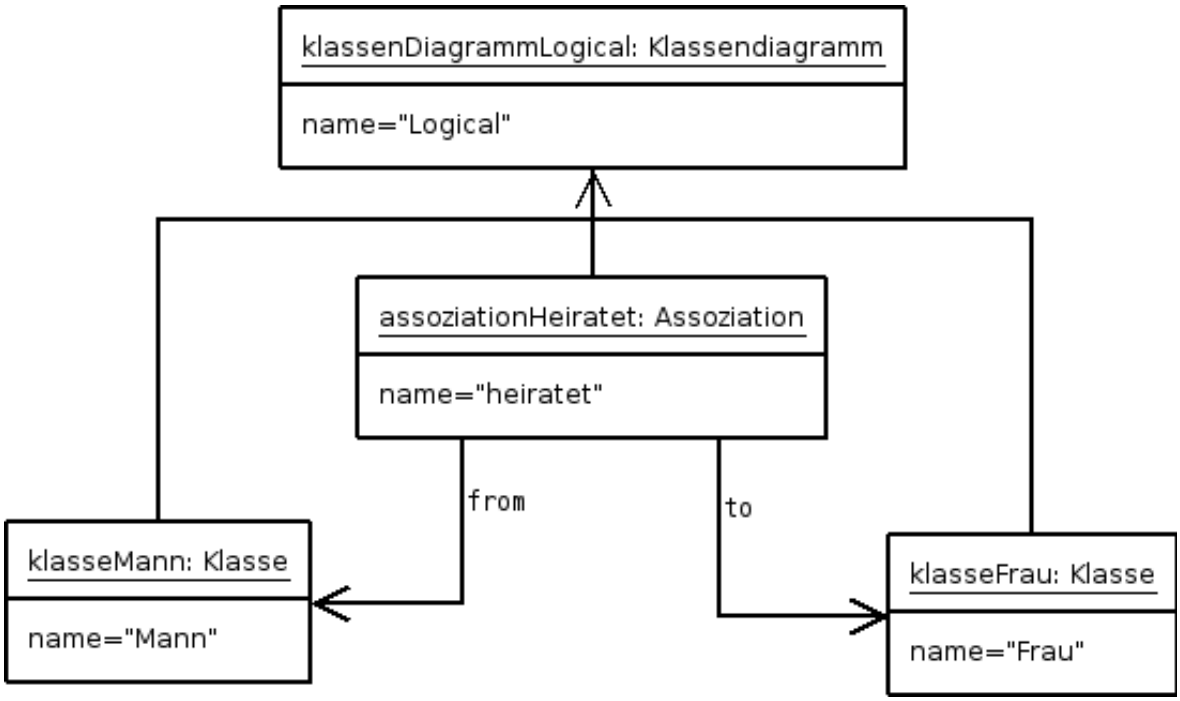

Abbildung 3.3: Objektdiagramm zum Listing 3.2

3.5 Lesen eines Graphen

Wenn man einen Graphen in das Repository des EA überführen möchte, so muss man vor dem Schreiben zuerst den Graph lesen. In diesem Abschnitt wird gezeigt, wie mit Hilfe von JGraLab ein Graph ausgelesen werden kann.
In Listing 3.3 wird eine Möglichkeit gezeigt, um alle Knoten und Kanten eines Graphen zu lesen. Dazu werden in diesem Beispiel alle Knoten in einer äußeren und alle von dem jeweiligen Knoten ausgehenden Kanten in einer inneren Schleife analysiert.

Listing 3.3: Lesen eines Graphen

```
1  Graph graph =
2    GraphIO.loadGraphFromTG("./hochzeit.tg", null);
3
4  Vertex nextVertex = graph.getFirstVertex();
5  while (nextVertex != null){
6          // Method shows details about the vertex
7          showVertexDetails(nextVertex);
8
9    Edge nextEdge =
10     nextVertex.getFirstEdge(EdgeDirection.OUT);
11   while (nextEdge != null){
```

```
12      // Method shows details about the edge
13      showEdgeDetails(nextEdge);
14              nextEdge = nextEdge.getNextEdge(EdgeDirection.OUT);
15          }
16
17    nextVertex = nextVertex.getNextVertex();
18  }
```

Zuerst wird der Graph aus einer *tg*-Datei geladen, dies geschieht mit Hilfe der statischen Methode *loadGraphFromTG*. Diese Methode besitzt zwei Parameter, wobei der erste den Pfad zur *tg*-Datei festlegt und der zweite eine *ProgressFunction* [Kah06], die hier *null* ist, referenziert. Das Laden eines Graphen ist in Zeile 1 von Listing 3.3 gezeigt.

Ein Objekt der Klasse *Graph* besitzt die Methoden *getFirstVertex* und *getNextVertex*, um über alle Knoten des Graphen zu laufen. *getFirstVertex* gibt den ersten Knoten des Graphen zurück und *getNextVertex* den nächsten. Wenn der Graph keinen nächsten oder ersten Knoten besitzt so ist der Rückgabewert beider Methoden *null*.

Die Instanzen der Klasse *Vertex* bieten die Methoden *getFirstEdge* und *getNextEdge* an, um über alle von einem Knoten ausgehenden Kanten zu iterieren. Sie werden analog zu den Methoden des Graphen verwendet. Zusätzlich kann noch festgelegt werden, ob beim Iterieren nur über die eingehenden, die ausgehenden oder alle Kanten iteriert werden soll. In Listing 3.3 wird nur über die ausgehenden Kanten iteriert.

4 Planung des Überführungstools

4.1 Aufgaben des Überführungstools

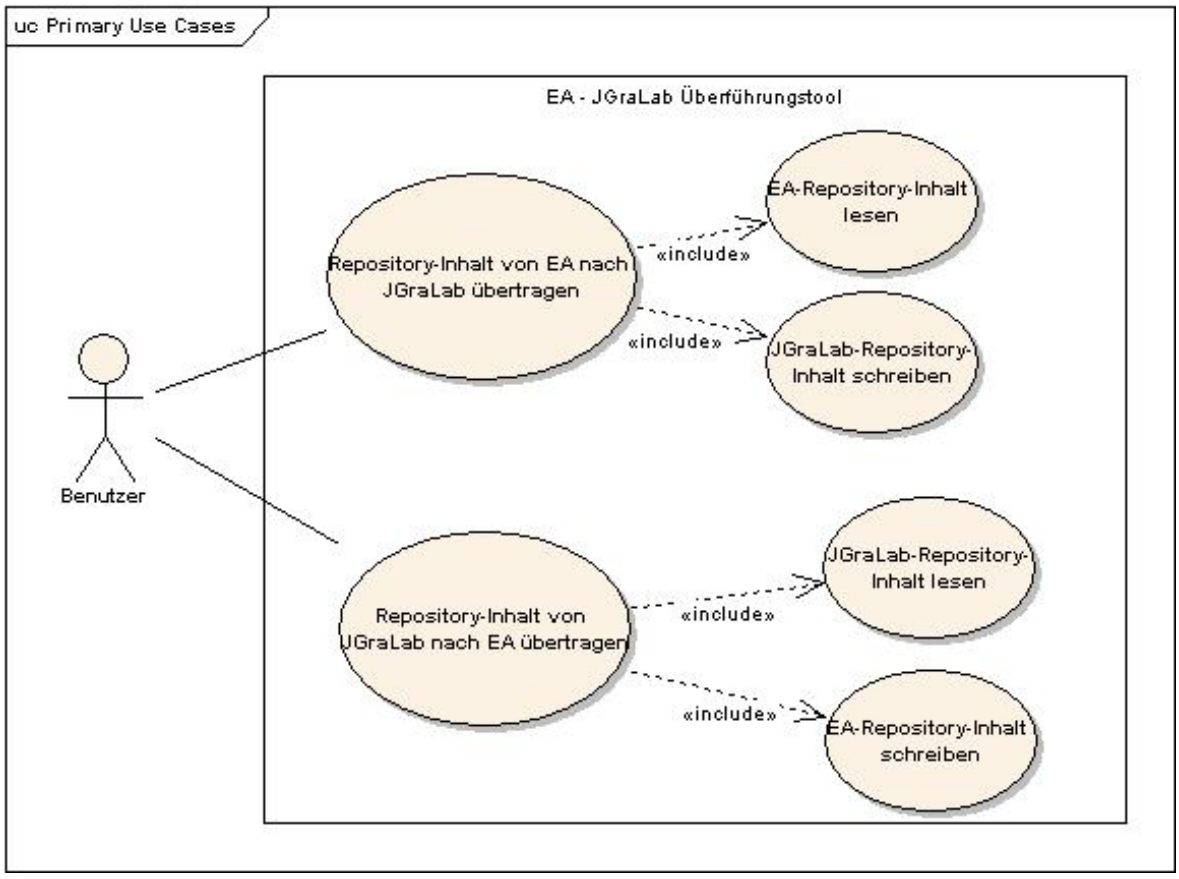

Abbildung 4.1: Anwendungsfalldiagramm des Überführungstools

Im Kontext dieser Studienarbeit soll ein Überführungstool geschrieben werden, das den Repository-Inhalt eines EA-Projektes ausliest und nach JGraLab überträgt. Falls möglich soll dieses Tool auch noch die umgekehrte Richtung realisieren, also das Auslesen des JGraLab-Repositorys und das Schreiben in das EA-Repository.
Dem Benutzer des Tools soll also die Möglichkeit geboten werden, die Repository-Inhalte in beide Richtungen zu übertragen. Zur Realisierung dieser Übertragung muss das Tool sowohl

Repository-Inhalte lesen als auch schreiben können. Diese Situation lässt sich gut in einem **Anwendungsfalldiagramm** beschreiben, siehe dazu Abb. 4.1.

4.2 Anforderungen an das Überführungstools

Nachdem man im vorherigen Abschnitt einen groben Überblick über die Funktionalität des zu entwickelnden Überführungstools bekommen hat, soll der folgende Abschnitt die Anforderungen an das Tool genau definieren.
Die Anforderungen sind in fünf Kategorien eingeteilt, es gibt funktionale Anforderungen, technische Anforderungen, Anforderungen an die Benutzerschnittstelle, Qualitätsanforderungen und sonstige Anforderungen. Die Gewichtung der Anforderungen wird mit Hilfe der Modalverben: „muss“, „soll“ und „kann“ vorgenommen. Dabei bedeutet „muss“, dass eine Anforderung zwingend umgesetzt wird und somit Pflicht ist. „soll“ hat die zweit höchste Priorität. Die niedrigste Priorität hat „kann“, es ist ein Vorschlag, der nur als optional gilt.
Die folgende Anforderungsliste beschreibt eine idealisierte Version, die vor dem eigentlichen Entwurf erstellt wurde.

Funktionale Anforderungen.

1. Das Überführungstool muss den Repository-Inhalt des Enterprise Architects auf der Basis von zu definierenden Metamodellen in ein JGraLab-Repository übertragen.
2. Das Überführungstool kann dazu in der Lage sein, den Repository-Inhalt von JGraLab, auf der Basis von zu definierenden Metamodellen, in das Repository des Enterprise Architects zu schreiben.
3. Das Metamodell soll auf angemessene Weise UML 2.0 - Diagramme beschreiben.
4. Das Überführungstool soll seine Arbeitsschritte bei jeder Überführung protokollieren.
5. Für das Überführungstool kann eine grafische Benutzungsoberfläche erstellt werden.
6. Das Überführungstool muss mit Hilfe der Konsole nutzbar sein.
7. Das Überführungstool muss anderen Anwendungen eine API-Schnittstelle bereitstellen.
8. Das Überführungstool muss mit jeder Ausführung mindestens eine Repository-Inhalts-Übertragung ermöglichen.

Technische Anforderungen.

1. Das Überführungstool muss in Java 5 geschrieben werden.
2. Das Überführungstool muss das API des Enterprise Architects verwenden.
3. Das Überführungstool muss die JGraLab-Bibliotheken verwenden.
4. Das Überführungstool soll mit Hilfe von Eclipse entwickelt werden.
5. Die Protokollierung der Arbeitsschritte soll in einer *log*-Datei festgehalten werden.
6. Die Protokollierung soll mit der Java-Klasse Logger durchgeführt werden.
7. Die Protokollierung der Arbeitsschritte kann innerhalb der Datei angemessen strukturiert werden.
8. Die grafische Benutzungsoberfläche soll möglichst nur mit Swing-Komponenten realisiert werden.
9. Das Überführungstool soll als *jar*-Datei ausgeliefert werden.
10. Der nach JGraLab zu übertragende Repository-Inhalt muss durch eine *eap*-Datei festgelegt sein.

Anforderungen an die Benutzerschnittstelle.

1. Die grafische Benutzungsoberfläche kann zum Auswählen des zu übertragenden Repositorys genutzt werden.
2. Die grafische Benutzungsoberfläche kann dem Benutzer beide Überführungsrichtungen per Button anbieten.
3. Die grafische Benutzungsoberfläche kann den Inhalt der *log*-Datei anzeigen.
4. Die grafische Benutzungsoberfläche kann dem Benutzer Fehlermeldungen anzeigen.
5. Der Benutzer muss mindestens mit Hilfe der Konsole das Überführungstool bedienen können.

Qualitätsanforderungen.

1. Das Überführungstool soll stabil laufen.
2. Fehler sollen abgefangen und behandelt oder direkt gemeldet werden.
3. Alle Methoden müssen mit JavaDoc kommentiert sein.
4. Der Quellcode kann weitere erklärende Kommentare besitzen.
5. Das Überführungstool soll nach eventueller Anpassung wiederverwendbar sein.
6. Das Überführungstool soll erweiterbar sein.
7. Das Überführungstool kann dazu in der Lage sein, möglichst performant zu arbeiten.

Sonstige Anforderungen.

