
D I P L O M A R B E I T

Entity Relationship Modelling Toolkit ERD

Ausgeführt im Schuljahr 2018/19 von:

Berndt FISCHBACHER	5BHIF
Christian PASSET	5BHIF
Andreas PRINZ	5BHIF
Nicolas HOMOLKA	5BHIF

Betreuer / Betreuerin:

Dipl.-Ing. Günther Burgstaller

Wiener Neustadt, am 5. April 2019

Abgabevermerk:

Übernommen von:

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche erkenntlich gemacht habe.

Wiener Neustadt, am 5. April 2019

Verfasser / Verfasserinnen:

Berndt FISCHBACHER

Christian PASSET

Andreas PRINZ

Nicolas HOMOLKA

Inhaltsverzeichnis

Eidesstattliche Erklärung	i
Vorwort	v
Diplomarbeit Dokumentation	vi
Diploma Thesis Documentation	viii
I Kurzfassung	x
Kurzfassung	xi
II Abstract	xii
Abstract	xiii
III Einführung	1
1 Aufgabenstellung	2
1.1 Auslöser	2
1.2 Einsatz und Nutzen	2
2 Wieso das ER-Modell?	3
2.1 Allgemeines zu semantischen Datenmodellen	3
2.2 Entity-Relationship-Modell	3
2.2.1 Grundkonzept	3
2.2.2 Beziehungstypen	6
2.2.3 Attribute	7
2.3 Andere Vertreter semantischer Datenmodelle	7
IV Methodik	10
3 Methoden	11
3.1 Python	11
3.1.1 Entwicklung	11
3.1.2 Idee und Zweck	11
3.1.3 Verwendung	11

3.2	Pycharm als IDE	12
3.2.1	Allgemeines	12
3.2.2	Funktionen	12
3.2.3	Vorteile	13
3.3	XML	14
3.3.1	Allgemeines zu XML	14
3.3.2	XERML	17
3.4	PIC-Code	21
3.4.1	Definition	21
3.4.2	Aufbau einer PIC Datei	21
3.4.3	Objekte zeichnen in PIC	22
3.4.4	Layout	28
3.4.5	Erstellen einer PIC-Datei	29
3.4.6	Vorteile und Nachteile von PIC	29
3.5	yEd	33
3.5.1	Varianten und Installation	33
3.5.2	Oberfläche	34
3.5.3	Editier-Hilfen	38
3.5.4	Layout-Algorithmen	39
3.6	GraphML	42
3.6.1	Hintergrund	42
3.6.2	Grundelemente und Aufbau	42
3.6.3	Generierung der Datei und Aufbau	43
3.6.4	Python-Code für die Generierung	47
3.7	Graphviz	50
3.7.1	Allgemeines	50
3.7.2	Dateiformate	54
3.7.3	Engines	57
3.7.4	Vor- und Nachteile von Graphviz	60
3.7.5	Vergleich von dot und neato	61
3.8	LibreOffice Draw	64
3.8.1	Einführung in LibreOffice Draw	64
3.8.2	Entwicklung	64
3.8.3	Generierungsvarianten	64
3.8.4	Layout	87
3.9	Git	87
3.10	Trac	88
3.11	Sphinx	89
3.11.1	Generierung der Dokumentation	89
3.11.2	Aufbau der Dokumentation	90
4	Testmodelle	92
4.1	Überblick	92
4.2	Schulinformationssystem	93
4.3	Rettungsstelle	94
4.4	Fußball	95
4.5	Weingut	97
4.5.1	Die Hauptdatei	97
4.5.2	Die Sprachdatei	99

4.5.3 Die Typdatei	99
5 Ergebnis	100
5.1 Diagramm mit GraphML	100
5.1.1 Aufbau der GraphML-Datei	100
5.1.2 Beispiel eines ER-Diagramms in yEd	100
5.2 Ergebnis der Darstellung mit Graphviz	101
5.3 Diagramm mit PIC-Code	103
5.3.1 Aufbau des PIC-Codes	103
5.3.2 Problemfälle bei dem erstellten ERD	103
5.3.3 Ist PIC-Code für das Projekt geeignet?	104
5.4 Diagramm mit LibreOffice Draw	107
5.4.1 Probleme bei der Darstellung	107
5.4.2 Beispiel eines ER-Diagramms in LibreOffice Draw	107
V Schlussfolgerung	109
6 Schluss	110
6.1 Ziele	110
6.1.1 Welche Ziele wurden erreicht?	110
6.2 Zukunft des Tools	110
6.3 Verwendung	111
6.3.1 Hilfe	111
6.3.2 Programm Aufruf	111
6.3.3 “erdgenerate,”	112
7 Technische Ergänzungen	113
7.1 Testmodelle	113
7.1.1 Schulinformationssystem	113
7.1.2 Rettungsstelle	113
7.1.3 Fußball	113
7.1.4 Weingut	113
7.2 Python-Code	113
7.2.1 GraphML	113
Index	116
Literatur	117

Vorwort

Diese Diplomarbeit entstand durch die Aufgabe die Erstellung von Entity-Relationship Diagrammen zu vereinfachen. Wir, das Diplomarbeitsteam, haben uns dieser Aufgabe gestellt und möchten uns bei unserem Betreuer Dipl.-Ing Günter Burgstaller für das Vertrauen zur Bewältigung der Aufgabe bedanken. Außerdem möchten wir uns bei ihm für die Geduld bedanken, die er uns entgegengebracht hat.

Zudem möchten wir uns bei unseren Freunden und Familien für die seelische Unterstützung bedanken. Sie standen hinter uns und haben uns angespornt weiter zu machen, wenn wir auf Probleme stießen.

Wir hoffen, das Ihnen diese Diplomarbeit eine wertvolle Hilfe ist im Bezug auf semantische Datenmodelle und die Varianten mit denen wir diese darstellen.

In diesem Sinne wünscht Ihnen das Diplomarbeitsteam viel Freude beim Lesen dieser Arbeit!

Diplomarbeit Dokumentation

Namen der Verfasser/innen	Berndt Fischbacher Nicolas Homolka Christian Passet Andreas Prinz
Jahrgang Schuljahr	5BHIF 2018 / 19
Thema der Diplomarbeit	Entity Relationship Modeling Toolkit ERD (ERMTK-ERD)
Kooperationspartner	
Aufgabenstellung	Automatische Generierung von ER-Diagrammen für diverse Ausgabeformate. Durch Angabe verschiedener Parameter kann das Aussehen des Diagramms verändert werden.
Realisierung	Die Daten über das ER-Diagramm aus der XERML-Datei extrahieren. Realisierung der Implementierung für die Konvertierung zu den einzelnen Ausgabeformaten mittels Python.
Ergebnisse	Das ER-Diagramm wird in jedem Ausgabeformat vollständig und korrekt generiert. Parametereingaben von dem Benutzer werden vollständig umgesetzt.

Typische Grafik, Foto etc. (mit Erläuterung)

Teilnahme an Wettbewerben, Auszeichnungen	
---	--

Möglichkeiten der Einsichtnahme in die Arbeit	HTBLuVA Wiener Neustadt Dr.-Eckener-Gasse 2 A 2700 Wiener Neustadt
---	--

Approbation	Prüfer	Abteilungsvorstand
(Datum, Unterschrift)	Dipl.-Ing. Günther Burgstaller	AV Dipl.-Ing. Felix Schwab

Diploma Thesis Documentation

Authors	Berndt Fischbacher Nicolas Homolka Christian Passet Andreas Prinz
Form	5BHIF
Academic Year	2018 / 19
Topic	Entity Relationship Modeling Toolkit ERD (ERMTK-ERD)
Co-operation partners	

Assignment of tasks	Automatic generation of ER-Diagrams for various output formats. By specifying various parameters, the appearance of the diagram can be changed.
---------------------	---

Realization	Extract the data about the ER-Diagram from the XERML-file. Realization of the implementation for the conversion to the individual output formats using Python.
-------------	--

Results	The ER-Diagram is generated completely and correctly in each output format. Parameter inputs from the user are fully implemented.
---------	---

Illustrative graph, photo (incl. explanation)	
--	--

1850 x 1950

1850

1750

1650

Participation in competitions, Awards	
---	--

Accessibility of diploma thesis	HTBLuVA Wiener Neustadt Dr.-Eckener-Gasse 2 A 2700 Wiener Neustadt
------------------------------------	--

Approval (Date, Sign)	Examiner Dipl.-Ing. Günther Burgstaller	Head of Department AV Dipl.-Ing. Felix Schwab
------------------------------	--	--

Teil I

Kurzfassung

Kurzfassung

In der Diplomarbeit ERMTK-ERD wurde ein Programm entwickelt, das dazu dient automatisch Entity-Relationship Diagramme aus XML-Dateien zu generieren. Diese Generierung wurde mit vier verschiedenen Ausgabeformaten implementiert. Um diese vier Varianten zu einem Tool zu kombinieren wurde die Diplomarbeit Entity-Relationship-Modeling-Toolkit ins Leben gerufen. Außerdem soll durch Angabe von Kommandozeilenparametern das Aussehen der Diagramme durch den Benutzer verändert werden können. Die vier verschiedenen Ausgabeformate sind:

- Graphml
- LibreOffice Draw
- PIC-Code
- Graphviz

Teil II

Abstract

Abstract

The goal of the ERMTK-ERD project was to implement a program that generates automatically, with an XML-File as input, an Entity-Relationship Diagram. This was implemented with four different output formats. To combine these four variations, this proprietary Entity-Relationship-Modeling-Toolkit has been written. Furthermore, the user should be able to adjust the appearance of the graph with specific commandline parameters. The four different base programmes are:

- Graphml
- LibreOffice Draw
- PIC-Code
- Graphviz

Teil III

Einführung

Kapitel 1

Aufgabenstellung

1.1 Auslöser

Das ERD (Entity-Relationship-Diagramm) ist seit seiner Entstehung nicht mehr vom Prozess des Datenbank-Designs wegzudenken. Durch dieses Diagramm lässt sich leichter ein konzeptioneller Entwurf für eine Datenbank entwickeln.

Das Erstellen eines solchen Diagramms ist, bis zu einer gewissen Größe des Datenmodells noch per Hand bewältigbar, wird jedoch mit ansteigenden Anzahl von Entitäten schnell komplex und zudem auch mühsam, im Bezug auf die Anordnung der einzelnen Elemente innerhalb des Diagramms.

Aus diesem Grund hat Prof. DI Günter Burgstaller das Diplomarbeitsteam beauftragt, ein *englischsprachiges Command-Line-Tool* zu entwickeln, das die Erstellung eines solchen Entity-Relationship-Diagramms vereinfachen soll.