1. Die Architektur des Überführungstools soll mit Hilfe eines Klassendiagramms veranschaulicht werden.
2. Die Struktur des Überführungstools soll mit Hilfe von Diagrammen gezeigt werden.
3. Die Entwicklung des Überführungstools muss in der zugehörigen Studienarbeit dokumentiert werden.
4. Kommentare im Quellcode müssen in englischer Sprache formuliert sein.
5. Die Bezeichnungen in der grafischen Benutzungsoberfläche sollen in englischer Sprache sein.
6. Die Protokollierung innerhalb der *log*-Datei muss in englischer Sprache stattfinden.
7. Fehlermeldungen müssen in englischer Sprache sein.

4.3 Architektur des Überführungstools

Komponentensicht. In diesem Abschnitt wird die Architektur des zu entwickelnden Systems beschrieben. Eine grobe Komponentensicht auf das Überführungstool wird in Abb. 4.2 gezeigt. Die wichtigste Komponente des System ist der Teil, der das Repository des Enterprise Architect nach JGraLab überführt (*TransformerEAToJGraLab*). In der Architektur hat dieser Teil aber keine wichtigere Stellung als die Rücktransformation (*TransformerJGraLabToEA*), die in einer von der Bauart ähnlichen Komponente umgesetzt wird.

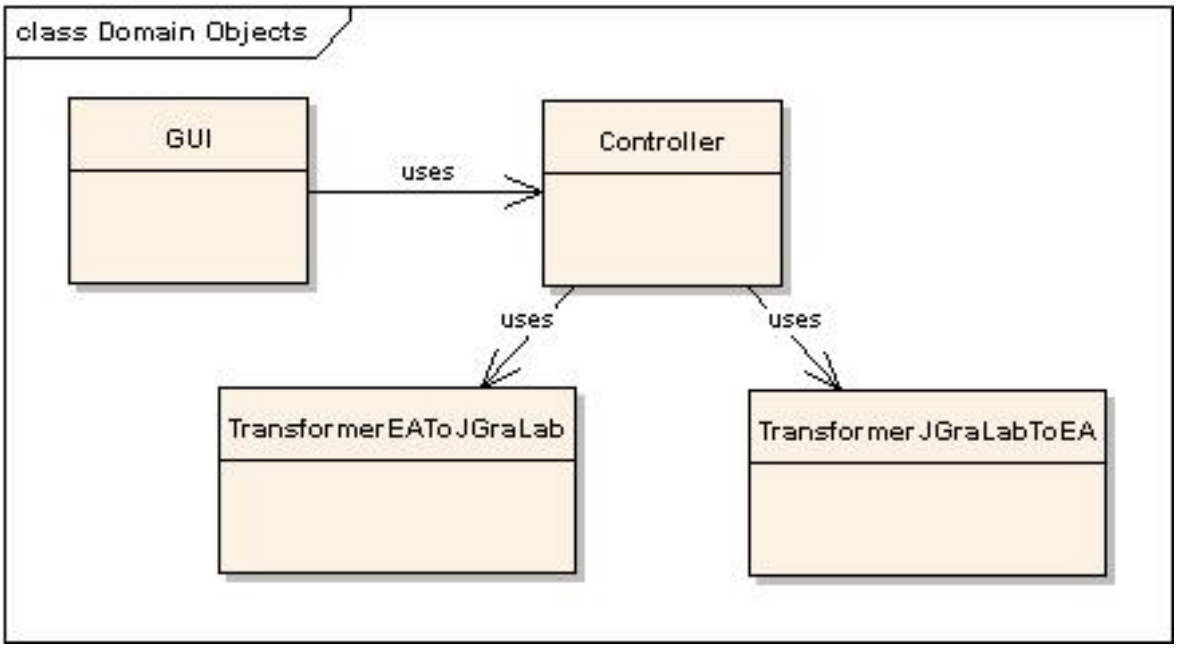

Abbildung 4.2: Komponentensicht auf das Überführungstools

Die beiden anderen Komponenten sind die grafische Benutzungsoberfläche (*GUI*) und die *Controller*-Komponente, diese bringen keine Funktionalität im eigentlichen Sinne, sondern helfen dem Benutzer das Tool zu verwenden. Dabei visualisiert die *GUI* nur die Möglichkeiten, Fehlermeldungen und Ergebnisse die das Überführungstool dem Benutzer liefert. Die *Controller*-Komponente wird verwendet, um die eigentlichen Überführungen zu konfigurieren und zu starten.

Klassendiagramm. Eine feinere Sicht auf die Architektur des Überführungstools ist in Abb. 4.3 zu sehen. Hierbei handelt es sich um ein Klassendiagramm, das alle Klassen enthält, die im Zuge der Studienarbeit geschrieben werden sollen.

Die *Controller*-Komponente wird durch die Klasse *Controller* umgesetzt. Ihre wichtigste Methoden ist *transform*. Mit Hilfe der Methode *transform* soll die Überführung ausgeführt werden. Dabei werden durch die Parameter die Pfade zu den beiden Dateien festgelegt, die das jeweilige Repository repräsentieren.
Um die Überführung durchzuführen, benötigt der *Controller* Zugriff auf *Transformer*-Klassen, die die Methode *transform* anbieten. In den *Transformer*-Klassen wird festgelegt, wie die Abbildung des einen Repository-Inhalts in das andere Repository aussehen soll. Die Klasse *Transformer* ist abstrakt, daher wird für eine konkrete Transformation, je nach Richtung, entweder die Klasse *TransformerEAToJGraLab* oder die Klasse *TransformerJGraLabToEA* verwendet.

Eine *Transformer*-Klasse besitzt eine *Writer*-Klasse. Die *Writer*-Klasse wird verwendet, um den neuen Repository-Inhalt zu schreiben, dabei wird die Spezialisierung *EAWriter* zum Schreiben eines EA-Repositorys verwendet und die Spezialisierung *JGraLabWriter* zum Schreiben eines JGraLab-Repositorys verwendet.
Reader-Klassen sind nicht nötig, da Leseoperationen durch einen einzelnen Methodenaufruf auf das entsprechende *Repository*-Objekt realisiert werden können. Daher liest die jeweilige Spezialisierung der *Transformer*-Klasse den zu schreibenden Inhalt. So hat die Klasse *TransformerJGraLabToEA* einen Lesezugriff auf das JGraLab-Repository und die Klasse *TransformerEAToJGraLab* greift lesend auf das EA-Repository zu.
Eine *Transformer*-Komponente wird also durch eine *Transformer*-Klasse und ihre *Writer*-Klasse umgesetzt.

Die *GUI*-Komponente soll durch die Klasse *MainFrame* und ihre beiden Panels *ControlPanel* und *ConsolePanel* umgesetzt werden. Dabei fungiert die Klasse *MainFrame* als ein Hauptcontainer für die grafische Benutzungsoberfläche. Dieser Hauptcontainer enthält dann die beiden Panels. Wobei sich das *ControlPanel* um alle Steuerungselemente, wie zum Beispiel Buttons, kümmert und das *ConsolePanel* die Ausgabe von Meldungen und Ergebnissen umsetzt.

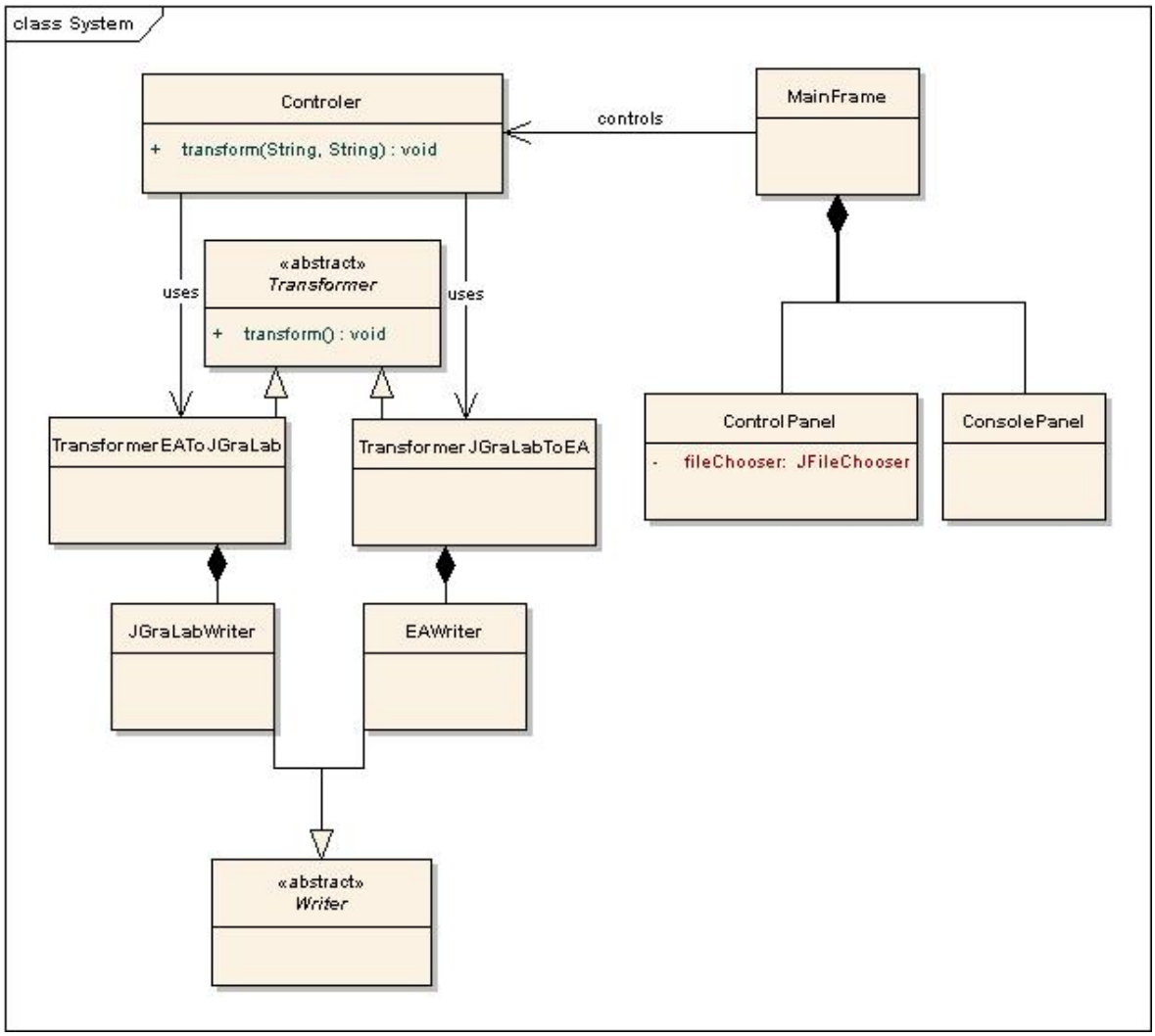

Abbildung 4.3: Klassendiagramm des Überführungstools

5 Entwicklung des Überführungstools

Nachdem in den bisherigen Kapiteln die Anforderungen, die Architektur und das nötige Fachwissen zum Enterprise Architect und zu JGraLab vorgestellt wurden, beschreibt dieses Kapitel die Entwicklung des Überführungstools.

5.1 Definition des Schemas für EA in JGraLab

Im Bezug auf die Hauptaufgabe der Studienarbeit, das Überführen von EA nach JGraLab, soll ein Schema definiert werden, das dem Metamodell des Enterprise Architects weitgehend entspricht. Da es nicht möglich ist über das EA-API auf das Metamodell des Enterprise Architects zuzugreifen und das vollständige UML-Metamodell [OMG05] zu mächtig ist, muss ein neues Schema erstellt werden, welches die Repository-Inhalte beschreiben kann. Zur Definition des Schemas wird die Modellierungssprache grUML [BRSS07] verwendet. Abb. 5.1 zeigt dieses Schema. Damit dieses Klassendiagramm nicht zu unübersichtlich wird, sind die Spezialisierungen der Klassen *Connector*, *Element* und *Diagram* nicht eingezeichnet. Diese fehlenden Unterklassen werden in den Tabellen 5.1, 5.2 und 5.3 aufgezählt.

Das Schema beschreibt zwei verschiedene Aspekte, zum Einen den Inhalt und die Struktur der Pakete, Modelle, Elemente und Diagramme des Repositorys und zum Anderen die Beziehungen zwischen den Elementen.

Die Beschreibung der Beziehungen zwischen den Elementen geschieht mit Hilfe der Klasse *Connector* und deren Spezialisierungen. Eine *Connector*-Instanz verbindet zwei nicht notwendigerweise verschiedene *Element*-Instanzen. Da es innerhalb des EA-Repositorys verschiedene Typen für solche Beziehungen gibt, muss es im Schema für jeden Typ eine Spezialisierung der Klasse *Connector* geben. Der Klassenname einer *Connector*-Spezialisierung setzt sich aus dem Präfix „C“ und der Typ-Bezeichnung aus dem Enterprise Architect zusammen. In Tabelle 5.1 werden alle möglichen Beziehungs-Typen gelistet.

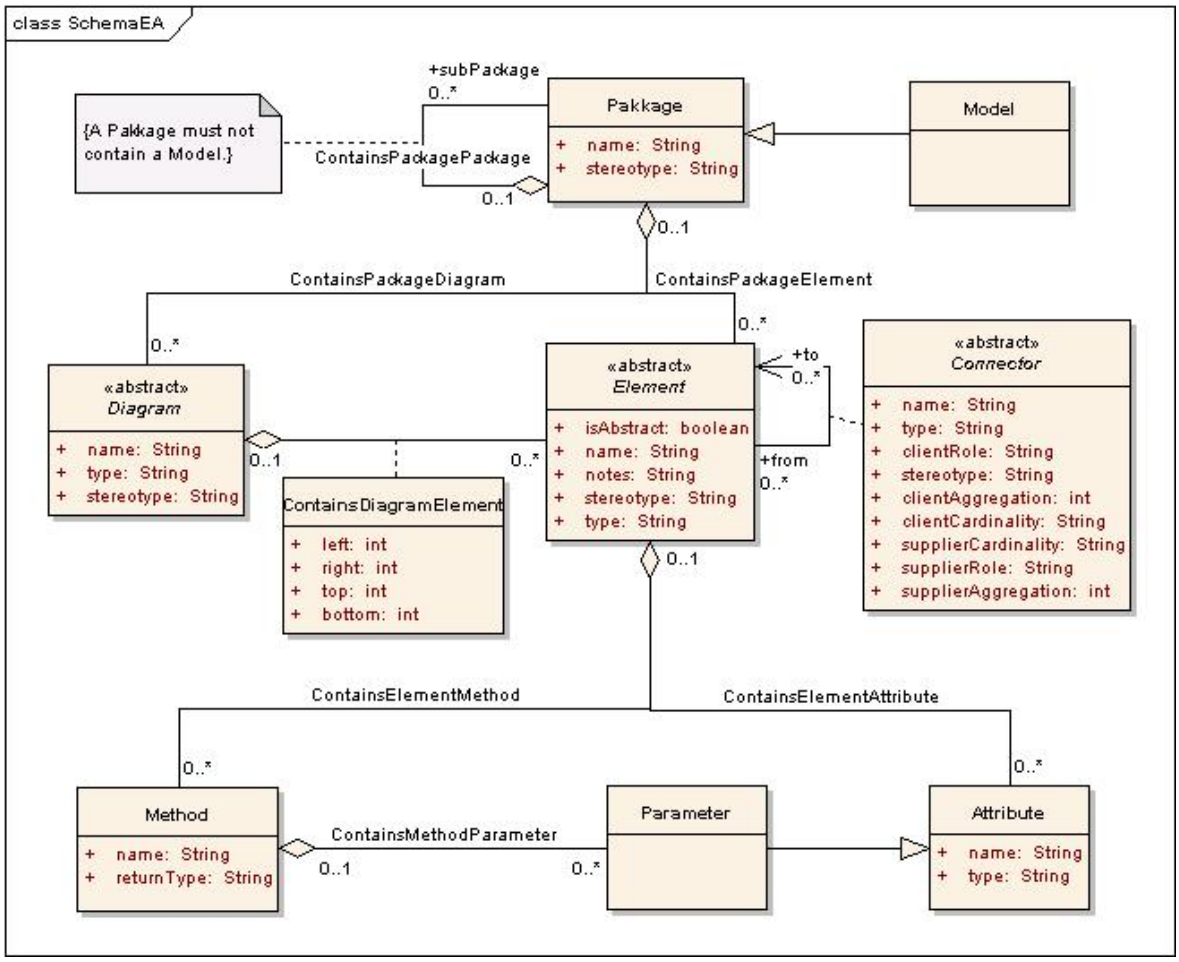

Abbildung 5.1: JGraLab-Schema für EA nach der grUML-Notation [BRSS07]

Connector-Typen.

Aggregation	Assembly	Association
Collaboration	ControlFlow	Connector
Delegate	Dependency	Deployment
ERLink	ForeignKey	Generalization
Instantiation	Interface	InterruptFlow
Manifest	Nesting	NoteLink
ObjectFlow	Package	Realisation
Sequence	StateFlow	UseCase

Tabelle 5.1: Beziehungs-Typen [Spa06b]

Element-Typen.		
Action	Activity	ActivityPartition
ActivityRegion	Actor	Artifact
Association	Boundary	Change
Class	Collaboration	Component
Constraint	Decision	DeploymentSpecification
DiagramFrame	EmbeddedElement	Entity
EntryPoint	Event	ExceptionHandler
ExitPoint	ExpansionNode	ExpansionRegion
GUIElement	InteractionFragment	InteractionOccurrence
InteractionState	Interface	InterruptibleActivityRegion
Issue	Node	Note
Object	Package	Parameter
Part	Port	ProvidedInterface
Report	RequiredInterface	Requirement
Screen	Sequence	State
StateNode	Synchronization	Text
TimeLine	UMLDiagram	UseCase
MessageEndpoint[a]		

[a] MessageEndpoint kann als Typ für ein Element verwendet werden, ist jedoch nicht in [Spa06b] gelistet.

Tabelle 5.2: Element-Typen [Spa06b]

Die Klasse *Element* besitzt innerhalb des JGraLab-Schemas ebenfalls Unterklassen. Diese Unterklassen werden zur Beschreibung der verschiedenen Element-Typen innerhalb des EA-Repositorys genutzt. Der Klassenname einer solchen *Element*-Spezialisierung wird analog zum Klassennamen einer *Connector*-Spezialisierung zusammengesetzt, das heißt, dass das Präfix „E“ und die Typ-Bezeichnung des Enterprise Architects den Klassennamen bilden. Alle Element-Typen werden in Tabelle 5.2 gezeigt.

Wenn man innerhalb des Enterprise Architects zum Beispiel eine Klasse modellieren möchte, so verwendet man ein Element und benötigt eventuell zusätzliche Methoden und Attribute. Attribute und Methoden werden durch die Klassen *Attribute* und *Method* innerhalb des Graphen dargestellt. Das bedeutet, dass jedes Element innerhalb des Enterprise Architects Attribute und Methoden besitzen kann, obwohl es beispielsweise keinen Sinn macht, dass ein Element vom Typ „Note“ Attribute oder Methoden hat. Methoden besitzen Parameter, die mit Hilfe der Klasse *Parameter* instanziert werden.

Diagram-Typen.		
Activity	Analysis	Component
Custom	Deployment	Logical
Sequence	Statechart	UseCase
Package[a]	Object[b]	

[a]Package kann als Typ für ein Diagramm verwendet werden, ist jedoch nicht in [Spa06b] gelistet.
[b]Object kann als Typ für ein Diagramm verwendet werden, ist jedoch nicht in [Spa06b] gelistet.

Tabelle 5.3: Diagramm-Typen [Spa06b]

Zur Beschreibung des Repository-Inhaltes gibt es im Schema aus Abb. 5.1 noch die Oberklasse *Diagram*, sowie die Spezialisierungen der Klasse *Diagram*, die Klasse *Package* und die Klasse *Model*.
Die Spezialisierungen der Klasse *Diagram* ergeben sich analog zu den Spezialisierungen der Klassen *Element* und *Connector*, so dass das Präfix „D“ und die Typ-Bezeichnung den Klassennamen ergeben. Tabelle 5.3 enthält alle möglichen Diagramm-Typen des Enterprise Architects.

Um die Ablage-Struktur des Repository-Inhaltes wiederzugeben, gibt es die *Contains*-Kantentypen. Diese beschreiben, welche Klassen als Container für die anderen fungieren können. Insgesamt gibt es sieben *Contains*-Kantentypen:

- *ContainsPackagePackage* beschreibt, dass sich ein Paket in einem anderen Paket befindet. Mit der Ausnahme, dass ein Paket niemals ein Modell enthalten darf.
- *ContainsPackageElement* beschreibt, dass sich ein Element in einem Paket befindet.
- *ContainsPackageDiagram* beschreibt, dass sich ein Diagramm in einem Paket befindet.
- *ContainsDiagramElement* beschreibt, dass sich ein Element in einem Diagramm befindet.
- *ContainsElementAttribute* beschreibt, dass ein Attribute zu einem Element gehört.
- *ContainsElementMethod* beschreibt, dass eine Methode zu einem Element gehört.
- *ContainsMethodParameter* beschreibt, dass ein Parameter zu einer Methode gehört.

5.2 Komponenten des Überführungstools

In diesem Abschnitt sollen alle Komponenten des Überführungstools und ihre Zusammenhänge dargestellt werden. Das Komponentendiagramm des Überführungstools ist in Abb. 5.2 zu sehen.

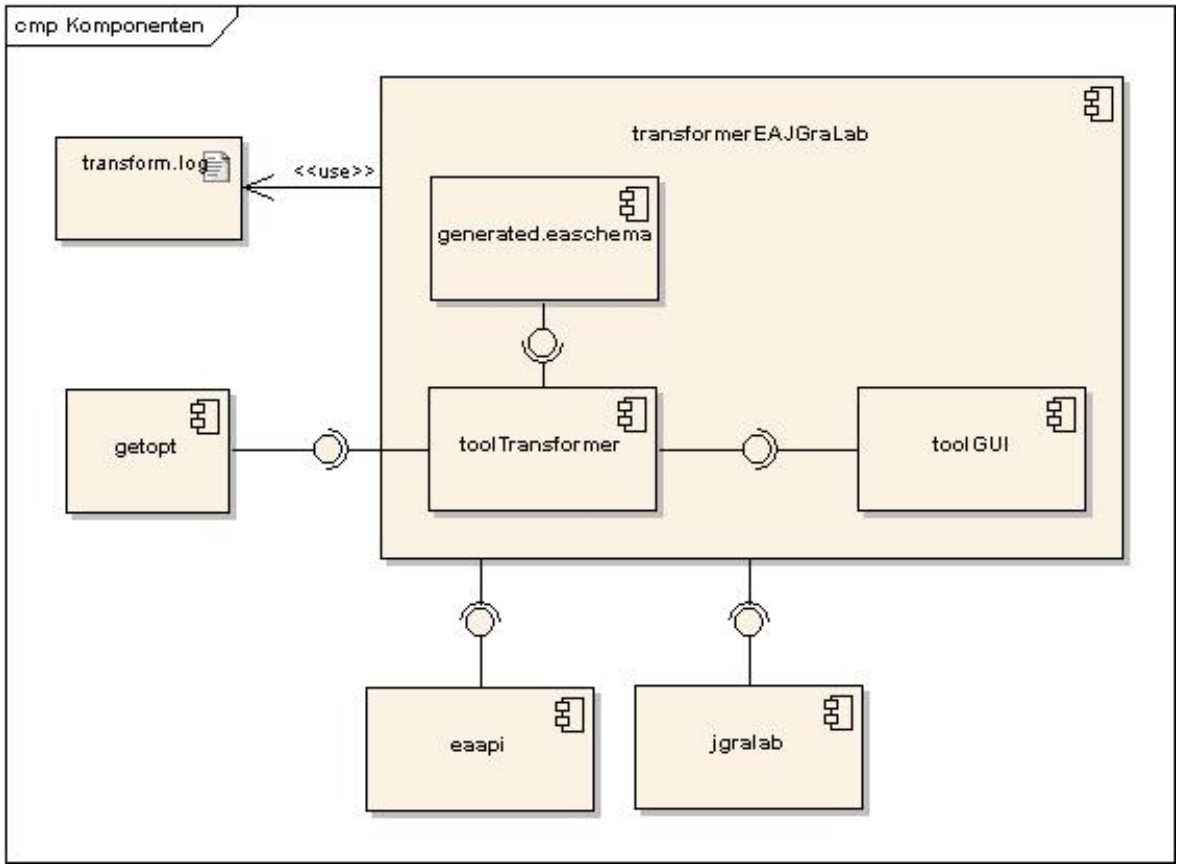

Abbildung 5.2: Komponentendiagramm des Überführungstools

Die Komponente Überführungstool bzw. „transformerEAJGraLab“ besteht aus drei internen und drei externen Komponenten. Die internen Komponenten wurden während der Implementierung des Überführungstools geschrieben bzw. generiert. Die externen Komponenten sind fertige Bibliotheken, deren Dienste das Überführungstool benötigt.

Interne Komponenten. Die internen Komponenten können auch als Quellcode, der in drei verschiedenen Paketen zusammengefasst wird, betrachtet werden. Das erste Paket „toolTransformer“ enthält den Quellcode, der die Überführung durchführt. Das zweite Paket „toolGUI“ beinhaltet den Quellcode für die grafische Benutzungsoberfläche. Im Unterschied zu den beiden ersten Paketen ist der Quellcode des dritten Paketes „generated.easchema“ generierter Code, der das EA-Schema aus Abschnitt 5.1 repräsentiert.

Externe Komponenten. Damit das Überführungstool EA-Projekte und TGraphen lesen bzw. schreiben kann, benötigt es Zugriff auf das EA-API und auf JGraLab. Dazu müssen JGraLab und das EA-API als externe Bibliotheken eingebunden werden. Dies ist in Abb. 5.2 durch die Komponenten „eaapi“ und „jgralab“ dargestellt.
Außerdem gibt es noch die externe Komponente „getopt“. Diese wird benötigt, um das Überführungstools mittels Optionen per Konsole zu benutzen, siehe dazu Abschnitt 5.8.

Wenn das Überführungstool arbeitet, protokolliert es sämtliche Schritte in einer *log*-Datei, sofern das Logging nicht deaktiviert wurde. Die Logging-Nachrichten werden in der Datei „transform.log“ gespeichert, sie ist in Abb. 5.2 als Artefakt dargestellt.

5.3 Klassen des Überführungstools

In diesem Abschnitt werden die Klassen beschrieben, die während der Entwicklung des Überführungstools programmiert wurden. Zur Beschreibung gehört die Bedeutung der einzelnen Klasse im Gesamtsystem und grobe Erklärungen zu den Methoden und Attributen der Klassen.
Zur Veranschaulichung zeigt Abb. 5.3 diese Klassen mit den wesentlichen Methoden und Attributen.