1.2 Einsatz und Nutzen

Bisher ist Erstellung eines ERD mit Applikationen wie dem Oracle SQL Developer Data Modeler eine aufwendige Arbeit. Um zum Beispiel ein Attribut zu einem bestehendem Entity hinzuzufügen sind mehrere Schritte notwendig. Damit dieser Arbeitsaufwand gesenkt wird kommt das Tool zum Einsatz, weil für das hinzufügen eines Attributes lediglich eine Zeile in der Eingabedatei ergänzt werden muss.

Der Nutzen, den man sich von diesem Tool verspricht ist es, sich ein Diagramm in Form von verschiedenen Ausgabeformaten automatisch zeichnen zu lassen, was einem Zeit ersparen soll. Für den Unterricht kann dies bedeuten, dass die Schüler in kürzerer Zeit lernen ein ERD zu lesen und es richtig zu modellieren.

Als Eingabe erwartet das Werkzeug eine XML-Datei im XERML-Format das von Prof. DI Günter Burgstaller entwickelt wurde.

Kapitel 2

Wieso das ER-Modell?

2.1 Allgemeines zu semantischen Datenmodellen

Ein Großteil der semantischen Datenmodelle wurde in den 1970er entwickelt¹. Sie dienen der Erstellung einer ersten formalen Beschreibung der Datenbank und werden im Buch „Taschenbuch Datenbanken“ wie folgt definiert:

Semantische Datenmodelle beschreiben einen Weltausschnitt als Menge von Gegenständen (Objekten), zwischen denen wohldefinierte Beziehungen existieren und die durch Eigenschaften charakterisiert werden.

Durch die Veranschaulichung von Informationen gehören die semantischen Datenmodelle in der Informationsmodellierung zu den Standards. Das am weitesten verbreitete ist das *Entity-Relationship-Modell* von Peter P. Chen¹.

2.2 Entity-Relationship-Modell

2.2.1 Grundkonzept

Im Grunde besteht das ER-Modell aus einer handvoll an Elementen:

- Das zu modellierende Objekt genannt Entity und dessen Typ
- Die Beziehung oder Relationship und deren Typ
- Attribute die die Eigenschaften eines Entitytypen repräsentieren
- Attribute die die Eigenschaften einer Beziehung wiedergeben

In der Abbildung 2.1 wird eine minimalistischste Version eines ER-Diagramms gezeigt, welches die grafische Darstellung des ER-Modells ist. Das Beispiel selbst gibt den Sachverhalt wieder, dass die Entitytypen „wein“ und „rebsorte“ über den Beziehungstyp „beinhaltet“ miteinander assoziieren.

¹Thomas Kudraß. *Taschenbuch Datenbanken*. Deutsch. 2. Aufl. München: Carl Hanser Verlag, 2015.
URL: <https://www.hanser-fachbuch.de/buch/Taschenbuch+Datenbanken/9783446435087>.



Abbildung 2.1: Teil eines simplen ER-Diagramms

Instanzebene

→ Passet

Bei einem *Entity* handelt es sich stets um ein eindeutiges, abgrenzbares Objekt aus der realen Welt oder anders formuliert, um die zu repräsentierende Informationseinheit innerhalb einer Datenbank. In einem ER-Diagramm werden jedoch nicht die einzelnen Entitäten dargestellt, sondern eine Menge von ihnen, die über den *Entitytyp* definiert sind.

Zwischen den Entitäten sind Beziehungen definiert, die ebenfalls nicht einzeln, sondern durch den dazugehörigen *Beziehungstyp* in einem Diagramm angezeigt werden.

Eine nähere Beschreibung von Entitäten und Beziehungen erfolgt durch die Verwendung von Attributwerten. Jene Werte werden wiederum in Wertemengen zusammengefasst und durch Wertebereiche definiert.

Ein Wertebereich wird im Normalfall durch einen Standard-Datentyp beschrieben. Die gängigsten darunter sind:

- **char**
 - für die Darstellung von Zeichenketten
- **integer**
 - für ganze Zahlen
- **date**
 - für Datumswerte

In der Abbildung 2.2 werden, mit einem Beispiel aus dem Skriptum „Datenbanken und Informationssysteme“, die Entitäten vom Typ „*Abteilung*“ durch die Werte „*Verkauf*“ und „*Forschung*“ und die Beziehung durch den Wert „*leitet*“ dargestellt.

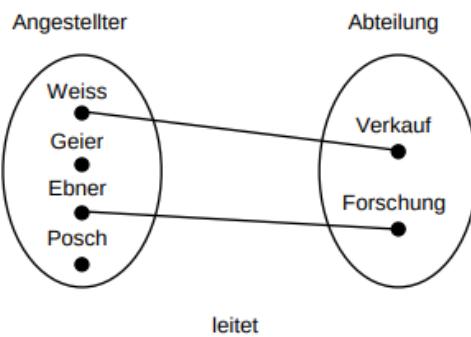


Abbildung 2.2: Beispiel für Instanzebene aus „Datenbanken und Informationssysteme“

Typebene

Bei der Typebene geht es um die bereits erwähnten Entity- und Beziehungs-Typen.

Unter einem Entitytyp versteht man die Menge der Entitäten mit gleichen Attributen². Diese werden wie in Abbildung 2.3 durch Rechtecke dargestellt. Ein jeder Entitytyp besitzt einen Namen. Üblicherweise handelt es sich dabei um ein Nomen in der Einzahl.

Bei den Beziehungs-Typen kommt das selbe Prinzip zu tragen, wie bei den Entitytypen, nur das es sich beim Namen nicht um ein Nomen sonder meist um ein Verb handelt. Im Skriptum „Datenbanken und Informationssysteme“ wird ein Beziehungstyp so definiert:

Beziehungen, an denen Entities des gleichen Entity-Typs beteiligt sind und die dieselbe Bedeutung haben, werden zu einem Beziehungs-Typ zusammengefasst.

Wie bereits in Abbildung 2.3 ersichtlich, wird ein Beziehungs-Typ durch eine Raute im ER-Diagramm dargestellt.

Das letzte grundlegende Element eines ER-Diagramms ist das *Attribut*. Ein Attribut wird durch eine Ellipse dargestellt und ist dem jeweiligem Entity- bzw. Beziehungs-Typen zugeordnet, um dessen Eigenschaften anzugeben. Durch sie wird es möglich, den jeweiligen Typ zu klassifizieren, charakterisieren und identifizieren³.

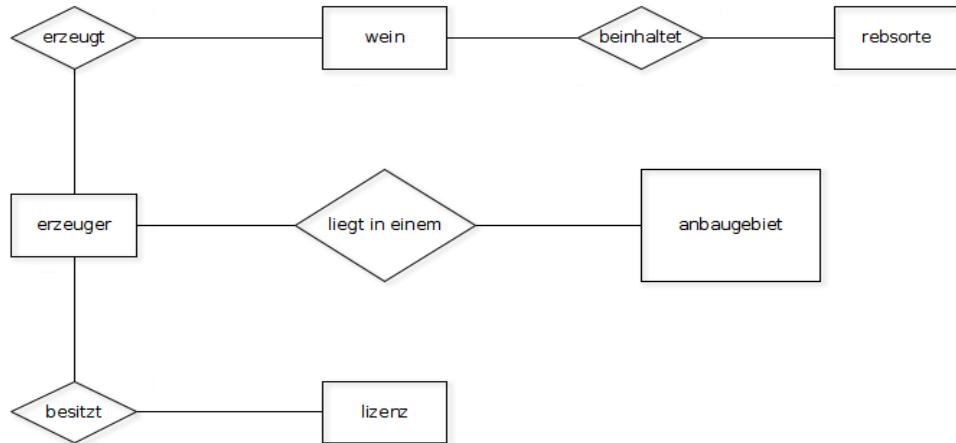


Abbildung 2.3: Darstellung von Attributen, Entity- und Beziehungs-Typen aus dem Datenmodell Weingut

2.2.2 Beziehungstypen

→ Passet

Kardinalitäten

Unter der *Kardinalität* eines Beziehungstypen versteht man die quantitative Beschreibung mit Hilfe der (1,M,N)-Notation³. Durch dieses Modell erhält man die Information, wie viele Entitäten des einen Entitytypen mit wie vielen Entitäten des anderen Entitytypen in Beziehung stehen können. Dabei wird jede Beziehung in zwei gerichtete Teile aufgeteilt, damit diese getrennt betrachtet werden können.

Dem Namen entsprechend kommen bei der Notation als Maximalwerte viele, sprich N bzw. M oder 1 in Frage woraus sich folgende Kombinationen bei den Beziehungstypen ergeben:

- 1:1 isting
- 1:N bzw. N:1
- M:N

(min,max)-Notation

Bei der (min,max)-Notation handelt es sich um eine Erweiterung der (1,M,N)-Notation, denn durch das Hinzufügen der 0 in den Wertebereich, ist es dem Nutzer möglich ein Minimum für die Beziehungstypen zum Ausdruck zu bringen.

Dargestellt werden die Angaben als Intervalle. Außerdem wird bei dieser Notation das „N“ in seiner grundlegenden Form durch einen „*“ dargestellt⁴. In der nachstehenden Abbildung finden Sie das vorangegangene Beispiel beschrieben durch die (min,max)-Notation.

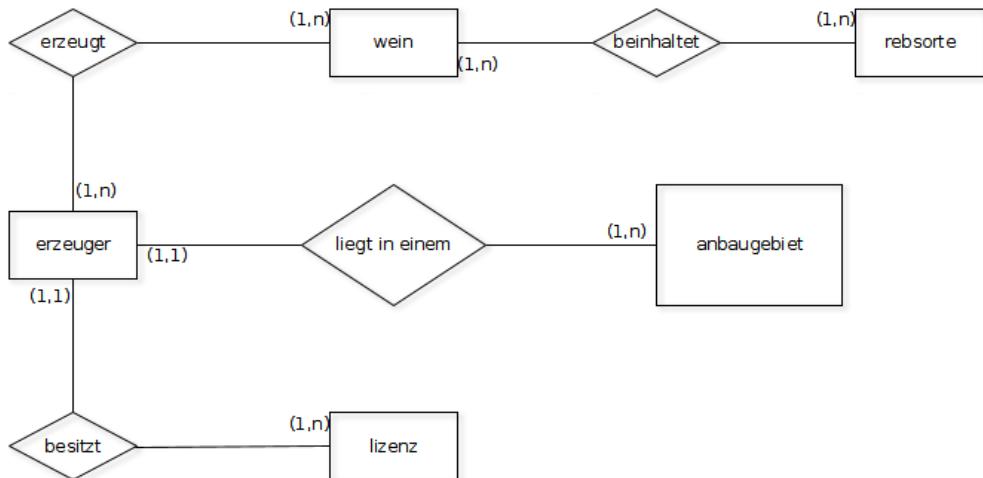


Abbildung 2.4: ER-Diagramm beschrieben durch (min,max)-Notation

²Kurt Hillebrand. „Datenbanken und Informationssysteme“. Skriptum das im DBI-Unterricht an der HTBLuVA Wiener Neustadt verwendet wird.