Controller. Die Klasse *Controller* ist die Startklasse des Überführungstools, wenn es nicht über die grafische Benutzungsoberfläche verwendet wird. Die *Controller*-Instanz startet die eigentliche Transformation und entscheidet je nach Aufruf-Parameter, welche Funktion des Überführungstools ausgeführt werden soll.
Die *Controller*-Instanz kann durch ihre eigene *main*-Methode erzeugt werden, durch die grafische Benutzungsoberfläche oder durch einen API-Zugriff eines externen Programms.
Falls man die *main*-Methode verwendet, so muss man darauf achten, dass die Parameter den Optionen zur Steuerung des Überführungstools aus Abschnitt 5.8 entsprechen. Die Methode *processArguments* bestimmt anhand der Argumente der *main*-Methode die Optionen zur Programmausführung.
Zum Starten über die grafische Benutzungsoberfläche gibt es den public Konstruktor *Controller* und die Methode *transform*, die zwei Parameter besitzt. Diese Parameter sind Pfadangaben zur Quell- und Zieldatei der Überführung. Dabei repräsentiert der erste Parameter die Quell- und der zweite die Zieldatei. Es muss beachtet werden, dass die Parameter genau eine *tg*- und genau eine *eap*-Datei referenzieren.
Bei einem API-Zugriff auf das Überführungstool wird die selbe Methode und der selbe Konstruktor verwendet, die beide auch von der grafischen Benutzungsoberfläche verwendet wer-

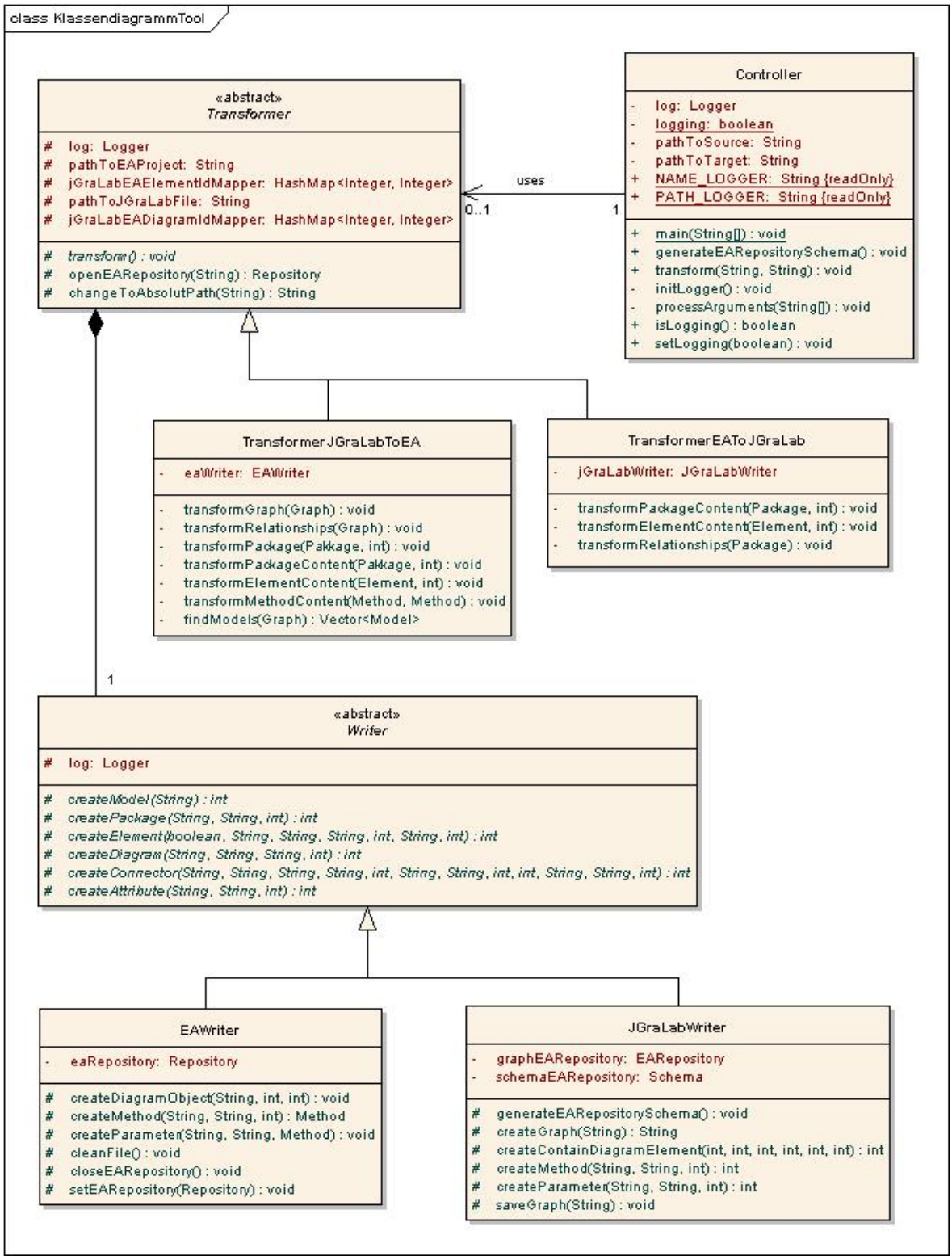

Abbildung 5.3: Klassendiagramm des Überführungstools

den. Das hängt damit zusammen, dass die grafische Benutzungsoberfläche einen API-Zugriff auf das Überführungstool macht und somit auch als externes Programm betrachtet werden kann.
Um ein EA-Schema mit JGraLab zu generieren, gibt es die Methode *generateEARepositorySchema*. Des weiteren wird in der *Controller*-Klasse der Logger erzeugt, mit dessen Hilfe anschließend alle relevanten Schritte protokolliert werden, wenn das Attribut *logging* den Wert „true“ besitzt. Das private Attribut *logging* kann durch die Methode *isLogging* abgefragt und durch *setLogging* gesetzt werden. Der verwendete Logger ist ein Singleton [GHJV95]. Damit die anderen Klassen die selbe *Logger*-Instanz nutzen können, gibt es die public Konstanten *NAME_LOGGER* und *PATH_LOGGER*.

Transformer. In einer *Transformer*-Klasse wird der Algorithmus einer Überführung festgelegt. *Transformer* ist die abstrakte Oberklasse für die Spezialisierungen *TransformerEAToJGraLab* und *TransformerJGraLabToEA*. Die *Transformer*-Klasse zieht in erster Linie die gemeinsamen Komponenten aus den Spezialisierungen heraus. Die Klasse *Transformer* besitzt eine Referenz auf den vom *Controller* instanzierten *Logger* und die Attribute *pathToEAProject*, *pathToJGraLabFile*, *jGraLabEAElementIdMapper* und *jGraLabEADiagramIdMapper*.
Diese Attribute sowie die Methoden *createEARepository* und *changeToAbsolutPath* werden somit an alle Spezialisierungen von *Transformer* vererbt. Die Methode *createEARepository* erzeugt eine *Repository*-Instanz des Enterprise Architects und die Methode *changeToAbsolutPath* wandelt relative in absolute Pfadangaben um, da das EA-API nur mit absoluten Pfadangaben umgehen kann.
Außerdem legt die *Transformer*-Oberklasse fest, dass alle Unterklassen die Methode *transform* implementieren müssen, diese führt dann die Überführung aus, die zur Überführungsrichtung der jeweiligen Unterklasse passt.
Die Attribute *pathToEAProject* und *pathToJGraLabFile* beinhalten die Pfade zu den Dateien, deren Inhalte gelesen bzw. geschrieben werden sollen. *jGraLabEAElementIdMapper* und *jGraLabEADiagramIdMapper* werden für die Überführung der Inhalte der Diagramme und für die Beziehungen zwischen den Elementen gebraucht.

TransformerEAToJGraLab. Wenn der Repository-Inhalt des Enterprise Architects in eine *tg*-Datei überführt werden soll, so wird eine *TransformerEAToJGraLab*-Instanz verwendet.
Zum Überführen läuft eine Schleife über alle Modelle innerhalb der *eap*-Datei, dabei werden die Modelle der Reihe nach überführt. Dies geschieht durch die Methode *transform*. Die Überführung eines Modells geschieht in zwei Schritten.

1. Im ersten Schritt werden die Pakete und deren Inhalte, also Elemente und Diagramme, durch den rekursiven Aufruf der Methode *transformPackageContent* in einen Graphen geschrieben. Die Methode ruft sich rekursiv auf, um die Paketstruktur zu durchlaufen. Damit die Elemente und deren Inhalte, also Attribute und Methoden, in den Graphen geschrieben werden, gibt es die Methode *transformElementContent*. Das Schreiben in den Graphen geschieht durch die entsprechenden Methoden der *JGraLabWriter*-Instanz.

2. Nach dem ersten Schritt befinden sich Pakete, Diagramme und Elemente in entsprechender Struktur im Graphen. Der zweite Schritt erzeugt die Beziehungen zwischen den Elementen und legt fest, welche Elemente in welchem Diagramm dargestellt werden sollen. Dazu wird die Methode *transformRelationships* verwendet. Für jedes Element werden die ausgehenden Beziehungen durch *Connector*-Kanten in den Graphen geschrieben. Dabei gilt eine Beziehung als von einem Element ausgehend, wenn sich dieses Element in der Client-Rolle der Beziehung befindet. Außerdem werden für jedes Diagramm *ContainsDiagramElement*-Kanten zu den Elementen innerhalb dieses Diagramms erzeugt.

Die beiden geerbten Attribute *jGraLabEAElementIdMapper* und *jGraLabEADiagramIdMapper* werden benötigt, um die ID eines JGraLab-Elements auf die eines EA-Elements bzw. die ID eines JGraLab-Diagramms auf die eines EA-Diagramms abzubilden. Diese Abbildungen werden im ersten Schritt erzeugt und im zweiten Schritt benötigt, um die der ID entsprechenden Knoten zu finden.

TransformerJGraLabToEA. Zur Überführung eines TGraphen in ein EA-Projekt wird die Klasse *TransformerJGraLabToEA* benutzt.
Da ein TGraph mehrere Modelle enthalten kann, werden zuerst alle Knoten, die ein Modell repräsentieren, durch die Methode *findModels* gesucht. Die Überführung der einzelnen Modelle in ein EA-Repository geschieht analog zur Klasse *TransformerEAToJGraLab* in zwei Schritten.

1. Im ersten Schritt werden Pakete, Diagramme und Elemente übertragen. Dazu läuft die Methode *transformPackage* ebenfalls rekursiv durch die Paketstruktur und überträgt Pakete und deren Inhalte. Außerdem werden die für den zweiten Schritt benötigten Abbildungen in die geerbten Attribute *jGraLabEAElementIdMapper* und *jGraLabEADiagramIdMapper* eingetragen.

2. Im zweiten Schritt werden dann die *Connector*-Kanten durch EA *Connector*-Instanzen und die *ContainsDiagramElement*-Kanten durch *DiagramObject*-Instanzen im EA-Repository repräsentiert. Dies geschieht durch die Methode *transformRelationships*.

Zum Schreiben in das EA-Repository besitzt die Klasse *TransformerJGraLabToEA* eine Instanz der Klasse *EAWriter*.

Writer. Die abstrakte Klasse *Writer* dient als Oberklasse für die Klassen *EAWriter* und *JGraLabWriter*. Sie vererbt eine Referenz des Loggers an ihre Unterklassen und besitzt abstrakte Methoden, die von den Unterklassen implementiert werden müssen. Diese abstrakten Methoden sind „create"-Methoden, die verwendet werden, um Modelle, Pakete, Diagramme, Elemente, Attribute und Beziehungen in das EA-Repository bzw. den TGraphen zu schreiben.
Da aber auch noch Graphen, Diagramm-Objekte, Contains-Beziehungen, Methoden und Parameter in TGraphen bzw. in das EA-Repository geschrieben werden müssen, ist es notwendig, dass die Spezialisierungen der *Writer*-Klasse entsprechende „create"-Methoden besitzen. Diese sind aufgrund der Unterschiede zwischen EA und JGraLab nicht in der *Writer*-Klasse definiert.

JGraLabWriter. *JGraLabWriter* ist eine Spezialisierung der Klasse *Writer*. Sie wird verwendet, um Knoten und Kanten in den TGraphen zu schreiben. Da sie in einen TGraphen schreibt, wird sie von der *Transformer*-Spezialisierung *TransformerEAToJGraLab* verwendet.
Diese Klasse besitzt die Attribute *schemaEARepository* und *graphEARepository*. Dabei ist *schemaEARepository* das Schema, nach dem ein TGraph erzeugt werden kann. Um ein neues Schema zu generieren, gibt es die Methode *generateEARepositorySchema*, die zur Schemagenerierung die Methoden der JGraLab Klassen *Schema* und *GraphClass* verwendet, genaueres hierzu findet der Leser in [SBR06]. Der TGraph, in den geschrieben werden soll, wird durch *graphEARepository* repräsentiert.
Zum Schreiben gibt es zum einen die Implementierungen der geerbten „create"-Methoden und zum anderen „create"-Methoden, die bedingt durch die Unterschiede von JGraLab zu dem Enterprise Architect nicht geerbt werden können.
Die Methode *saveGraph* speichert den TGraphen in eine *tg*-Datei.

EAWriter. Die Klasse *EAWriter* erbt ebenfalls von der Klasse *Writer*. *EAWriter* wird verwendet, um den Repository-Inhalt zu schreiben. Methoden, die in das EA-Repository schreiben, beginnen analog zu den Methoden der anderen *Writer*-Klassen mit „create". Es werden also zum einen die geerbten „create"-Methoden implementiert und zum anderen eigene „create"-Methoden bereit gestellt.
Das Attribut *eaRepository* ist eine Referenz auf das EA-Repository, in das geschrieben werden soll. Mittels *setEARepository* wird diese Referenz gesetzt.

Zum Löschen aller Modelle im referenzierten EA-Repository gibt es die Methode *cleanFile*. Nachdem alle Schreiboperationen innerhalb des Repositorys ausgeführt wurden, kann mittels *closeEARepository* die *eap*-Datei, die den Repository-Inhalt festlegt, geschlossen werden.