³Kudraß, *Taschenbuch Datenbanken*.

⁴Kudraß, *Taschenbuch Datenbanken*.

Bei manchen Umsetzungen von Datenmodellen kann es zu einem Punkt kommen an dem man die Objekteigenschaft als Attribut oder Entitytyp definieren kann⁵. Deshalb stellt sich die Frage, wann man eine Informationseinheit im ER-Modell als Attribut oder Entitytyp modelliert. Die folgenden Merkmale helfen bei der Einordnung:

- Das Attribut ist immer einem Entitytyp zugeordnet, welcher eigenständig ist.
- Der Entitytyp wird durch seine Merkmale definiert. Attribute wiederum durch den erhaltenen Namen, Wert und Datentyp.

Die Attribute werden in zwei Kategorien aufgeteilt. In Schlüssel- und Nichtschlüsselattribute. Ein Schlüssel wird im Buch „Taschenbuch Datenbanken“ folgend definiert:

Eine Attributmenge wird als **Schlüssel** bezeichnet, wenn sie eine eindeutige Identifizierung eines Entities eines Entitytyps ermöglicht und minimal ist.

Der für die Identifizierung verwendete Schlüssel wird *Primärschlüssel* (*Primary Key*) genannt und üblicherweise unterstrichen um ihn hervorzuheben. Sollte man keinen eindeutigen, minimalen Schlüssel finden wird ein „*künstlicher*“ *Schlüssel* verwendet. Diese sind meist Zahlen die pro weiteren Entity um eins erhöht werden.

Die bereits erwähnten Nichtschlüsselattribute sind jene, die nicht als Schlüssel geeignet sind und deshalb rein der Charakterisierung dienen⁶. Bei der Zuordnung der Attribute sollte man die Vermeidung von Redundanzen beachten, dass das Merkmal innerhalb des Schemas eindeutig ist und nicht in mehreren Entities, bezüglich auf den Wert des Merkmals, zugeordnet ist.

2.3 Andere Vertreter semantischer Datenmodelle

Neben dem Entity-Relationship-Modell von Chen sind während den 70er und 80er Jahren noch weitere semantische Datenmodelle entstanden. Abgesehen vom „Relational Model/Tasmania“ von Codd, und dem „Functional Data Model“ von Kerschberg⁶ gibt es noch das folgende Modell:

SDM (Semantic Data Model)

- Wurde von Hammer und McLoed 1981 erfunden.
- Laut dem Buch Prabhu, „Object - Oriented Database Systems : Approaches and Architectures“ befolgt es folgende Grundsätze:
 1. To serve as a formal specification mechanism for describing the meaning of a database.
 2. To provide a documentation and interaction medium for database users.
 3. To provide a high-level, semantic based front-end user interface to a database.

⁵Hillebrand, „Datenbanken und Informationssysteme“.

⁶Kudraß, *Taschenbuch Datenbanken*.

4. To serve as a foundation for supporting effective and structured design of databases.

UML (Unified Modeling Language)

→Passet

Obwohl der Nutzen von UML hauptsächlich der Modellierung von Software dienen sollte, wurden die Klassendiagramme auch für die Modellierung von semantischen Datenmodellen verwendet.

Bei dieser Form der Modellierung werden die Entitytypen als Klassen dargestellt und die Attribute in Form von Klassenattributten. Der Bereich für die Methoden der Klasse bleibt dabei leer. In der Abbildung 2.5 sieht man wie laut Hills, *NoSQL and SQL Data Modeling* die Modellierung eines semantischen Datenmodells aussehen kann.



Abbildung 2.5: Semantisches Datenmodell in UML

Die Umsetzung mit UML ist aber nicht perfekt und bringt einige Schwachstellen mit sich. Zum einen wäre das die Möglichkeit, dass Attribute durch ein vorstehendes „-“ als privat dargestellt werden können. Dieser Umstand ist in der Realität für die Daten einer Datenbank nicht gegeben. Des Weiteren fehlt einem die Möglichkeit Attribute als Schlüssel darzustellen. Dadurch kann ein Datenmodell nicht vollständig über UML realisiert werden ⁷.

ORM (Object Role-Modeling)

Das ORM gehört zu den Fakt-basierten Modellierungen⁷. Das besondere an dieser Umsetzung ist die Herangehensweise. Um das Modell am Ende richtig darstellen zu können, hat man als Modellierer Aussagen mit Fakten als Ausgangspunkte. Für das Beispiel in diesem Unterkapitel wurde ein Beispiel aus dem *NoSQL and SQL Data Modeling* übernommen. Dieses Beispiel hat folgende Aussagen mit Fakten verwendet:

⁷Hills, *NoSQL and SQL Data Modeling*.

- Sam Houston works at 123 East Main Street, Dallas, Texas 75208.
- Dolly Doolittle works at 123 East Main Street, Dallas, Texas 75208.
- Sam Houston lives at 456 Pine Street, Fort Worth, Texas 76104.
- Dolly Doolittle lives at 789 Elm Street, Fort Worth, Texas 76104.
- Sam Houston's mobile phone number is 214-555-1212.
- Sam Houston's FAX phone number is 214-555-9999.
- Dolly Doolittle's home phone number is 214-555-1234.

Durch diese Aussagen wird das Diagramm in Abbildung 2.6 erstellt. Die abgerundeten Rechtecke repräsentieren hier Objekt der realen Welt oder einen Wert, also eine Zahl oder Zeichenkette. Bei dieser Fakt-basierten Modellierung werden die Attribute durch Beziehungen zu den jeweiligen abgerundeten Rechtecken dargestellt. Das kann man in Abbildung 2.6 beim Objekt *Postal Address* sehen⁸.

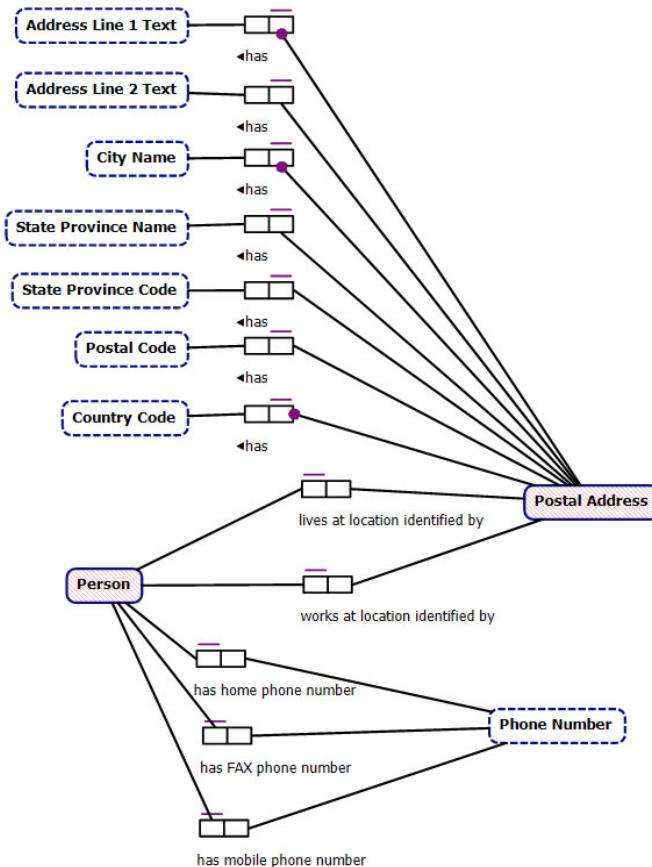


Abbildung 2.6: Beispiel eines Datenmodells dargestellt in ORM

⁸Hills, *NoSQL and SQL Data Modeling*.

Teil IV

Methodik

Kapitel 3

Methoden

3.1 Python

→Fischbacher

3.1.1 Entwicklung

Die im Jahre 1991 von Guido van Rossum, einem niederländischen Software-Entwickler, veröffentlichte Programmiersprache Python wurde ursprünglich für das Betriebssystem Amoeba entwickelt und sollte die Programmier-Lehrsprache ABC ablösen. Mit der ersten Vollversion, die unter dem Namen Python 1.0 im Jahre 1994 erschienen ist, wurden einige Konzepte der funktionalen Programmierung implementiert, jedoch wurden diese später wieder aufgegeben. Im Jahre 2000 erschien Python 2.0 mit einer voll funktionsfähigen Garbage Collection, sowie die Unterstützung für den Unicode-Zeichensatz. Python 3.0 wurde 8 Jahre später am 3. Dezember 2008 veröffentlicht. Mit der Version 3.0 kamen tiefgreifende Änderungen an der Sprache, die dazu führten, dass sich die Python Software Foundation dafür entschied, Python 2.7 und Python 3.0 bis Ende 2019 parallel mit neuen Versionen zu unterstützen. Die neueste Version ist Version 3.7, die am 27. Juni 2018 erschienen ist.¹

3.1.2 Idee und Zweck

Python ist eine sehr übersichtliche und einfache Programmiersprache und war in erster Linie dafür gedacht, das Programmieren zu erlernen. Das wird erzielt, in dem Python mit wenigen Schlüsselwörtern auskommt und die Syntax, im Vergleich zu anderen Programmiersprachen, sehr reduziert ist. Außerdem ist die Standardbibliothek von Python überschaubar und leicht erweiterbar, was zu folge hatte, dass heute eine Vielzahl an Bibliotheken zu Verfügung stehen.²

3.1.3 Verwendung

Da Python über eine große Vielfalt an Bibliotheken verfügt, einfach zu programmieren ist und der Code leicht zu lesen ist, bot es sich für dieses Projekt hervorragend an. Beispielsweise für die Implementierung der Umsetzung mittels Graphviz, die durch die von Graphviz zu Verfügung gestellten Bibliothek mit dem gleichen Namen programmiert wurde. Ein anderes Beispiel wo dieses Projekt von Python profitiert hat, ist die Eigenschaft von Python,

¹ *History and License — Python 3.7.3rc1 documentation.*

² *General Python FAQ — Python 3.7.3rc1 documentation.*

dass es möglich ist, Python-Programme als Module in anderen Sprachen einzubetten.³

3.2 Pycharm als IDE

→Fischbacher

3.2.1 Allgemeines

3.2.1.1 Version

PyCharm ist eine IDE vom Unternehmen JetBrains, die im Juli 2010 erschienen ist und viele Eigenschaften sowie Features mit den anderen IDEs von JetBrains teilt. Dabei hat JetBrains wie der Name PyCharm schon andeutet, die IDE explizit für die Programmiersprache Python entwickelt. Die aktuellste Version ist die Version 2018.3.5, die am 27. Februar 2019 erschienen ist. Jedoch ist die neue Version 2019.1 schon in Entwicklung und wird noch im zweiten Quartal von 2019 verfügbar sein. In der Regel werden jedes Jahr ca. drei neue Versionen veröffentlicht, jedoch handelt es sich dabei eher um kleinere Updates oder Hotfixes und nicht um große Erneuerungen.⁴

3.2.1.2 Editionen

Außerdem gibt es PyCharm in drei verschiedenen Varianten, einmal die PyCharm Community Edition, die PyCharm Professional Edition und die PyCharm Educational Edition.⁵

- Die Community Edition ist Open-Source und damit gratis für jeden erhältlich.
- Die Professional Edition verfügt über mehr Features, man benötigt für diese Version jedoch eine Lizenz die man kaufen muss. Für dieses Projekt wurde die Professional Edition verwendet, da JetBrains die Lizenzen gratis für Schulen anbietet.
- Die dritte Edition ist zum Erlernen von Python gedacht.