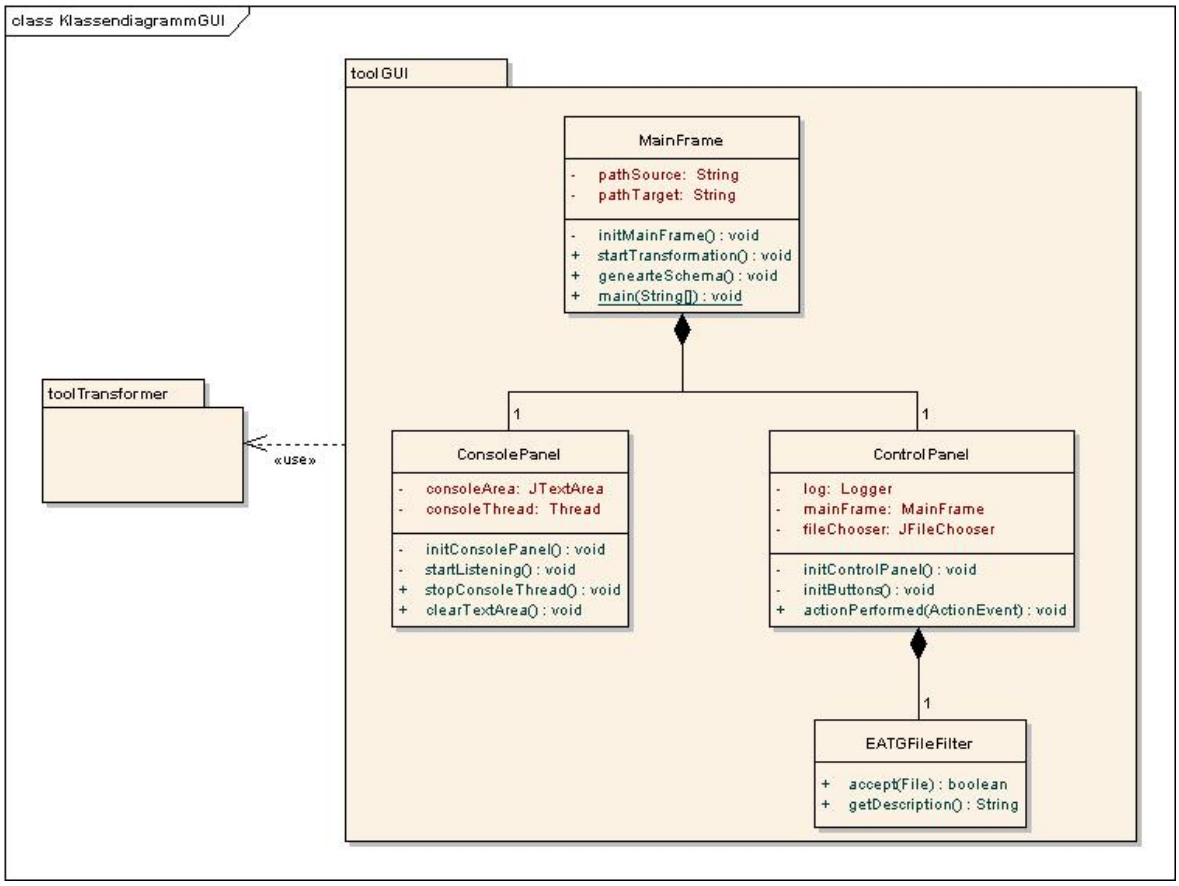

Abbildung 5.4: Klassendiagramm der GUI des Überführungstools

Damit das Überführungstool nicht nur mit der Konsole bedient werden kann, gibt es auch noch eine grafische Benutzungsoberfläche. Diese besteht aus den vier im Folgenden beschriebenen Klassen. Abb. 5.4 zeigt diese vier Klassen und das Paket *toolGUI* in dem sie sich befinden. Außerdem gibt es noch das Paket *toolTransformer*, dessen Inhalt entspricht den Klassen aus Abb. 5.3.

MainFrame. Das Hauptfenster wird durch die Klasse *MainFrame* erzeugt. Diese Klasse ist bedingt durch die Methode *main* auch eine Startklasse und wird verwendet, wenn man das Überführungstool mittels grafischer Benutzungsoberfläche bedienen möchte.
Die Inhalte und Eigenschaften des Hauptfensters werden durch die Methode *initMainFrame* festgelegt. Diese wird durch den Konstruktoraufruf der Klasse *MainFrame* aufgerufen. Das

Hauptfenster beinhaltet die beiden Panels *ConsolePanel* und *ControlPanel*.
Die beiden Attribute *pathSource* und *pathTarget* beinhalten als String die Pfade zur Quell- bzw. zur Zieldatei der Überführung. Um die Überführung zu starten, gibt es die Methode *startTransformation*. Des weiteren gibt es die Methode *generateSchema*, um das EA Schema in JGraLab zu generieren.

ConsolePanel. Das *ConsolePanel* ist Teil des Hauptfensters und hat die Aufgabe dem Benutzer alle Meldungen anzuzeigen, die auch in der *log*-Datei protokolliert werden.
Dieses Panel wird von der Methode *initConsolePanel* initialisiert, dabei werden in erster Linie die nötigen Einstellungen für die Textarea *consoleArea* vorgenommen. Nach der Initialisierung wird der Thread *consoleThread* durch die Methode *startListening* gestartet. Dessen Aufgabe ist: neue Inhalte der *log*-Datei feststellen und in *consoleArea* anzeigen. Dieser Thread kann durch die public Methode *stopConsoleThread* beendet werden. Um den Text in der Textarea zu löschen, gibt es die Methode *clearTextArea*.

ControlPanel. Um das Überführungstool zu benutzen, bietet das *ControlPanel* dem Benutzer sieben Buttons. Damit das *ControlPanel* das Anklicken der Buttons registriert, implementiert es das Interface *ActionListener*. Die Methode *actionPerformed* behandelt anschließend den *ActionEvent*, der durch das Anklicken eines Buttons ausgelöst wurde.
Die Buttons werden durch die Methode *initButtons* initialisiert, während der Rest des Panels durch die Methode *initControlPanel* initialisiert wird. Weitere Informationen zu den Buttons bzw. den Steuerungsmöglichkeiten der grafischen Benutzungsoberfläche befinden sich in Abschnitt 5.6.
Das *ControlPanel* besitzt eine Referenz auf das *MainFrame*, so dass es über dieses die Benutzereingaben an das Überführungstool weiterleiten kann. Zum Auswählen von Dateien, die später als Quell- und Zieldatei der Überführung fungieren sollen, hat das *ControlPanel* eine *JFileChooser*-Instanz. Außerdem gibt es eine Referenz auf den Logger, um zu protokollieren welche Dateien ausgewählt wurden.

EAPTGFileFilter. Die Klasse *EAPTGFileFilter* wird verwendet, um der *JFileChooser*-Instanz des *ControlPanels* mitzuteilen, dass nur Dateien vom Typ *tg* oder *eap* angezeigt und ausgewählt werden dürfen.

5.4 Ausnahmebehandlung des Überführungstools

Dieses Kapitel beschäftigt sich mit den Exceptions, die durch das Überführungstool verursacht werden können. Abb. 5.5 zeigt die Hierarchie der durch das Überführungstool mög-

lichen Exceptions.

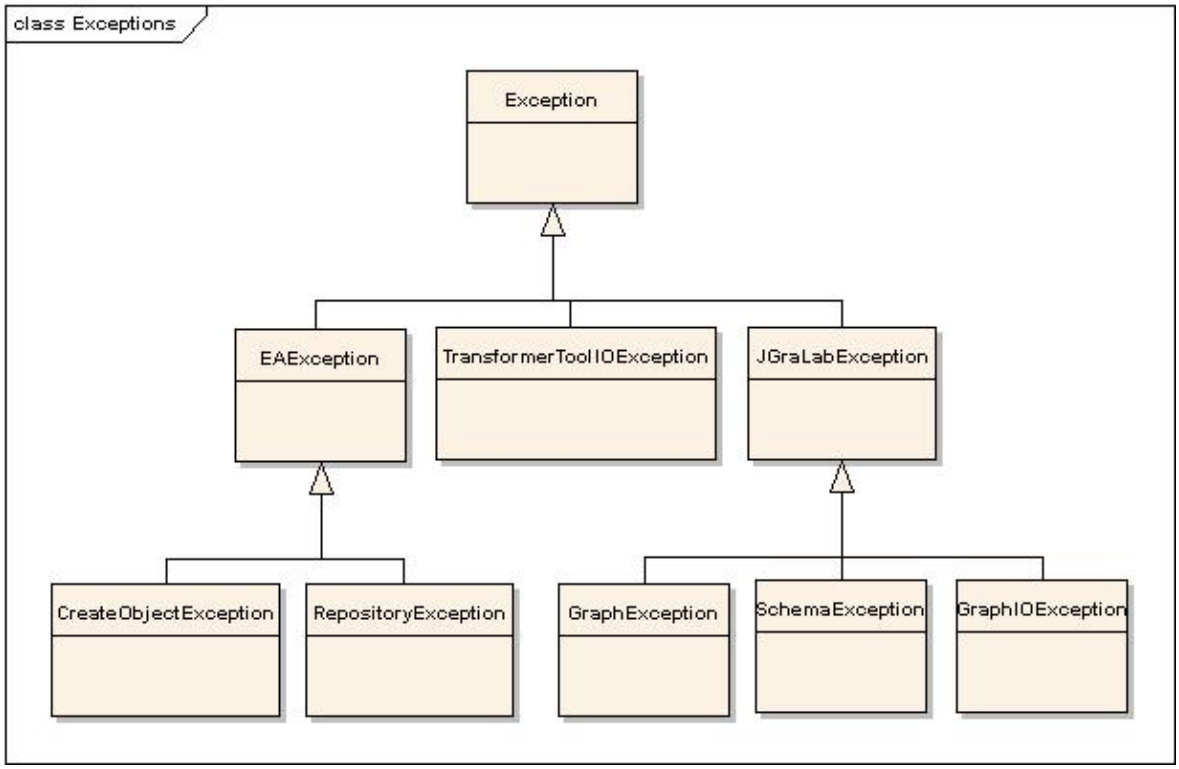

Abbildung 5.5: Hierarchie der Exceptions des Überführungstools

***Exception*-Klassen.** Es gibt drei Ebenen innerhalb der Exception-Hierarchie. Die erste Ebene ist die Oberklasse Java-*Exception* von ihr erben die *Exception*-Klassen *EAException*, *TransformerToolIOException* und *JGraLabException* [Kah06]. Die zweite Ebene beschreibt den Ursprung der Ausnahme.

Das Überführungstool verwendet zwei externe APIs, zum einen das EA-API und zum anderen das JGraLab-API. Beide können Ursprung von verschiedenen Ausnahmen sein. Wenn der Ursprung direkt von der EA-API abstammt, so wird eine *EAException* geworfen. Das EA-API wirft von sich aus jedoch keine Exceptions, sondern liefert Rückgabewerte wie „null“ oder „false“, um Ausnahmen zu signalisieren. Analog dazu wird eine *JGraLabException* geworfen, wenn sich der Ursprung in JGraLab befindet. Das Überführungstool selbst kann auch Ausnahmen produzieren, diese werden durch die Klasse *TransformerToolIOException* beschrieben.

In der dritten Ebene wird der Auslöser der Ausnahme noch genauer beschrieben. So gibt es *CreateObjectException* und *RepositoryException*, um Ausnahmen, die von der EA-API stammen, genauer zu beschreiben. Zur genaueren Beschreibung der Ausnahmen in JGraLab gibt es *GraphException*, *GraphIOException* und *SchemaException*. Weiterführende Infor-

mationen zu den Auslösern dieser Exceptions findet man in [Kah06].

Eine *TransformerToolIOException* kann beim Start der Überführung durch falsche Parameter, die nicht den Pfad zu genau einer *eap*- und genau einer *tg*-Datei beschreiben, ausgelöst werden.
CreateObjectException werden ausgelöst, wenn das Erzeugen eines Objektes im EA-Repository fehlschlug. Objekte, die ins EA-Repository geschrieben werden können, sind Modelle, Pakete, Elemente, Diagramme, Diagramm-Objekte, Attribute, Methoden, Parameter und Beziehungen. Diese EA-Objekte besitzen eine *Update*-Methode, die zum Schreiben in das Repository benutzt wird. Wenn diese *Update*-Methode „false“ liefert, wird vom Überführungstool eine *CreateObjectException* geworfen.
Falls es beim Öffnen des EA-Repositorys zu Ausnahmen kommt, so wirft das Überführungstool eine *RepositoryException*. Diese Ausnahmen können beim Öffnen einer *eap*-Datei oder beim Anlegen einer *Repository*-Instanz entstehen.

5.5 Installation des Überführungstools

Zur Installation des Überführungstools benötigt man das Überführungstool selbst. Dieses befindet sich als *jar*-Datei *transformerEAJGraLab.jar* und als Quellcode auf der CD im Anhang der Studienarbeit.

Das Überführungstool benötigt die *jar*-Dateien *getopt.jar* und *jgralab.jar* beide müssen sich im Java-Classpath befinden. Diese beiden Dateien befinden sich ebenfalls auf der CD im Anhang der Studienarbeit.

Damit das Überführungstool das EA-API verwenden kann, muss die Datei *SSJavaCOM.dll* vorhanden sein und sich im Windows Pfad befinden. Außerdem muss sich das EA-API selbst als *jar*-Datei *eaapi.jar* im Java-Classpath befinden.
Diese beiden Dateien befinden sich aus lizenzrechtlichen Gründen nicht auf der CD im Anhang. Man erhält sie aber mit der **Corporate Edition** des Enterprise Architects oder mit der Demoversion der **Corporate Edition**.

Um das Überführungstool zu starten, muss Java 5 installiert sein.

5.6 Die Benutzungsoberfläche des Überführungstools

Zur Überführung von EA nach JGraLab und umgekehrt besitzt das Überführungstool eine grafische Benutzungsoberfläche, die dem Benutzer die wesentlichen Funktionen grafisch anbietet. Die Benutzungsoberfläche besteht aus einem Fenster, das sechs Buttons und eine Textarea enthält. Die Buttons werden zur Steuerung der Funktionalität verwendet, während die Textarea lediglich die Meldungen aus der *log*-Datei anzeigt. Ein Screenshot der grafischen Benutzungsoberfläche des Überführungstools ist in Abb. 5.6 zu sehen.

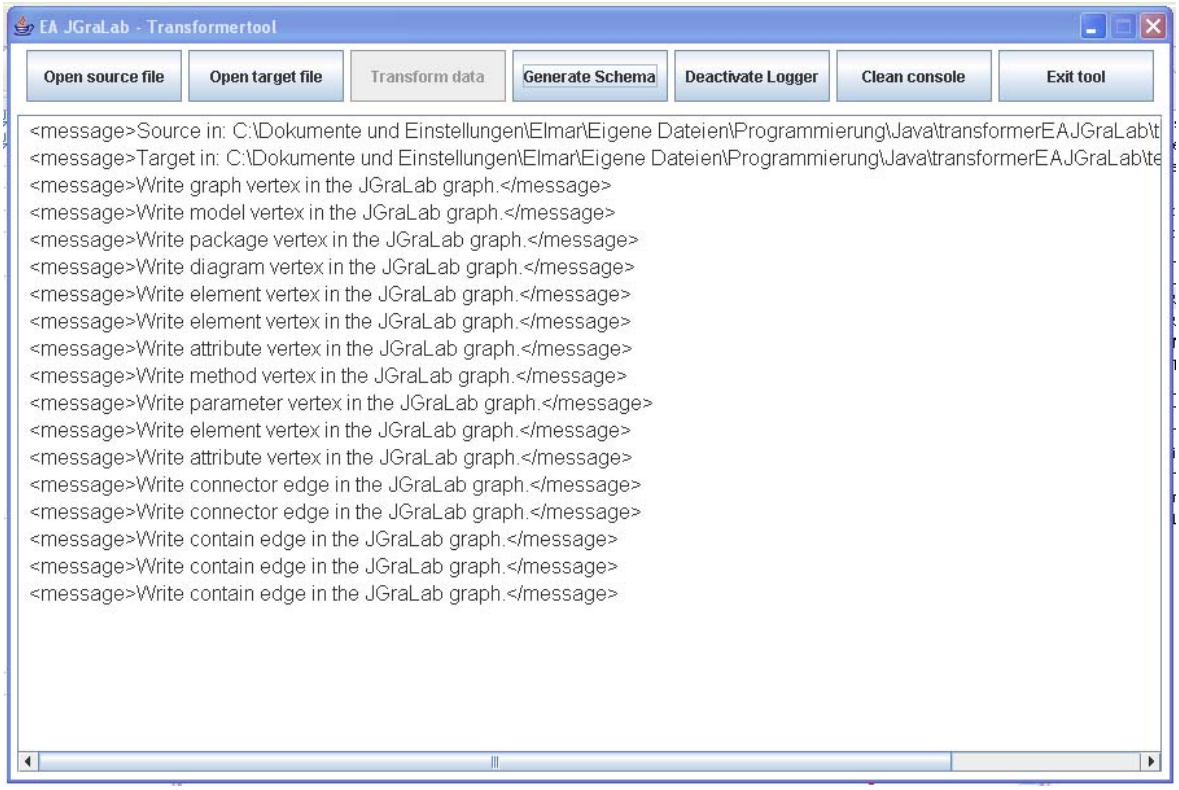

Abbildung 5.6: Die grafische Benutzungsoberfläche des Überführungstools

„Open source file“ / „Open target file“. Um eine Überführung durchführen zu können, müssen zuerst die Quell- und die Ziel-Datei ausgewählt werden. Dazu gibt es die beiden Buttons „Open source file“ und „Open target file“, beide starten einen Dateibrowser mit dem nur *tg*- und *eap*-Dateien geöffnet bzw. ausgewählt werden können. Mit „Open source file“ wird die Quell-Datei und mit „Open target file“ die Ziel-Datei festgelegt. Dabei sollte man beachten, dass Quell- und Ziel-Datei nicht vom gleichen Typ sind.

„Transform data“. Zum Starten der Überführung gibt es den „Transform data“-Button, dieser ist jedoch nur aktiv, wenn vorher Quell- und Ziel-Datei ausgewählt wurden. Nachdem er einmal benutzt wurde, bleibt er bis zum Beenden des Programms deaktiviert, weil das Überführungstool aufgrund einer Komplikation im EA Repositorys immer nur eine *eap*-Datei verarbeiten kann.

„Generate Schema“. Falls Änderungen am EA Schema im Quellcode gemacht wurden und ein neues Schema generiert werden soll, so kann dazu der „Generate Schema“-Button verwendet werden. Dieser starten dann eine Methode, welche die neuen Klassen generiert und anschließend den „Transform data“-Button bis zum Beenden des Programms deaktiviert. Diese Deaktivierung muss gemacht werden, da das Generieren eines neuen Schemas den Quellcode für die Transformation ändern kann und somit eine sichere Funktion des Überführungstools nicht mehr gewährleistet werden kann.

„Activate Logger“ / „Deactivate Logger“. Standardmäßig ist die Logging-Funktion des Tools aktiviert. Wenn keine *log*-Datei erzeugt werden soll, so kann die Logging-Funktion mit dem Button „Deactivate Logger“ ausgeschaltet werden. Zum Aktivieren des Loggers verwendet man den Button „Activate Logger“. Man sollte allerdings beachten, dass bei deaktiviertem Logger keine Logging-Meldungen in der grafischen Benutzungsoberfläche angezeigt werden.

„Clean console“. Zum Löschen der Meldungen innerhalb der Textarea gibt es den Button „Clean console“. Da die Textarea nur Meldungen wiedergibt, die sich auch in der Konsole (und der *log*-Datei) befinden, wird sie auch als „console“ bezeichnet.

„Exit tool“. Das Beenden des Überführungstools muss mit dem „Exit tool“-Button gemacht werden, da nur so sichergestellt werden kann, dass alle parallel laufenden Threads ebenfalls beendet werden.

5.7 API des Überführungstools

Das Überführungstool bietet Programmierern ein API an. Über dieses API kann man alle Funktionen des Überführungstools nutzen. Es ist möglich:

- Den Repository-Inhalt des Enterprise Architects nach JGraLab überführen.
- Einen dem Schema aus Abschnitt 5.1 entsprechenden TGraphen in das EA-Repository schreiben.

- Das im Quellcode definierte EA-Schema generieren.
- Die Logging-Funktion aktivieren und deaktivieren.

transform(String, String): void. Um diese Funktionen nutzen zu können, wird eine Instanz der Klasse *Controller* benötigt. Zum Überführen in beide Richtungen bietet sie die Methode *transform*. Diese Methode hat zwei String-Parameter, die die Pfade zu einer *tg*- und einer *eap*-Datei beschreiben müssen. Dabei definiert der erste Parameter den Pfad zur Quelldatei und der zweite den Pfad zur Zieldatei. Durch die beiden Parameter wird also auch die Überführungsrichtung festgelegt.

generateEARepositorySchema(): void. Innerhalb der Klasse *JGraLabWriter* ist das EA-Schema definiert. Wenn man Java-Klassen anhand dieses Schemas generieren möchte, so verwendet man die Methode *generateEARepositorySchema* der Klasse *Controller*.

isLogging(): boolean* / *setLogging(boolean): void. Die beiden Methoden *isLogging* und *setLogging* der Klasse *Controller* werden verwendet um festzulegen, ob die ausgeführten Schritte innerhalb der Datei *transform.log* protokolliert werden sollen oder nicht. Mit *setLogging* kann das Logging mit dem Parameterwert „true“ aktiviert und mit „false“ deaktiviert werden. Die Methode *isLogging* gibt „true“ zurück, wenn das Logging aktiviert ist.

5.8 Steuerung des Überführungstools per Konsole

Optionen. Möchte man das Überführungstool per Konsole bedienen, so kann man mit Hilfe von Optionen alle Funktionen ausführen. Die Optionen des Überführungstools wurden mit „GNU getopt“ [GNU06] realisiert.
Listing 5.1 zeigt den regulären Ausdruck, der den Startbefehl des Überführungstools und die möglichen Kombinationen der Optionen definiert. Dabei ist die Reihenfolge der Optionen unerheblich.

Listing 5.1: Regulärer Ausdruck zur Verwendung des Überführungstools

```
java (-cp | --classpath) <classpath> Controller
  (((-s | --sourcefilepath) <path><.tg | .eap>
  (-t | --targetfilepath) <path><.tg | .eap>) |
  (-g | --generateschema))
  [(-n | --nologging)]);
```

- Die Option „sourcefilepath“ oder kurz „s“ wird verwendet, um den Pfad zur Quelldatei der Überführung zu setzen. Das Argument dieser Option ist der Pfad zur Quelldatei. Wenn diese Option gesetzt wird, so muss auch die Option zur Bestimmung der Zieldatei gesetzt werden und es darf kein neues EA-Schema generiert werden.
- Die Option „targetfilepath“ oder kurz „t“ wird verwendet, um den Pfad zur Zieldatei der Überführung zu setzen. Das Argument dieser Option ist der Pfad zur Zieldatei. Wenn diese Option gesetzt wird, so muss auch die Option zur Bestimmung der Quelldatei gesetzt werden und es darf kein neues EA-Schema generiert werden.
- Die Option „nologging“ oder kurz „n“ deaktiviert das Logging in der Datei *transform.log*. Standardmäßig ist das Loggen aktiviert. Diese Option besitzt keine Argumente.
- Die Option „generateschema“ oder kurz „g“ führt die Methode aus, die das EA-Schema generiert. Sie besitzt keine Argumente und darf nur gesetzt werden, wenn die Optionen zur Überführung von einer Quell- in eine Zieldatei nicht gesetzt sind.
- Die Option „help“ oder kurz „h“ zeigt den Hilfetext zur Verwendung des Überführungstools per Konsole an. Sie besitzt keine Argumente.

Classpath. Außer den Optionen zur Steuerung des Überführungstools muss noch der Java-Classpath gesetzt werden. Dieser muss die Pfadangaben zu den Dateien des EA-API, von JGraLab, von getopt und von dem Überführungstool selbst enthalten.

***main*-Methode.** Die zum Starten des Überführungstools relevante *main*-Methode befindet sich in der Klasse *Controller*, die innerhalb des Paketes *toolTransformer* ist.
Soll die grafische Benutzungsoberfläche des Überführungstools verwendet werden, so muss die *main*-Methode innerhalb der Klasse *MainFrame* aufgerufen werden. Die Klasse *MainFrame* befindet sich innerhalb des Paketes *toolGUI*. Die Optionen zum Steuern des Überführungstools werden in der *main*-Methode der Klasse *MainFrame* nicht verarbeitet.

Listing 5.2 zeigt eine Möglichkeit für den Befehl zur Überführung der Datei „earep.eap“ in die Datei „eagraph.tg“. In Listing 5.3 findet der Leser ein Beispiel für den Befehl zum Generieren des EA Schema mit deaktiviertem Logger.

Listing 5.2: Beispiel: Überführung einer *eap*-Datei in eine *tg*-Datei

```
java -cp ./bin;./lib/jgralab.jar;
  ./lib/eaapi.jar;./lib/getopt.jar
  toolTransformer.Controller
```

```
  -s  ./test/earep.eap -t  ./test/eagraph.tg
```

Listing 5.3: Beispiel: Generieren des EA Schemas mit deaktiviertem Logging

```
java -cp  ./bin;./lib/jgralab.jar;
  ./lib/eaapi.jar;./lib/getopt.jar
  toolTransformer.Controller -g -n
```

6 Evaluation

Dieses Kapitel befasst sich mit der Evaluation des Überführungstools. Zuerst soll an einem Beispiel demonstriert werden, wie das Überführungstool ein EA Projekt bestehend aus einem Aktivitätsdiagramm in einen TGraphen und diesen TGraphen wieder zurück in ein EA Projekt überführt.
Anschließend wird über die Erfahrungen beim Transformieren des ReDSeeDS [IST06] EA Projekts, einer ziemlich umfangreichen *eap*-Datei, berichtet.
Zuletzt sollen die Probleme, die noch bestehen, angesprochen und erläutert werden.

6.1 Beispiel

Dieser Abschnitt zeigt anhand eines Beispiels, wie eine Überführung eines EA Projektes nach JGraLab und die anschließende Rücktransformation aussieht.
Die als Quelle verwendete *eap*-Datei ist ein kleiner Ausschnitt des EA Projektes von ReDSeeDS. Diese Datei beinhaltet geschachtelte Pakete, die der Paketstruktur der ReDSeeDS *eap*-Datei entsprechen und ein Aktivitätsdiagramm, an dem die Überführung nachvollzogen werden soll.