3.2.2 Funktionen

3.2.2.1 Intelligenter Code Editor

PyCharm bietet intelligente Code Vervollständigung, Syntax Hervorhebung, automatische Code Refactorings. Die IDE erkennt wenn Code-Teile kopiert wurden und macht dem entsprechenden Vorschläge für das Refactoring.⁶

3.2.2.2 Web Development Frameworks

PyCharm unterstützt auch spezifische Web Development Frameworks wie z.B.:

- Django
- Flask
- Google App Engine
- web2py

³ IntegratedDevelopmentEnvironments - Python Wiki.

⁴ PyCharm.

⁵ Previous Releases - PyCharm.

⁶ Features - PyCharm.

3.2.2.3 Cross-technology Development

Neben Python unterstützt die IDE auch noch andere Sprachen wie z.B.: JavaScript, TypeScript, SQL und HTML/CSS.

3.2.2.4 Built-in Developer Tools

Debuggen, Testen und Profiling ist mit PyCharm durch eine Vielzahl an GUI Elementen einfach und schnell möglich. Automatisches Einsätzen auf einem entfernten Rechner kann einfach eingestellt werden und außerdem bietet die IDE eine schnelle und benutzerfreundliche GUI für Versionsverwaltungsprotokolle wie z.B.: Git, Mercurial und SVN.

3.2.3 Vorteile

→Fischbacher

- Ein Vorteilen der IDE ist der Support der mittels Forum oder Mail Kontakt direkt mit den Entwicklern erfolgt.
- Neben dem Support wird bei JetBrains auch die Benutzerfreundlichkeit ihrer IDE. Die einzelnen Funktionen sind übersichtlich von den anderen getrennt und in den Untermenü mit gut erkennbaren Symbolen gekennzeichnet.
- Der WSL Interpreter war bei dem Projekt eine große Hilfe, da nicht jeder auf dem selben Betriebssystem arbeitete. Dadurch konnte man selbst, wenn auf einem Rechner mit Windows gearbeitet wurde, in einer Linux Umgebung testen.
- Durch die große Community von PyCharm und JetBrains gibt es für jede IDE eine Vielzahl an Plugins, die nicht nur das Programmieren vereinfachen, sondern auch die Versionsverwaltung.
- Durch die REPL Python Konsole wird dem Entwickler das Testen vereinfacht, da man direkt von der IDE aus das Programm starten, testen und Fehlerbehebung kann. Außerdem wird einem der derzeitige Zustand der Variablen angezeigt.

⁷

⁷ Features - PyCharm.

3.3 XML

3.3.1 Allgemeines zu XML

→ Prinz

3.3.1.1 Definition

Das Datenformat *Extensible Markup Language (XML)* ist mittlerweile weit verbreitet und dient häufig als Basis für viele Technologien. XML wird im Buch „*Taschenbuch Datenbanken*“⁸ wie folgt definiert:

XML wurde mit der Zielsetzung des erleichterten Datenaustausches als Vereinfachung seines Vorgängerstandards *SGML* eingeführt. Inzwischen ist es aber viel mehr als das: Es bildet beispielsweise die Grundlage vieler Technologien der serviceorientierten Architektur. XML wird dabei zur Definition von Sprachen verwendet und legt gleichzeitig die Basissyntax für diese Sprachen fest.

3.3.1.2 Aufbau einer XML Datei

→ Prinz

Der Aufbau einer *XML*-Datei ist mit *XML-Elementen* versehen. Die einzelnen *XML-Elemente* sind selbst wählbar. Die Daten werden innerhalb dieser *Elemente* gespeichert.⁹

In einem *XML-Document* befindet sich der Code der eigentlichen Daten, jedoch kann auch eine *XML-Deklaration* angegeben werden. Diese ist jedoch optional. In dieser *Deklaration* befinden sich Informationen zu der Version des Datenformats sowie über die *XML-Attribute encoding* und *standalone*.⁹

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
```

- *version* gibt die aktuelle Version an, welche in dem Dokument verwendet wird. Standardmäßig beträgt dieser Wert 1.0.
- *encoding* gibt an, welche Kodierung der Zeichen benutzt werden soll. Standardmäßig gilt *UTF-8* als verwendete Kodierung.
- *standalone* gibt an, ob auf eine externe *Document-Type-Definition* zugegriffen werden muss, um korrekte Werte für bestimmte Teile zu ermitteln. Der Standardwert für dieses *XML-Attribut* ist mit *no* definiert.⁹

Sowohl *encoding* als auch *standalone* stellen optionale *XML-Attribute* dar. Falls diese nicht explizit angegeben werden, gelten die zuvor erwähnten Standardausprägungen.⁹

Im restlichen Dokument können dann beliebig viele *XML-Elemente* erstellt werden. Jedoch gilt zu beachten, dass *XML* eine gewisse Syntax verlangt. Unter anderem muss jeder Datensatz über einen Start-Tag und einen End-Tag verfügen.⁹

```
<Autor>
  <name> Andreas Prinz </name>
</Autor>
```

⁸ Thomas Kudraß. *Taschenbuch Datenbanken*. Deutsch. 2. Aufl. München: Carl Hanser Verlag, 2015.
URL: <https://www.hanser-fachbuch.de/buch/Taschenbuch+Datenbanken/9783446435087>.

⁹ ausarbeitung1.pdf. URL: <http://www.lgis.informatik.uni-kl.de/archiv/wwwdvs.informatik.uni-kl.de/courses/seminar/WS0203/ausarbeitung1.pdf> (besucht am 04.04.2019).

Bei der Namensgebung der *XML-Elemente* ist zu beachten, dass Start- und End-Tag gleich heißen. Groß- und Kleinbuchstaben sind in beliebigem Ausmaß erlaubt. Die Namen der *XML-Elemente* und *XML-Attribute* zwischen den Tags dürfen Unterstriche, Bindestriche, Punkte und alphanumerische Zeichen enthalten.⁹

```
<Autor>
  <name> Andreas Prinz </name>
  <geb_datum> 01.11.1999 </geb_datum>
</Autor>
```

Weiters können den *XML-Elementen* *XML-Attribute* hinzugefügt werden. Dabei ist zu beachten, dass eine Eigenschaft nur bei dem Start-Tag eingefügt werden kann. Der Wert wird dem *XML-Attribut* mittels einem Gleichheitszeichen zugewiesen. Der Inhalt muss jedoch von einfachen oder doppelten Anführungszeichen eingeschlossen sein. Ein Beispiel für ein Element mit einer Eigenschaft sieht wie folgt aus:

```
<diplomarbeit schuljahr="2018/19">
  <Autor> Andreas Prinz </autor>
<diplomarbeit>
```

In einem *XML-Dokument* können Namensräume vergeben werden. Diese gestalten die Datei in einem übersichtlicheren Format. Es besteht jedoch keine Pflicht, Namensräume zu verwenden, da sie optional sind. Häufig tritt der Fall ein, dass verschachtelte Namensräume zum Einsatz kommen, um unterschiedliche Teile einer Anweisung besser differenzieren zu können.¹⁰

3.3.1.3 Gültigkeit von XML-Dokumenten

→Prinz

Gültige *XML-Dokumente* entsprechen dem *XML-Standard* und verfügen über eine *DTD*. Unter einer *Dokumenttyp-Definition (DTD)* wird eine Grammatik für eine *XML-Datei* verstanden. In diesem Dokument befinden sich alle *XML-Elemente*, *XML-Attribute* und *XML-Datentypen*, welche in dem *XML-Code* verwendet werden dürfen. Weiters beinhaltet eine *DTD* den Kontext, in dem die *XML-Elemente* auftauchen. Eine weitere Eigenschaft einer *DTD* besteht darin, dass es die Möglichkeit gibt, die Anzahl eines *XML-Elementes* an einer bestimmten Stelle festzulegen.¹⁰

Dafür stehen 3 Operatoren zur Verfügung:

- '?' Das *XML-Element* darf maximal einmal vorkommen
- '+' Das *XML-Element* muss mindestens einmal vorkommen
- '*' Das *XML-Element* kann beliebig oft vorkommen

Neben einer *DTD* verfügt *XML* auch über ein *Schema*, mit dem weitere Regeln definiert werden können. Weiters kann mit Hilfe dieses *Schemas* ein *XML-Dokument* auf seine Gültigkeit überprüft werden. Ein großer Unterschied zu einer *DTD* liegt darin, dass das *XML-Schema* ebenfalls eine Datei vom Typ *XML* darstellt und über kein eigenes Datenformat verfügt. *XML-Schema* bietet im Gegensatz zu *DTD*, wo es nur einen Typ *names PCDATA* gibt, eine große Auswahl an Datentypen an, wie zum Beispiel *string*, *integer*, *float* oder *date*. Falls in den 44 bereits bekannten Typen nicht der richtige enthalten ist, besteht die Möglichkeit, aus einfachen Datentypen neue abzuleiten.¹⁰

→Prinz

¹⁰ ausarbeitung1.pdf. URL: <http://wwwlgis.informatik.uni-kl.de/archiv/wwwdvs.informatik.uni-kl.de/courses/seminar/WS0203/ausarbeitung1.pdf> (besucht am 04.04.2019).

Weiters kann eine Grammatik auf zwei verschiedene Arten beschrieben werden:

- RNC
- RelaxNG/RNG

Die zwei Grammatiken unterscheiden sich dadurch, dass *RNC* (*RelaxNG compact*) die kompakte Syntax von *RNG* aufweist und *RelaxNG* (*Regular Language Description for XML New Generation*) die genauere Beschreibung des *XML*-Dokuments ist.¹¹

Der unten abgebildete Code beschreibt eine *XML*-Datei mittels *RNC*, wobei es die *XML-Elemente* mit dem Namen *RelType* und *PartEnt* gibt. Diese haben jeweils *XML-Attribute* mit einem Datentyp, wobei dieser in geschwungenen Klammern hinter dem Namen eines Teilelements steht.

```
RelType = element rel {
    attribute to { xsd:string },
    attribute from { xsd:string }?,
    (PartEnt+)
}

PartEnt = element part {
    attribute ref { xsd:string },
    attribute min { MinCardType },
    attribute max { MaxCardType },
    attribute weak { xsd:boolean }?
}
```

Im Vergleich zu *RNC* ist die *RNG*-Grammatik schwerer zu lesen, da sie selbst auch ein *XML*-Dokument ist. Der Aufbau eines *RelaxNG*-Dokuments sieht wie folgt aus:

→Prinz

```
<define name="RelType">
    <element name="rel">
        <attribute name="to">
            <data type="string"/>
        </attribute>
        <optional>
            <attribute name="from">
                <data type="string"/>
            </attribute>
        </optional>
        <optional>
            <attribute name="qoute">
                <data type="boolean"/>
            </attribute>
        </optional>
        <oneOrMore>
            <ref name="PartEnt"/>
        </oneOrMore>
    </element>
</define>
```

¹¹ ausarbeitung1.pdf. URL: <http://wwwlgis.informatik.uni-kl.de/archiv/wwwdvs.informatik.uni-kl.de/courses/seminar/WS0203/ausarbeitung1.pdf> (besucht am 04.04.2019).

```

</oneOrMore>
</define>

<define name="PartEnt">
    <element name="part">
        <attribute name="ref">
            <data type="string"/>
        </attribute>
        <attribute name="min">
            <ref name="MinCardType"/>
        </attribute>
        <attribute name="max">
            <ref name="MaxCardType"/>
        </attribute>
        <optional>
            <attribute name="weak">
                <data type="boolean"/>
            </attribute>
        </optional>
    </element>
</define>

```

→ Prinz

Wie klar zu erkennen ist, kann die *RNC*-Datei um einiges einfacher gelesen werden. Weiters verfügt der Code über weitaus weniger Zeilen als der in dem *RelaxNG*-Dokument. In *XML* stehen dem Benutzer einige Möglichkeiten zur Verfügung, wie die Daten gespeichert werden sollen. Zum Beispiel kann eine relationale oder eine objektorientierte Datenbank eingebunden werden.¹²

3.3.2 XERML

3.3.2.1 Definition

Das Dateiformat *XERML* ist eine Erweiterung der Sprache *XML*, die von Dipl.-Ing. Günter Burgstaller erstellt wurde. Die Abkürzung steht für *Extensible Entity Relationship Modelling Language*. Wie bereits im Namen erwähnt wird, ist diese Erweiterung von Vorteil, wenn *Entity Relationship Diagramme* mittels *XML* beschrieben werden sollen.

3.3.2.2 Verwendung

Die Verwendung des Dateiformates *XERML* erleichtert die Beschreibung eines *ER-Diagrammes*, da das ganze Dokument besser strukturiert werden kann und somit die Informationen über das *ERD* leicht auszulesen sind. Jedoch ist bei *XERML* zu beachten, dass es nicht nur eine einzige Datei geben kann. Zusätzlich können für ein Dokument weitere Files erstellt werden, welche unter anderem den Inhalt der Datei mittels Datentypen beschreiben oder eine Übersetzung der Daten in eine andere Sprache beinhalten.

→ Prinz

¹² *ausarbeitung1.pdf*. URL: <http://wwwlgis.informatik.uni-kl.de/archiv/wwwdvs.informatik.uni-kl.de/courses/seminar/WS0203/ausarbeitung1.pdf> (besucht am 04.04.2019).

3.3.2.3 Aufbau der Dateien

Der Aufbau einer *XERML*-Datei gleicht in gewissen Ansichtspunkten dem eines *XML*-Dokuments. Die Syntax erlaubt ebenfalls, dass ein Kopf angegeben werden kann. Dieser muss jedoch nicht geschrieben werden, da er optional ist. Das *root Element* in *XERML* hat den Namen *erm*. Alle Daten, die das *Entity Relationship Diagramm* beschreiben, werden innerhalb dieses *XERML-Elements* definiert. Als erster Wert innerhalb des *erm*-Element muss sich der Name des Modells befinden. Dieser kann mittels den *XERML-Element title* angegeben werden, indem als *Attribut* der Name mittels *name=’ ’* gesetzt wird. Anschließend werden die *Entites* beschrieben. Dies geschieht indem ein *XERML-Element* mit dem Tag *ent* geöffnet wird. Innerhalb dieser Sektion wird zuerst der Name des *Entities* mittels *name* angegeben und anschließend können *Attribute* mit dem Element *attr* definiert werden. Diesen *Attributen* kann ebenfalls ein Name mittels *name* gegeben werden. Sobald alle *Entities* beschrieben wurden, werden die Beziehungen zwischen den *Entity-Typen* definiert. Die Beziehungen sind ähnlich wie die *Entities* aufgebaut, wobei die *XERML-Elemente* mit dem Tag *rel* beschrieben werden. Innerhalb einer Beziehung können die *Attribute from* und *to* angegeben werden. Weiters beinhaltet eine Beziehung das *Attribut part*, wobei innerhalb dieses Tags die Werte *ref*, *min*, *max* angegeben werden können.

→ Prinz

Ein Beispiel für den Aufbau einer *XERML-Datei* anhand des erzeugten Datenmodells *Fußball* sieht wie folgt aus (nur Teile des Datenmodells abgebildet):

```
<?xml version="1.0" encoding="utf-8"?>

<erm version="0.2">

<!-- Front Matter -->

<title name="Fussball"/>
<title name="soccer" lang="en"/>

<!-- Entity-Types -->

<ent name="mannschaft">
    <attr name="name" prime="true"/>
    <attr name="gründungsjahr"/>
    <attr name="adresse"/>
</ent>

<ent name="spiel">
    <attr name="spielort" prime="true"/>
    <attr name="datum" prime="true"/>
    <attr name="mannschaft_heim"/>
    <attr name="mannschaft_ausw"/>
    <attr name="schiedsrichter"/>
    <attr name="ergebnis"/>
</ent>

<!-- Relationship-Types -->
```

```

<rel to="spielt mit bei">
    <part ref="mannschaft" min="1" max="n"/>
    <part ref="spiel" min="1" max="n"/>
</rel>

</erm>

```

→ Prinz

Das oben gezeigte XERML-Dokument verfügt jedoch über keine Typdefinitionen. Dafür wird eine zusätzliche Datei erstellt, in welcher die Datentypen der einzelnen *Attribute* der *Entities* beschrieben werden. Das Dokument beginnt mit dem *Wurzelement desc*, in dem die aktuelle Version angegeben wird. Innerhalb dieses XERML-Elements werden die *Entities* mit dem XERML-Element *typdsc* beschrieben. Mittels dem XERML-Element *entref* kann auf ein bereits definiertes *Entity* referenziert werden. Die *Attribute* dieses *Entities* werden mittels dem XERML-Element *attr* beschrieben, wobei der Name mittels *name* und der Typ eines *Attributes* mittels *type* festgelegt werden können. Außerdem muss eine Klasse mittels *class* gesetzt werden.

Das dazugehörige Typen-Dokument für die XERML-Datei sieht wie folgt aus:

```

<desc version="0.2">

<typdsc entref="mannschaft">
    <attr name="name"           type="char" class="name"/>
    <attr name="gründungsjahr" type="integer" class="year"/>
    <attr name="adresse"        type="char" class="address"/>
</typdsc>

<typdsc entref="spiel">
    <attr name="spielort"       type="char" class="address"/>
    <attr name="datum"          type="date" class="date"/>
    <attr name="mannschaft_heim" type="char" class="name"/>
    <attr name="mannschaft_ausw" type="char" class="name"/>
    <attr name="schiedsrichter" type="char" class="name"/>
    <attr name="tore"            type="integer" class="number"/>
    <attr name="ergebnis"        type="char" class="name"/>
</typdsc>

</desc>

```

→ Prinz

Außerdem kann für das XERML-Dokument ebenfalls eine Übersetzungsdatei(Lokalisierung) existieren. Diese Datei enthält das *Wurzelement loc*, in welchem das Attribut *version* gesetzt werden kann. Innerhalb des XERML-Elements *loc* werden die Übersetzungen der *Entities* und Beziehungen angegeben. Mittels dem XERML-Element *entlo* werden die *Entities* beschrieben. In dem XERML-Element müssen die XERML-Attribute *entref*, *name-lo*, *lang* angegeben werden. Der Wert von *entref* ist das *Entity*, für das diese Übersetzung gilt, *name-lo* ist der Name des *Entities* in der Sprache, welche in *lang* angegeben wird. Innerhalb des XERML-Elements *entlo* können die Attribute des *Entities* mittels *attr* angegeben und mit *name-lo* übersetzt werden. Beziehungen werden mit dem XERML-Element *relo*

beschrieben, wobei dieses *XERML-Element* ebenfalls über die *XERML-Attribute* *relref*, *name-lo* und *lang* verfügt.

Diese Lokalisierungs-Datei ist für das bereits gezeigte *XERML* mit folgendem Inhalt gefüllt:

```
<loc version="0.2">

    <entlo entref="mannschaft"      name-lo="team" lang="eng">
        <attr name="name"           name-lo="name"/>
        <attr name="gründungsjahr"   name-lo="date_establishment"/>
        <attr name="adresse"         name-lo="adress"/>
    </entlo>

    <entlo entref="spiel"          name-lo="match" lang="eng">
        <attr name="svnr"           name-lo="ssn"/>
        <attr name="datum"           name-lo="dates"/>
        <attr name="mannschaft_heim" name-lo="home_team"/>
        <attr name="mannschaft_ausw" name-lo="guest_team"/>
        <attr name="schiedsrichter"  name-lo="referee"/>
        <attr name="tore"             name-lo="goals"/>
        <attr name="ergebnis"         name-lo="result"/>
    </entlo>