Überführung EA nach JGraLab. Abb. 6.1zeigt das Aktivitätsdiagramm, das in einen TGraphen überführt werden soll. Zur Überführung übergibt man dem Überführungstool als Quelldatei die *eap*-Datei, die dieses Diagramm enthält. Als Zieldatei wird eine beliebige *tg*-Datei gewählt. Anschließend führt das Überführungstool die Transformation aus und erzeugt eine *tg*-Datei, deren Inhalt Listing 6.1 zeigt.
Die erzeugte *tg*-Datei beginnt mit einer Beschreibung des Schemas, dieses entspricht dem definierten EA-Schema aus Abschnitt 5.1. Nach der Schema Beschreibung wird der eigentliche Graph beschrieben, der der *eap*-Datei entspricht. Er besitzt 25 Knoten und 62 Kanten, dabei werden die 25 Knoten vor den 62 Kanten gelistet.
Innerhalb der *tg*-Datei werden alle Bestandteile (also Knoten, Kanten, Schema, Graph usw.) durch „;“ getrennt. Die Beschreibung eines Knoten oder einer Kante beginnt mit einer ID gefolgt von dem Typ. Die Knoten enthalten anschließend eine Liste der ein und ausgehenden

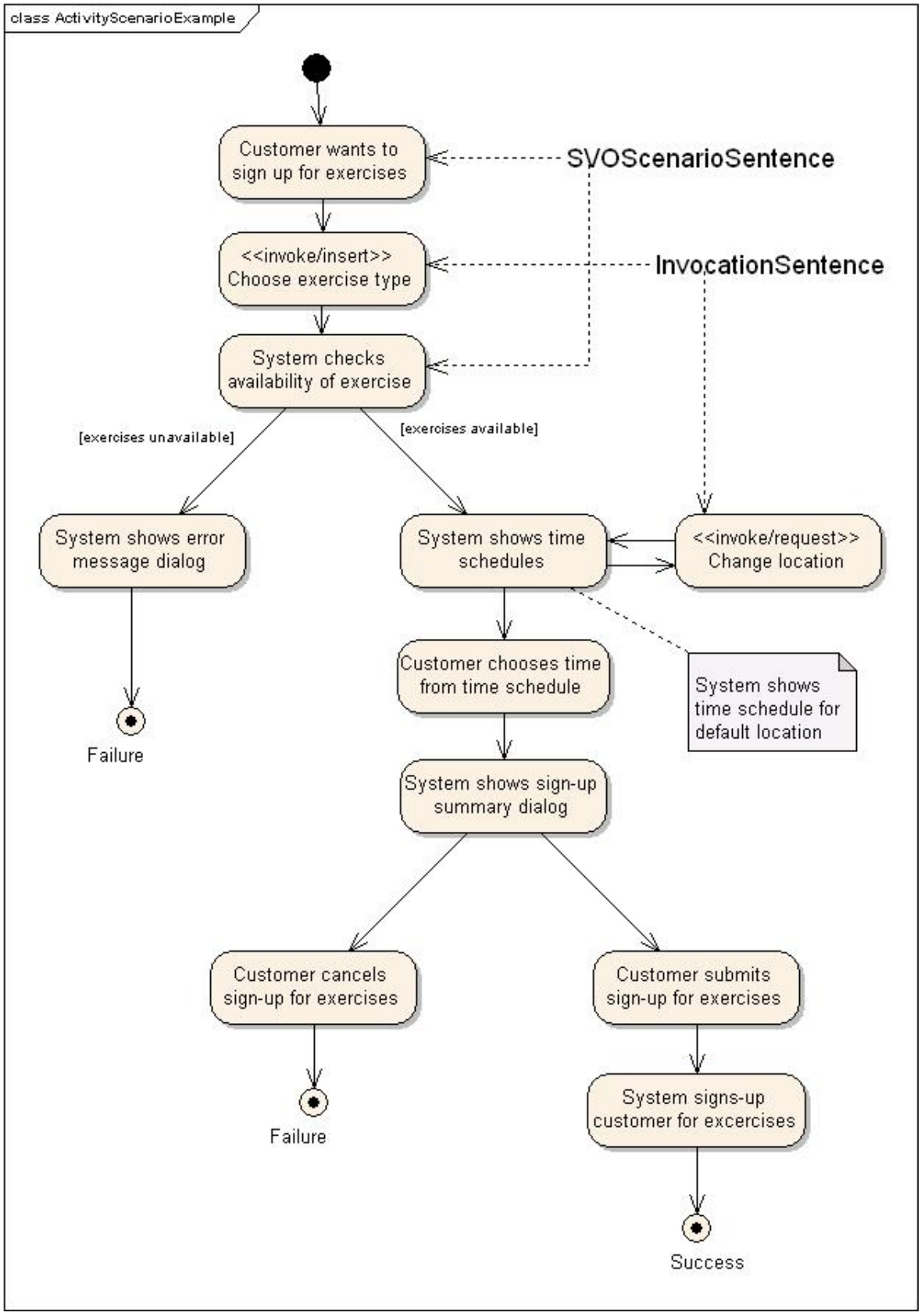

Abbildung 6.1: Aktivitätsdiagramm aus dem ReDSeeDS [IST06] EA Projekt

Kanten und ihre Attribute. Den Kanten folgen nur ihre Attribute. Genaueres zum Aufbau einer *tg*-Datei kann man in [Kah06] finden.
In Abb. 6.1 sieht man, dass es nur einen Startknoten gibt, der einen Pfeil zu dem Knoten mit der Beschriftung „Customer wants to sign up for exercises“ besitzt. Der Startknoten entspricht Knoten 20 und in seiner Kantenliste sieht man, dass die Kanten 19 und 48 eingehende Kanten sind, während Kante 40 die einzige ausgehende Kante ist. Die Kanten 19 und 48 beschreiben die Paketstruktur und die Inhalte des Aktivitätsdiagramms. Kante 40 ist vom Typ „CControlFlow“ und entspricht damit der Kante in Abb. 6.1.
Wenn man sich den Knoten 13 betrachtet, so erkennt man, dass unter anderem Kante 40 in diesen Knoten eingeht. Auf diese Weise wird auch der Rest der *eap*-Datei in einem TGraphen beschrieben.
Die Paketstruktur aus der *eap*-Datei ist in Abb. 6.1 nicht zu sehen. Da anhand des Aktivitätsdiagramms und des TGraphen nachvollzogen werden kann, wie das Überführungstool einen TGraphen aufbaut, wird auf weitere Erklärungen zur Paketstruktur verzichtet.

Listing 6.1: Aus Abb. 6.1 erstellter *tg*-Dateiausschnitt

```
Schema easchema.EARepositorySchema;
...
Graph "5B1374CBA328" EARepository (5000 5000 25 62);
1 Model <1> "Model" "";
2 Pakkage <-1 2> "ReDSeeDS␣Model" "";
...
6 Pakkage <-5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
  20 21 22 23 24> "ActivityScenarioExample" "";
7 DLogical <-6 45 46 47 48 49 50 51 52 53 54 55 56 57
  58 59 60 61 62> "ActivityScenarioExample" "" "Logical";
8 EActivity <-7 25 -30 -43 -61> f
  "<<invoke/insert>>␣␣Choose␣exercise␣type" "" "" 0 "Activity";
9 EActivity <-8 26 -37 -44 -57> f
  "<<invoke/request>>␣Change␣location" "" "" 0 "Activity";
10 EActivity <-9 27 -34 -54> f
  "Customer␣cancels␣sign-up␣for␣exercises" "" "" 0 "Activity";
...
13 EActivity <-12 30 -40 -42 -62> f
  "Customer␣wants␣to␣sign␣up␣for␣exercises" "" "" 0 "Activity";
...
19 ENote <-18 -36 -47> f ""
  "System␣shows␣time␣schedule␣for␣default␣location" "" 0 "Note";
20 EStateNode <-19 40 -48> f "" "" "" 100 "StateNode";
```

```
21 EStateNode <-20 -33 -49> f "Failure" "" "" 101 "StateNode";
22 EStateNode <-21 -27 -51> f "Failure" "" "" 101 "StateNode";
23 EStateNode <-22 -39 -50> f "Success" "" "" 101 "StateNode";
24 EText <-23 41 42 -45> f "" "SVOScenarioSentence" "" 0 "Text";
25 EText <-24 43 44 -46> f "" "InvocationSentence" "" 0 "Text";
1 ContainsPackagePackage;
...
5 ContainsPackagePackage;
6 ContainsPackageDiagram;
7 ContainsPackageElement;
...
19 ContainsPackageElement;
...
25 CControlFlow 0 "" "" "" "" 0 "" "" "" "ControlFlow";
...
31 CControlFlow 0 "" "" "" "" 0 "" ""
  "exercises␣available" "ControlFlow";
...
36 CNoteLink 0 "" "" "" "" 0 "" "" "" "NoteLink";
37 CControlFlow 0 "" "" "" "" 0 "" "" "" "ControlFlow";
...
40 CControlFlow 0 "" "" "" "" 0 "" "" "" "ControlFlow";
41 CDependency 0 "" "" "" "representation" 0 "" "" "" "Dependency";
...
48 ContainsDiagramElement -123 292 312 -103;
...
62 ContainsDiagramElement -203 235 377 -152;
```

Überführung JGraLab nach EA. Ein TGraph kann mit Hilfe des Überführungstools wieder in eine *eap*-Datei umgewandelt werden. Dazu wird als Quelldatei die *tg*-Datei gewählt und als Zieldatei eine existierende *eap*-Datei. Nach der Überführung des TGraphen aus Listing 6.1 in ein EA Projekt enthält dieses Projekt unter anderem das Aktivitätsdiagramm aus Abb. 6.2.

Auch hier entsprechen die Knoten und Kanten aus Listing 6.1 den Aktivitäten, Pfeilen usw. des Aktivitätsdiagramms aus Abb. 6.2. Wenn man die Qualität des Überführungstool analysieren möchte, so kann man die beiden Abbildungen 6.1 und 6.2 miteinander vergleichen, da diese Diagramme die Ergebnisse einer round trip Transformation visualisieren.

Es ist zu erkennen, dass das Überführungstool dazu in der Lage ist, alle inhaltlichen Bestandteile eines EA Projektes in einen TGraphen zu überführen und aus diesem wieder ein EA

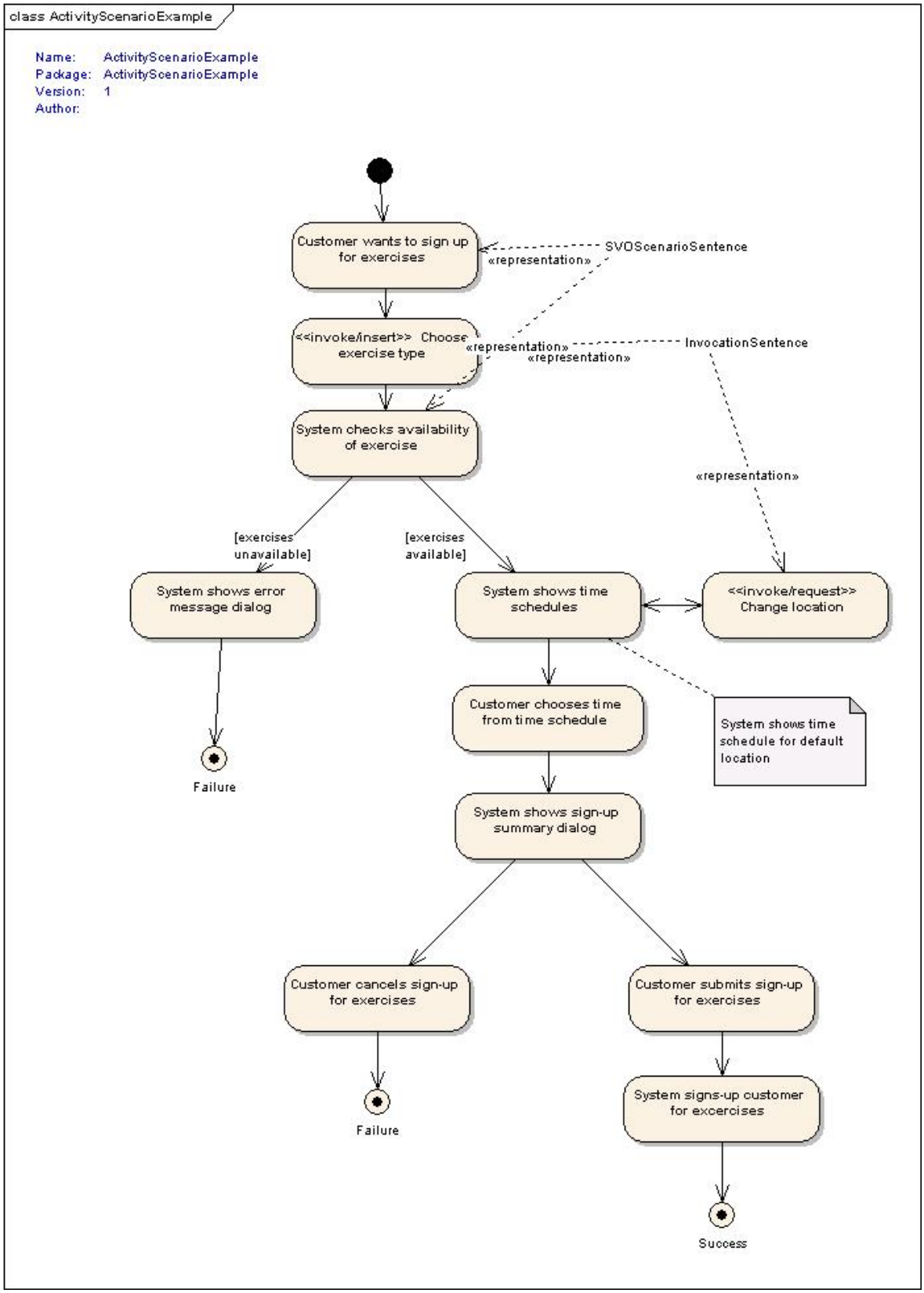

Abbildung 6.2: Aktivitätsdiagramm, das aus Listing 6.1 erstellt wurde

Projekt zu erzeugen. Es fehlen im Zieldiagramm 6.2 keine Details, die im Quelldiagramm 6.1 vorhanden waren. Dennoch existieren einige Unterschiede bezüglich der grafischen Darstellung des Diagramms:

- Die Schriftgrößen unterscheiden sich in beiden Diagrammen.
- Start- und Endpunkt der Pfeile sind zwar gleich, jedoch ist ihr Verlauf in beiden Diagrammen unterschiedlich dargestellt. Dadurch entsteht zwischen den Aktivitäten „System shows time schedules“ und „Change location“ in Abb. 6.2 ein Doppelpfeil, obwohl es sich tatsächlich um zwei verschiedene Pfeile handelt, siehe Abb. 6.1.
- Die Beschriftungen der Pfeile befinden sich zwar an den richtigen Pfeilen, jedoch an unterschiedlichen Positionen in den beiden Diagrammen.
- Im Zieldiagramm wurden Stereotypen an den Pfeilen eingeblendet, die im Quelldiagramm ausgeblendet sind. Ein Beispiel hierfür ist der Pfeil mit der Beschriftung „SVOScenarioSentence“.