    <rello relref="spielt mit bei" name-lo="participates in" lang="eng"/>
</loc>
```

→Prinz

3.3.2.4 Vergleich mit XML

Wie bereits erwähnt wurde, verfügt das erstellte *XML-Vokabular XERML* hinsichtlich der Verwendung zur Beschreibung eines *Entity Relationship Diagrammes* über ein paar Vorteile gegenüber *XML*. Es wäre theoretisch in *XML* ebenfalls möglich, ein *ERD* zu beschreiben, jedoch ist diese Datei um einiges schwerer lesbarer und der Code ist ebenfalls nicht so strukturiert wie bei dem Vokabular *XERML*.

3.4 PIC-Code

→Prinz

3.4.1 Definition

Die Sprache *PIC-Code* ist eine Erweiterung von **troff**. Das Textsatzsystem **troff** erlaubt dem Benutzer einen qualitativ hochwertigen Text zu erstellen oder ein Diagramm zu zeichnen, wofür jedoch Erweiterungen benötigt werden.¹³ *PIC* stellt zusätzlich Features zur Verfügung, mit welchen zum Beispiel Boxen, Linien oder Ellipsen einfach dargestellt und verbunden werden können.

Falls in einem **troff**-Dokument beispielsweise ein Diagramm gezeichnet werden soll, ist dies ohne weitere Extras nicht möglich. Um dies umzusetzen, kann *PIC* in den Code eingebunden werden. Mithilfe der Erweiterung sollte es nun möglich sein, das gewünschte Objekt zu generieren, da *PIC* die Möglichkeiten bietet, Gegenstände zu zeichnen, zu verbinden oder zu färben.¹³

3.4.2 Aufbau einer PIC Datei

→Prinz

Der Aufbau einer PIC-Datei wird in Listing 3.1¹³ veranschaulicht:

Listing 3.1: Beispielaufbau einer PIC-Datei

```

1 .PS
2 ellipse "document";
3 arrow;
4 box "\fIpic\fP(1)"
5 arrow;
6 box width 1.2 "\fIgtbl\fP(1) or \fIgeqn\fP(1)" "(optional) dashed;
7 arrow;
8 box "\fIgtroff\fP(1)";
9 arrow;
10 ellipse "PostScript";
11 .PE

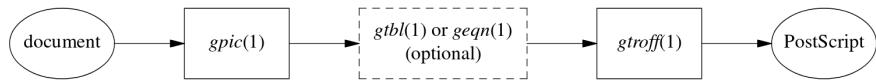
```

Der Aufbau von jedem *PIC* Programm ist im Prinzip gleich. Jede Datei muss zu Beginn des Codes ein **.PS** stehen haben, da dies dem Compiler kennzeichnet, dass ab diesem Zeitpunkt Befehle in der Sprache *PIC* zu erwarten sind. Falls der Eintrag **.PS** fehlt, so gibt der Compiler während des Ausführens einen Syntax-Fehler aus und weist darauf hin, dass der erforderliche Ausdruck fehlt.

Das Ende eines *PIC* Programmcodes ist mittels **.PE** gekennzeichnet. Das Fehlen dieses Eintrages trägt dazu bei, dass der Compiler kein Ende der Datei vorfindet und es tritt ebenfalls ein Syntaxfehler auf.

Die Ausgabe des oben gezeigten Codes sieht wie folgt aus (Abbildung 3.1)¹³:

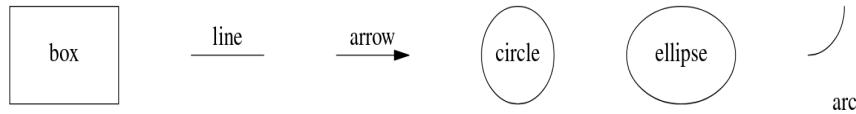
¹³ *Making Pictures With GNU PIC*. URL: <https://www.complang.tuwien.ac.at/doc/groff/html/pic.html> (besucht am 04.04.2019).

**Abbildung 3.1:** Erzeugter Graph des Beispielcodes

3.4.3 Objekte zeichnen in PIC

In *PIC-Code* ist es möglich verschiedene Objekte zu zeichnen. Hierbei gibt es vordefinierte Objekte, jedoch können auch neue Objekte generiert werden.

Abbildung 3.2¹⁴ zeigt die grundsätzlichen Objekte von PIC:

**Abbildung 3.2:** Vordefinierte Objekte

→Prinz

Die Form einer Raute ist standardmäßig nicht definiert, lässt sich jedoch leicht implementieren. Es besteht nämlich die Möglichkeit, den Rand der Raute mit Hilfe von verbundenen Linien zu zeichnen. Listing 3.2 veranschaulicht die Generierung einer Raute:

Listing 3.2: Code zum Zeichnen einer Raute

```

1 .PS
2 box invis;
3 line from last box .n to last box .e then
4 to last box .s then to last box .w then to
5 last box .n
6 .PE
  
```

Die generierte Raute ist in Abbildung 3.3 zu sehen.

¹⁴ *Making Pictures With GNU PIC*. URL: <https://www.complang.tuwien.ac.at/doc/groff/html/pic.html> (besucht am 04.04.2019).

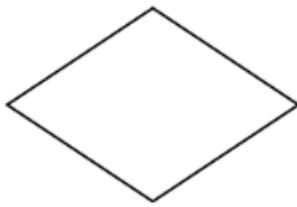


Abbildung 3.3: Gezeichnete Raute aus dem obigen Codebeispiel

PIC verfügt über eine große Auswahl an Möglichkeiten, um ein bestimmtes Objekt zu zeichnen. Man kann durch zusätzliche Argumente zum Beispiel die Breite und die Höhe einer Box festlegen, ob der Rahmen eine durchgezogene Linie oder nur strichiert dargestellt werden soll oder ob die Box in einer bestimmten Farbe zu generieren ist.

Der benötigte Code, um eine gelbe Ellipse zu zeichnen, wird in Listing 3.3 dargestellt:

→Prinz

Listing 3.3: PIC-Code zum Zeichnen einer gelben Ellipse

```

1 .PS
2 ellipse shaded "yellow";
3 .PE

```

Die gefärbte Ellipse wird in Abbildung 3.4 veranschaulicht:

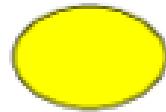


Abbildung 3.4: Gelbe Ellipse

→Prinz

In *PIC-Code* sind standardmäßig einige Farben vordefiniert. Es besteht jedoch auch die Möglichkeit, Farben mittels zum Beispiel hexadezimalen Werten zu definieren. Die Farbwerte können ebenfalls mit Hilfe von dem *RGB*-Modell angegeben werden. Um eine neue Farbe in *PIC-Code* zu definieren wird folgender Code benötigt (Listing 3.4):

Listing 3.4: Code zum Definieren einer neuen Farbe

```

1 .PS
2 .defcolor medblue rgb #89D8D6
3 .PE

```

Die definierte Farbe `medblue` kann so wie jede andere Farbe normal verwendet werden. Listing 3.5 zeigt einen Beispielaufgruf:

Listing 3.5: Anwendung der neu definierten Farbe

```

1 .PS
2   box shaded medblue;
3 .PE

```

Der Code liefert als Ergebnis eine Box in einem helleren Blauton. Die Grafik wird in Abbildung 3.5 veranschaulicht.

**Abbildung 3.5:** Box in der selbst definierten Farbe *medblue*

In ERDs müssen unter anderem auch *abhängige Entitytypen* gezeichnet werden. Diese werden dargestellt, indem die Umrandung des Objektes doppelt gezeichnet wird. Das betrifft Entities und Beziehungen. →Prinz

Der Code für eine Box mit doppelt gezeichneten Linien sieht wie folgt aus (Listing 3.6):

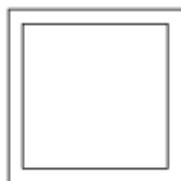
Listing 3.6: PIC-Code für eine Box mit doppelter Umrandung

```

1 .PS
2 boxht=1;boxwid=1;box at (0, 0);
3 boxht=1.2;boxwid=1.2; box at (0, 0);
4 .PE

```

Abbildung 3.6 zeigt das Ergebnis des oben angeführten Codes.

**Abbildung 3.6:** Box mit doppelter Umrandung

Für die Implementierung einer abhängigen Beziehung werden zwei Rauten übereinander gezeichnet, wobei eine der beiden Rauten über eine größere Höhe und Breite verfügt. In der Praxis sieht der generierte *PIC-Code* wie folgt aus (siehe Listing 3.7):

Listing 3.7: PIC-Code zur Generierung einer Raute mit doppelter Umrandung

```
1 .PS
```

```

2 boxht=1;boxwid=1;box invis at (0, 0);
3 line from last box .n to last box .e then to last box .s
4 then to last box .w then to last box .n;
5 boxht=1.2;boxwid=1.2; box invis at (0, 0);
6 line from last box .n to last box .e then to last box .s
7 then to last box .w then to last box .n;
8 .PE

```

→Prinz

Aus diesem Code wird folgende Grafik erstellt(siehe Abbildung 3.7):

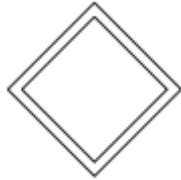


Abbildung 3.7: Raute mit doppelter Umrandung

Für die Implementierung einer *Super/Sub*-Beziehung muss die Form eines Dreiecks gezeichnet werden. Der benötigte Code wird in Listing 3.8 gezeigt:

→Prinz

Listing 3.8: PIC-Code zum Zeichnen eines Dreiecks

```

1 .PS
2 box invis;
3 line from last box .n to last box .se then to last box .sw
4 then to last box .n
5 .PE

```

Listing 3.8 hat folgende Ausgabe (siehe Abbildung 3.8) :



Abbildung 3.8: gezeichnetes Dreieck von dem abgebildeten Code

→Prinz

Ein Feature von *PIC*-Code ist, dass man mit Labels arbeiten kann. Darunter versteht man, dass Objekten eine bestimmte ID zugewiesen werden kann, um im nachhinein einfach darauf zuzugreifen. Dies kann unter anderem nützlich sein, wenn zwei bestimmte Elemente verbunden werden sollen, die nicht direkt nacheinander gezeichnet wurden. Dabei sind gewisse Syntaxregeln für ein Label einzuhalten. Es darf nämlich keine Leerzeichen enthalten und muss mit einem Großbuchstaben beginnen. Weiters dürfen keine Ziffern und Umlaute vorkommen.

→Prinz

Listing 3.9: Objekte mit Labels versehen und verbinden

```
1 .PS
```

```

2 A: box;
3 move;
4 B: circle ;
5 down;
6 move;
7 C: ellipse ;
8 line from A to C;
9 .PE

```

Das entstandene Bild aus Listing 3.9¹⁵ ist in Abbildung 3.9¹⁵ zu sehen.

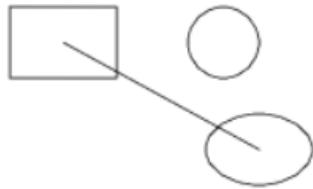


Abbildung 3.9: Objekte mit Labels versehen und verbinden

→Prinz

Um das Layout der gezeichneten Elemente zu bestimmen, gibt es mehrere Methoden. Wenn keine zusätzlichen Befehle angegeben werden, generiert *PIC* das erste Objekt in der linken oberen Ecke des Bildes und fügt die restlichen rechts daneben ein. Es gibt jedoch die Möglichkeit, das Layout der generierten Datei mittels Addieren oder Subtrahieren der Koordinaten zu verändern und somit den Standort der Graphen zu bestimmen. Eine weitere Art der Spezifizierung kann dadurch erreicht werden, indem x- und y-Werte direkt in den Befehl eingebunden werden. In Listing 3.10 ist ein Beispiel abgebildet, wie Objekte an bestimmten Koordinaten gezeichnet werden.

Listing 3.10:]PIC-Code zur Generierung einer Box an den Koordinaten [100, 100]

```

1 .PS
2 A:box at (100, 100);
3 .PE

```

Um den *Primary Key* von einem *Entity* zu bestimmen, muss der Inhalt des *Primary-Key*-Attributes unterstrichen werden. Dies wird in *PIC*-Code implementiert, indem 2 Ellipsen übereinander gezeichnet werden, wobei in einer Ellipse der Name des *Attributes* steht und in der zweiten Ellipse befinden sich anstatt eines Textes als Name mehrere aufeinanderfolgende Unterstriche. Der Code, um dieses Ergebnis zu erzielen, wird in Listing 3.11 veranschaulicht:

→Prinz

Listing 3.11:]PIC-Code zum Unterstreichen von Text in einer Ellipse]

```

1 .PS
2 ellipse "Unterstrichen";
3 ellipse at last ellipse "_____";
4 .PE

```

¹⁵ Making Pictures With GNU PIC. URL: <https://www.complang.tuwien.ac.at/doc/groff/html/pic.html> (besucht am 04.04.2019).

Aus Listing 3.11 wird folgende Grafik erzeugt (siehe Abbildung 3.10)):



Abbildung 3.10: Text der Ellipse wird unterstrichen

Weiters müssen die *Primary Keys* von *abzähligen Entities* mit einer strichlierten Linie unterstrichen werden. Der *PIC-Code* zur Darstellung eines Attributes mit einem strichliert unterstrichenem Text sieht wie folgt aus (Listing 3.12):

Listing 3.12: Text der Ellipse wird strichliert unterstrichen

```

1 .PS
2 ellipse "Unterstrichen";
3 ellipse at last ellipse "-----";
4 .PE
  
```

Der in Listing 3.12 gezeigte Code generiert folgende Grafik (siehe Abbildung 3.11)):



Abbildung 3.11: Text der Ellipse wird strichliert unterstrichen dargestellt

3.4.3.1 Anpassen der Objekte

In *PIC-Code* steht die Option zur Verfügung, dass Eigenschaften von Objekten wie zum Beispiel von Boxen, Ellipsen oder von Kreisen per Hand hinzugefügt werden können. Dies betrifft unter anderem die Höhe und die Breite. Weiters können Attribute für das ganze Dokument angegeben werden, wie zum Beispiel die maximale Höhe und Breite von dem gesamten Bild. Standardmäßig sind diese Werte bei Boxen und Ellipsen 0.5 cm und 0.75 cm. Der Radius für einen Kreis beträgt 0.25 cm, falls dieser nicht angegeben wird.

→Prinz

Um die Breite und die Höhe beispielsweise auf jeweils 4 Zentimeter zu setzen, muss der Code wie folgt aussehen (siehe Listing 3.13):

Listing 3.13: Setzen der Boxbreite und -höhe auf 4cm

```

1 .PS
2 quadrat: boxwid=4;boxht=4; box;
3 .PE
  
```

Es gibt ausserdem die Option, ein bestimmtes Objekt zu skalieren. Diese kann hilfreich sein, wenn die Grenzen des zu generierenden Bildes sehr groß sind. Listing 3.14 zeigt, wie der Skalierungsfaktor einer Box gesetzt werden kann:

Listing 3.14: Setzen des Skalierungsfaktors einer Box auf 15

```

1 .PS
2 scaledbox: scale=15; box;
3 .PE

```

3.4.4 Layout

→Prinz

Das Layout des ganzen Diagrammes wird mit Hilfe von `graphviz-layout` erzeugt. Der Python-Code für die Generierung der Koordinaten wird in Listing 3.15 gezeigt:

Listing 3.15: Python Code zur Generierung des Layouts mittels `graphviz-layout`

```

1 def nx_graph(rel):
2     erd = nx.Graph()
3     for curr in rel:
4         if len(curr) > 1:
5             for i in range(len(curr)):
6                 erd.add_edge(curr[0], curr[i])
7         else:
8             continue
9     pos = graphviz_layout(erd, prog='neato', args="--Gsplines=true --Goverlap=scale --Gsize
10 =100,200! --Gmaxiter=10000 --Gepsilon=0.00001 --Gdpi=1 --Gratio=fill" )
11 return pos

```

Der Befehl `graphviz_layout` verfügt über einige Parameter, die das zu generierende Layout bestimmen. Das Argument `prog` gibt den Algorithmus für das Layout an. Als gültige Parameter dieses Arguments gelten¹⁶:

→Prinz

- `dot` kann sinnvoll verwendet werden, wenn das gewünschte Ergebnis hierarchisch oder geschichtet dargestellt werden soll.
- `neato` produziert die Grafik gemäß des *Sprungfedermodells*. Dieser Parameter ist der Standard für ungerichtete Graphen.
- `twopi` erstellt für gerichtete und ungerichtete Graphen eine Darstellung, bei der die Knoten radial angeordnet werden.
- `fdp` beinhaltet einen Mehrgitterlöser zur Behandlung von großen und geclusterten Graphen.
- `sfdp` ist eine skalierbare Version von `fdp`. Diese Option eignet sich besonders gut für große Diagramme.
- `circo` ordnet die Knoten kreisförmig an. Dieser Parameter erweist sich als sinnvoll für bestimmte Diagramme von mehrfach zyklischen Strukturen.

Jeder dieser Layoutalgorithmen hat seine Vorteile, jedoch bietet `neato` angewendet auf alle Datenmodelle das bestgeeignete Layout für ein *Entity Relationship Diagramm* mit den wenigsten Überschneidungen der Linien.

¹⁶Graphviz - Graph Visualization Software. URL: <https://www.graphviz.org/> (besucht am 04.04.2019).

Es können weitaus mehr Layoutalgorithmen angegeben werden, jedoch sind die oben aufgezählten Parameter am Sinnvollsten beziehungsweise erstellen diese Algorithmen ein *Entity Relationship Diagramm* in einem gut lesbaren Layout.

Die Argumente des Parameters `args` sind Zusatzeinstellungen, die für die Generierung der Grafik nützlich sind. Unter anderem kann als Attribut `-Gsize` angegeben werden um die Größe des Dokuments zu bestimmen. Weitere Argumente sind zum Beispiel `-Gsplines`, `-Goverlap`, `-Gmaxiter`, `-Geplison`, `-Gdpi` und `-Gratio`.

3.4.5 Erstellen einer PIC-Datei

→ Prinz

Der *Python*-Code, indem die *PIC*-Datei erstellt und beschrieben wird, sieht wie folgt aus (Listing 3.16):

Listing 3.16: Python Code, indem der generierte PIC-Code in eine selbsterstellte PIC-Datei geschrieben wird

```

1 def write_erd_to_file(pic):
2
3     if os.path.exists("erd.pic"):
4         os.remove("erd.pic")
5     file = open("erd.pic", "w")
6     file.write(pic)
7     file.close()
8     print('Your file is created at: ' + os.getcwd() + '/erd.pic')

```

Der Parameter `pic` ist der generierte *PIC*-Code, der das *ERD* zeichnet.

Sobald in das erstellte Dokument der *PIC*-Code ergänzt wurde, kann die Datei kompiliert und angezeigt werden.

3.4.6 Vorteile und Nachteile von PIC

→ Prinz

3.4.6.1 Vorteile

Die Erweiterung *PIC* bietet einige Vorteile, die für die Generierung von Dokumenten und Graphen hilfreich sein können:

- Da die Sprache sehr einfach aufgebaut ist und die Namen der Befehle bzw. Objekte sprechend gewählt wurden, fällt es einem späteren Leser nicht schwer, den Code zu verstehen. Vor allem mit Hilfe der vordefinierten Objekte wird das Generieren grundlegender Graphen um einiges vereinfacht und erfordert keinen großen Aufwand.

Listing 3.17: Leicht lesbarer Code zur Generierung einer Grafik

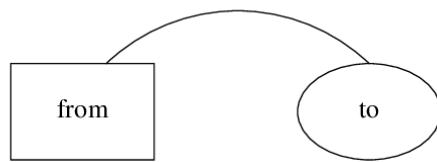
```

1 .PS
2 box "from"
3 move 0.75;
4 ellipse "to";
5 arc cw from 1/3 of the way \
6 between last box .n and last box .ne to last ellipse .n;
7 .PE
8

```

Aus Listing 3.17¹⁷ wird folgende Grafik erzeugt (sieht Abbildung 3.12¹⁷):

¹⁷ Making Pictures With GNU PIC. URL: <https://www.complang.tuwien.ac.at/doc/groff/html/pic.html> (besucht am 04.04.2019).

**Abbildung 3.12:** Ergebnis des Codes

- Weiters kann man dank der Fehlermeldungen und des leicht zu lesenden Codes sowohl syntaktische als auch logische Fehler ohne großem Aufwand finden und beheben.

3.4.6.2 Nachteile

→Prinz

Neben den Vorteilen gibt es jedoch auch Nachteile, die im Vergleich zu den Pro-Argumenten überwiegen. Zu den negativen Aspekten zählen:

- Die Anzahl der standardmäßig definierten Objekte in *PIC* ist nur gering. Daraus folgt, dass der Aufwand andere Objekte zu zeichnen steigt, auch wenn das nur geringfügig bzw. der Code leicht zu generieren ist. *PIC* verfügt zwar über die Möglichkeit, Makros zu definieren, jedoch ist die Definition dieser Makros ein Aufwand, der bei den anderen Implementierungsvarianten nicht existiert, da dort die benötigten Objekte bereits vordefiniert sind.

→Prinz

- Weiters ist über *PIC*-Code zu sagen, dass das Verbinden von zwei unterschiedlichen Objekten in der Theorie recht einfach ist, praktisch jedoch nur mit viel Aufwand verbunden umgesetzt werden kann. Es müssen nämlich die Bezugspunkte (Norden, Osten, Süden, Westen) angegeben werden, denn ohne diese Information befinden sich der Beginn und das Ende der gezeichneten Linie in der Mitte des jeweiligen Objektes. Falls dieser Gegenstand eine Beschriftung aufweist, hat dies zur Folge, dass der Inhalt der Box schwerer bis nicht mehr lesbar wird. Dies ist insofern ein Nachteil, da das Verbinden der Objekte viele Abfragen benötigt, weil der Start- und der Endpunkt nicht sofort bekannt sind. Die Koordinaten der zwei zu verbindenden Elemente müssen zuerst verglichen werden, um dann die richtigen Bezugspunkte zu wählen. Ohne diese Abfragen sieht die Verbindung zwischen zwei Boxen aus wie in Abbildung 3.13 zu sehen ist.

**Abbildung 3.13:** Konnektoren ohne zusätzlicher Information über Start- und Endpunkt

PIC-Code unterscheidet sich in diesem Punkt von *Graphviz*, *Libre Office Draw* und *Graphml*. Bei den Varianten werden die Bezugspunkte automatisch gesetzt.

- Die Generierung des Layouts des ganzen Diagrammes stellt generell ein großes Problem dar. Mittels *PIC*-Code ist es nicht möglich, ein gut aussehendes *ERD* zu er-

zeugen. Daher muss mit Hilfe von *Graphviz* das Layout im vorhinein generiert werden, wobei nur die Koordinaten der Knoten erzeugt werden und nicht das ganze Diagramm. Ausgehend von diesen Werten können dann die Objekte einfach im *PIC*-Code positioniert werden (siehe Kapitel 3.4.4). Auf Grund dieser Methode tritt ein weiteres Problem auf und zwar, dass man beim Generieren der Koordinaten die richtigen Parameter für den Befehl `graphviz_layout` benötigt. Diese müssen mit dem Dokument übereinstimmen, da sonst die Werte der Koordinaten zu groß werden und deswegen Elemente ausserhalb des Dokuments gezeichnet werden. Außerdem erzeugt dieser Befehl die generierten Punkte in einem viel zu großen Ausmaß, sodass die Werte trotz richtigen Parametern noch bearbeitet werden müssen.

- Bei der Generierung des Layouts treten häufig Probleme auf. Das liegt daran, dass PIC die Koordinaten in *inches* berechnet, jedoch eine PDF Datei mit Pixel arbeitet. Dies erschwert das Generieren mittels absoluten Koordinaten, da von PIC nicht überprüft wird, ob der Wert noch innerhalb des anzeigbaren Bereiches liegt oder ob dieser ausserhalb des gegebenen Bereiches gezeichnet wird. Dies kann zur Folge haben, dass ein Objekt teilweise abgeschnitten wird, wie in der Abbildung 3.14 zu sehen ist.

Listing 3.18: PIC-Code, um den Fehlerfall zu zeigen, dass eine Box bei zu großen Koordinaten abgeschnitten wird