Diese Unterschiede zwischen den beiden Diagrammen beruhen darauf, dass ein TGraph nicht zur visuellen Repräsentation von Diagrammen gedacht ist. Damit die Diagramme nach einer round trip Transformation aber noch erkennbar sind, werden im TGraphen die grafischen Positionen eines Elements als Attribute einer „ContainsDiagramElement“-Kante gespeichert.

Um die Qualität einer round trip Transformation durch das Überführungstool zu beurteilen, muss man den EA Repository Inhalt bzw. den TGraphen vor und nach der Überführung vergleichen. Das Betrachten zweier Diagramme vor und nach der Überführung macht den Vergleich zwar leichter, ist aber trotzdem keine ideale Vergleichsbasis, da TGraphen von den visuellen Informationen eines Diagramms abstrahieren. Aufgrund der schwierigen Darstellung zweier EA Repository Inhalte wurde in diesem Abschnitt trotzdem der Vergleich anhand zweier Diagramme durchgeführt.

6.2 Überführung eines großen EA Projekts

Zum Testen des Tools wurde nicht nur wie im vorherigen Abschnitt beschrieben ein Teil bzw. ein Aktivitätsdiagramm des ReDSeeDS EA Projektes überführt, sondern auch das vollständige ReDSeeDS RSL (requirements specification language) Metamodell von Januar 2007.
Die Überführung dieses größeren EA Projektes konnte anhand des definierten Schemas aus Abschnitt 5.1 fehlerfrei ausgeführt werden. Das Überführungstool stürzte also nicht ab, hat

alle verwendeten und im EA Schema definierten Inhalte des Repositorys erkannt und in einen TGraphen überführt.
Die Rücktransformation des TGraphen in ein EA Projekt konnte ebenfalls erfolgreich ausgeführt werden. Die so entstandene *eap*-Datei kann nun mit dem ursprünglichen ReDSeeDS EA Projekt verglichen werden. Dabei fällt auf, dass die round trip Transformation anhand des definierten EA Schemas funktioniert. Trotzdem gibt es noch einige Probleme, die daraus resultieren, dass das EA Schema nicht dem Metamodell des Enterprise Architects entspricht und dass TGraphen nicht die visuellen Aspekte eines Diagramms berücksichtigen. Eine genauere Beschreibung der offenen Punkte befindet sich in Abschnitt 6.3.

Die ReDSeeDS *eap*-Datei hatte im Januar 2007 eine Größe von 9.410 KB. Der TGraph, der beim Überführen des ReDSeeDS EA Projektes entstand, besteht aus circa dreitausend Knoten und achttausend Kanten. Diese Mächtigkeit sollte gängigen IT-Projekten entsprechen, somit ist gezeigt, dass das Überführungstool praxistauglich ist. Die Überführungen in beide Richtungen wurden auf einem Rechner mit 1,6 GHz und 512MB DDR Ram durchgeführt und dauerten jeweils circa 5 Minuten.

6.3 Offene Punkte und Weiterentwicklung

Bei der Entwicklung des Überführungstools kam es zu einigen Problemen, die durch Weiterentwicklungen gelöst werden können. Dieser Abschnitt zählt die noch nicht bearbeiteten, offenen Punkte auf.

EA Schema anpassen. Aufgrund des fehlenden EA Metamodells konnte nur ein manuell definiertes EA Schema erstellt werden, welches nicht vollständig ist.
Beim Testen sind einige Punkte aufgefallen, die noch im definierten EA Schema fehlen und daher nicht in einen Graphen übertragen werden.

- Beziehungen können mehrere *Constraint*-Instanzen besitzen, welche nicht durch das Überführungstool transformiert werden, obwohl sie innerhalb des ReDSeeDS EA Projektes verwendet wurden.
- Assoziierte Klassen können ebenfalls nicht überführt werden.
- Visuelle Features der Diagramme in einem EA Projekt werden nicht übertragen[1]. Beispiele für visuelle Features sind farbige Elemente, der genaue Verlauf einer Beziehung

[1] Informationen zu visuellen Features sind zwar als Attribute in den entsprechenden Objekten vorhanden, werden aber in dieser Studienarbeit nicht verarbeitet, da dies in dieser Studienarbeit zu weit führen würde.

(unabhängig von Start-, Endpunkt und Richtung), die Position von Beschriftungen und versteckte Beziehungen, die zwischen Elementen bestehen, aber im Diagramm nicht gezeigt werden sollen.

- Aufgrund dieser fehlenden visuellen Features gibt es zum Beispiel bei der Überführung eines Interaktionsdiagramms von JGraLab nach EA Probleme mit den Lebenslinien, da die Reihenfolge der Aktionen durch untereinander angeordnete Pfeile zwischen den einzelnen Lebenslinien unterschieden wird.

Mit dem Erscheinen von neueren Versionen des EA-API kann es dazu kommen, dass das EA Schema weiter angepasst werden muss. Die Liste der offenen Punkte in Abhängigkeit von dem EA-API wird also in Zukunft noch weiter wachsen, daher benötigt das Überführungstool in diesem Bereich ständige Aktualisierungen, wenn es den neusten UML- oder EA-Version gerecht werden soll.

Mehrerer EA Projekte bearbeiten. Die Verwendung des EA-API verursacht ein weiteres Problem. Das Überführungstool kann pro Ausführung immer nur eine *eap*-Datei lesend oder schreibend verarbeiten. Dieses Problem hängt mit dem EA-API zusammen und ist in Abschnitt 2.2 genauer beschrieben. Falls eine neuere Version des EA-API dieses Problem löst, so muss die grafische Benutzungsoberfläche des Überführungstools angepasst werden.

Performance steigern. Der Algorithmus, der zur Überführung von EA nach JGraLab und umgekehrt verwendet wird, arbeitet in zwei Schritten und speichert die IDs aller Diagramme und Elemente, siehe dazu Abschnitt 5.3. Dies ist zwar kein Problem im eigentlichen Sinne, trotzdem entsteht somit überflüssiger Ressourcenverbrauch und eine Performanceverschwendung. Im Zuge einer Weiterentwicklung sollte es Möglich sein, einen Algorithmus zu entwickeln, der nur einen Schritt benötigt und deshalb auch nicht die IDs aller Diagramme und Elemente zwischenspeichern muss.

Refaktorisieren. Wenn man sich Abb. 5.3 genauer betrachtet, so erkennt man, dass die Methoden der *Writer*-Klassen ziemlich lange Parameterlisten besitzen. Für jedes weitere Attribut, das dem EA Schema hinzugefügt wird, wächst die Parameterliste einer „create“-Methode.
Die langen Parameterlisten riechen nach einem „Datenklumpen“ [Fow00]. Um diesen schlechten Geruch zu beseitigen, werden in [Fow00] die Refaktorisierungen „Parameterobjekt einführen“ und „Ganzes Objekt übergeben“ empfohlen. Bevor also das EA Schema weiterentwickelt wird, sollten diese Refaktorisierungen in den *Writer*-Klassen durchgeführt werden.

7 Fazit

7.1 Bewertung

Die Entwicklung des Überführungstools soll in diesem Abschnitt an den Anforderungen aus Abschnitt 4.2 gemessen werden. Dazu wird der Entwicklungsstand mit den Anforderungen verglichen.

Bewertung: funktionale Anforderungen. Die funktionalen Anforderungen 4 bis 7 konnten vollständig erfüllt werden. Die Anforderungen 1, 2 und 3 sind unter Berücksichtigung des zu definierenden Metamodells ebenfalls erfüllt, allerdings sollte das Metamodell im Zuge von Weiterentwicklungen noch genauer definiert werden, so dass es später UML-Diagramme vollständig repräsentiert. Die 8. Anforderung ist ebenfalls erfüllt, wobei das Überführungstool tatsächlich immer nur genau eine Transformation pro Ausführung vollziehen kann.

Bewertung: technische Anforderungen. Die technischen Anforderungen wurden vollständig erfüllt.

Bewertung: Anforderungen an die Benutzerschnittstelle. Die Anforderungen an die Benutzerschnittstelle wurden teilweise in abgewandelter Form realisiert. Zur 2. Anforderung ist zu sagen, dass die Überführungsrichtung von der Quell- und Zieldatei festgelegt wird, so dass nur ein Button zum Ausführen der Transformation nötig ist und keine zwei separaten Buttons für die Überführungsrichtung nötig sind. Die anderen Anforderungen sind entsprechend erfüllt.

Bewertung: Qualitätsanforderungen. Die Qualitätsanforderungen wurden vollständig erfüllt.

Bewertung: sonstige Anforderungen. Die sonstigen Anforderungen wurden vollständig erfüllt.

7.2 Zusammenfassung

Das Hauptziel dieser Studienarbeit war die Entwicklung eines Überführungstools, das sowohl den Repository Inhalt eines EA Projektes in einen TGraphen als auch einen TGraphen in ein EA Projekt überführen kann. Außerdem sollte das application programming interface des Enterprise Architects analysiert und beschrieben werden. Diese beiden Ziele konnten erfüllt werden und wurden in den vorherigen Kapiteln beschrieben. Zusätzlich beinhaltet die Studienarbeit noch eine Beschreibung und ein Anwendungsbeispiel zu JGraLab.

Diese Studienarbeit eignet sich als praktische Einführung in die Nutzung des EA-API, da zum einen das EA-API genauer vorgestellt wird und zum anderen auch auf die Probleme, die das EA-API mitbringt, hingewiesen wird. Insbesondere wird die Nutzung anhand eines ausführlichen Beispiels in Java erklärt, was in sofern besonders ist, da der EA User Guide [Spa06b] in erster Linie Beispiele in Visual Basic Dot Net bietet.
Neben der Einführung in die EA-API bietet die Studienarbeit auch eine praktische Einführung in das Java Graphenlabor. Diese hat ihren Fokus ebenfalls auf einem Beispiel, das zeigen soll, wie ein Schema erstellt und in einem, diesem Schema entsprechenden, TGraphen geschrieben und gelesen wird.

Der Hauptteil der Studienarbeit befasst sich mit dem Entwicklungsprozess des Überführungstools. Dazu gehören zum einen die Planung und zum anderen eine genaue Beschreibung der Entwicklung des Überführungstools.
Die Planung beschränkt sich auf eine kurze Beschreibung der Aufgaben und Architektur des Überführungstools und eine ausführliche Anforderungsliste.
Die Entwicklung wird am ausführlichsten beschrieben. Der erste Schritt war die Definition des EA Schemas in JGraLab, da auf diesem Schema erste Prototypen, die die Machbarkeit des Überführungstools demonstrierten, arbeiteten. Die Komponenten, die Klassen und die Ausnahmebehandlung des Überführungstools werden ebenfalls in eigenen Kapitel beschrieben, weil sie wichtige Dokumente für die Weiterentwicklung des Überführungstools sind. Für den Benutzer des Überführungstools gibt es eine Beschreibung des API, der Benutzungsoberfläche und der Steuerung per Konsole.

In einem abschließenden Kapitel wird die Funktion des Überführungstools anhand eines Beispiels veranschaulicht. In diesem Beispiel wird ein Aktivitätsdiagramm aus einem EA Projekt nach JGraLab überführt und anschließend wird es zurück in ein neues EA Projekt geschrieben. Außerdem werden die offenen Punkte, die bei der Überführung des ReDSeeDS EA Projektes aufgefallen sind, genannt und überlegt wie man das Überführungstool weiterentwickeln kann.

Literaturverzeichnis

[BRSS07] BILDHAUER, Daniel ; RIEDIGER, Volker ; SCHWARZ, Hannes ; STRAUSS, Sascha. *Eine UML-basierte Modellierungssprache für T-Graphen*. ISSN (Print) 1864-0346, ISSN (Online) 1864-0850. Januar 2007

[Cha85] CHARTRAND, Gary: *Introductory Graph Theory*. Ney York : Dover Publications Inc., 1985. – ISBN 978–0486247755

[DW03] DAHM, Peter ; WIDMANN, Friedbert. *GraLab Das Graphenlabor Benutzungshandbuch*. `http://www.uni-koblenz.de/~ems/GraLab4/`. Juli 2003

[Fow00] FOWLER, Martin: *Refactoring*. München : Addison Wesley, 2000. – ISBN 3–8273–1630–8

[GHJV95] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns. Elements of Reusable Object-Oriented Software*. 1st ed. Amsterdam : Addison Wesley, 1995. – ISBN 0–201–63361–2

[GNU06] GNU. *GNU getopt - Java port*. `http://www.urbanophile.com/arenn/hacking/getopt/Package-gnu.getopt.html`. August 2006

[IST06] IST. *Requirements-Driven Software Development System*. `http://www.redseeds.eu/`. Septemer 2006

[Kah06] KAHLE, Steffen: *JGraLab: Konzeption, Entwurf und Implementierung einer Java-Klassenbibliothek für TGraphen*, Universität Koblenz-Landau, Diplomarbeit, Juni 2006

[OMG05] OMG. *Unified Modeling Language: Superstructure*. `http://www.omg.org/cgi-bin/doc?ptc/06-04-02.pdf`. August 2005

[SBR06] SCHWARZ, Hannes ; BILDHAUER, Daniel ; RIEDIGER, Volker. *JGraLab Citymap Tutorial*. noch nicht veröffentlicht. November 2006

[Spa06a] SPARX. *Enterprise Architect - UML Design Tools*. `http://www.sparxsystems.com/products/ea.html`. 2006

[Spa06b] SPARX. *Enterprise Architect Version 6.5 User Guide*. http://www.sparxsystems.com.au/bin/EAUserGuide.pdf. September 2006

[Ste06] STEFFENS, Tim: *Kontextfreie Suche auf Graphen*, Universität Koblenz-Landau, Diplomarbeit, Januar 2006

[Sun99] SUN. *Code Conventions for the Java Programming Language*. http://java.sun.com/docs/codeconv/. April 1999