```

1 .PS
2 box at (0, 300);
3 .PE
4

```

Listing 3.18 wird in Abbildung 3.14 veranschaulicht:



Abbildung 3.14: Box wird wegen zu großen Koordinaten abgeschnitten

Wenn die Koordinaten zu groß werden, kann es auch zu dem Fall kommen, dass diese nicht einmal mehr in der Nähe der maximalen Grenzen liegen. Das hat zur Folge, dass das Element nicht mehr angezeigt werden kann.

Listing 3.19: Code, um den Fehlerfall zu erzeugen, dass eine Box ausserhalb des sichtbaren Bereiches gezeichnet wird

```

1 .PS
2 box at(500, 0)
3 boxwid=35;boxht=15; bot at (0, 0);
4 line from 1st box to last box;
5

```

Die generierte Grafik wird in Abbildung 3.15 verbildlicht.

→Prinz



Abbildung 3.15: Box wird wegen zu großen Koordinaten ausserhalb des Sichtfeldes gezeichnet

3.5 yEd

Das Programm yEd ist ein Graph Editor, mit dem die verschiedensten Diagrammtypen erstellt werden können. Die Applikation wurde von dem deutschen Unternehmen yWorks entwickelt und basiert auf der Java-Bibliothek yFiles. Dadurch punktet der Editor nicht nur mit den Layout-Algorithmen und den Analyse-Tools die durch die Bibliothek zur Verfügung gestellt werden sondern auch mit einem übersichtlichen User-Interface, dass das erstellen und bearbeiten von Diagrammen vereinfacht.

→ Passet

3.5.1 Varianten und Installation

Um die Anwendung zu nutzen gibt es zwei Möglichkeiten. Die erste wäre den „yEd Live“-Dienst in Anspruch zu nehmen. Dabei handelt es sich um eine Online-Variante. Der Nachteil dieses Services ist, dass die Verarbeitung und auch Darstellung der Daten im Hintergrund anders ist als bei der Offline-Version. Dies führt unter anderem dazu, dass die erzeugten GraphML-Dateien bei „yEd Live“ nicht richtig bis hin zu gar nicht angezeigt werden.

Bei der zweiten Methode handelt es sich um die Offline-Version. Diese kann über die offizielle Internetseite von yWorks für das entsprechende Betriebssystem heruntergeladen und danach entsprechend installiert werden.

Wichtig ist es bei der Auswahl der Datei für die Installation darauf zu achten, ob diese auch die benötigte Java JRE beinhaltet und wenn nicht, ob man selbst auf dem Rechner die dazugehörige installiert hat. Wenn man sich dann im Installationsprozess befindet kommt man an einen Punkt, an dem man nach einem Installationspfad gefragt wird. Hierbei muss man darauf achten das Programm an einem Ort zu installieren zu dem man als Benutzer auch Zugriff hat, weil das Command-Line-Tool bei der Erstellung eines ER-Diagrammes anbietet, die generierte Grafik, sofern es sich um eine GraphML Datei handelt und man auf Linux arbeitet, direkt von der Kommandozeile heraus in yEd zu öffnen.

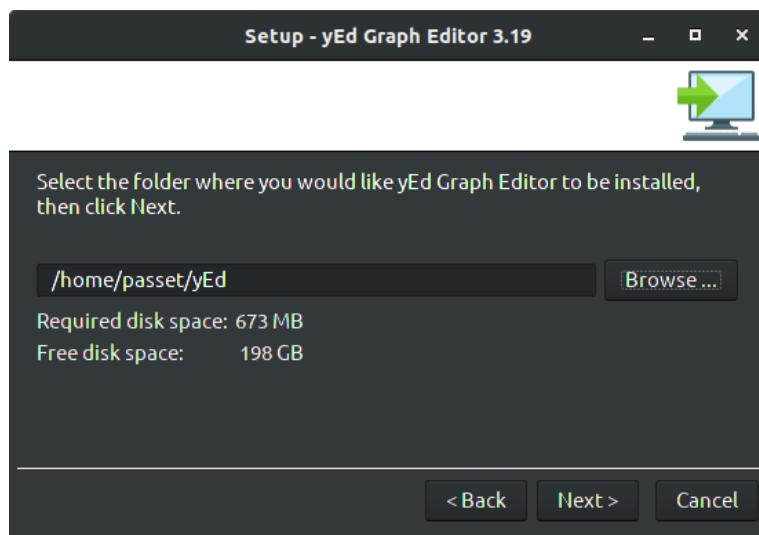


Abbildung 3.16: Angabe des Installationspfades im Installer von yEd 3.19

3.5.2 Oberfläche

→Passet

Das grafische Interface mit dem der Benutzer interagiert entspricht zwar nicht den modernsten Design Ideen aber bietet alles was man braucht. Grundlegend kann man die Oberfläche des Editors in 7 Bereiche unterteilen.

3.5.2.1 Menü

Den Kopf der Oberfläche bildet das Menü. Hier kann der Benutzer verschiedene Ansichten und Funktionen aktivieren bzw. deaktivieren. Es umfasst die Menüs Datei, Bearbeiten, Ansicht, Layout, Werkzeuge, Gruppierung, Fenster und Hilfe.

3.5.2.2 Palette

In der Palette befinden sich die Elemente mit denen man einen Graphen erstellen kann. Diese Komponenten sind in Gruppen aufgeteilt um für einen besseren Überblick zu sorgen. Zu Beginn bietet der Editor einem die folgenden Gruppierungen mit den entsprechenden Elementen:

- Geometrische Knoten
- Moderne Knoten
- Kantentypen
- Gruppenknoten
- Swimlane- und Tabellenknoten
- Personen
- Computer-Netzwerk
- UML
- Flussdiagramm
- BPMN
- Entity Relationship
- SBGN
- Aktuelle Elemente

Es gibt auch für den Benutzer die Möglichkeit selbst eine Gruppe mit Elementen zu erstellen und diese dann auch in die Palette einzubinden¹⁸.

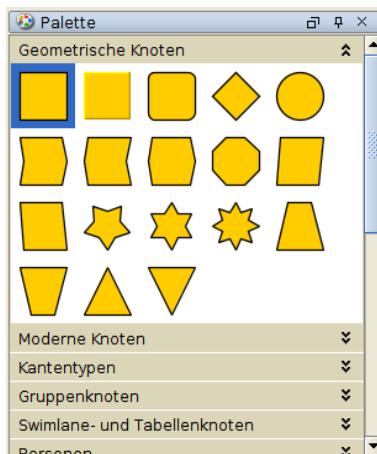


Abbildung 3.17: Palette in yEd bei Programmstart

3.5.2.3 Eigenschaften

→ Passet

Der Bereich Eigenschaften zeigt die charakteristischen Werte des momentan ausgewählten Elementes an. Hier können allerlei verschiedener Veränderungen vorgenommen werden wie zum Beispiel das Ändern der Beschreibung, Größe oder auch Farbe, vorausgesetzt natürlich das diese änderbar sind. Die Merkmale in diesem Bereich werden in die Gruppen Allgemein, Beschriftung, Daten und eine für das Element spezifische Gruppe aufgeteilt. In der Abbildung 3.18 wäre dann die spezifische Gruppe „Entity Relationship“.

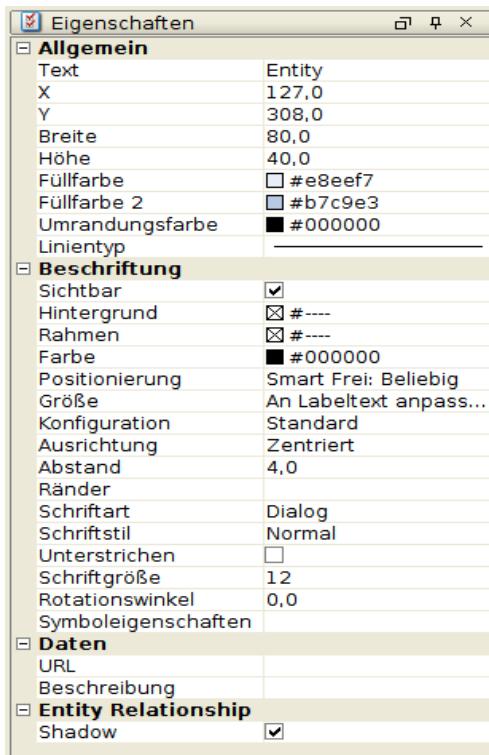


Abbildung 3.18: Eigenschaften eines Entitytypen in yEd

Sollte man kein Element seines Graphen ausgewählt haben, sieht man in diesem Bereich die Gesamtzahl an Knoten und Kanten.

3.5.2.4 Nachbarschaft

Auf der linken Seite der GUI befinden sich Bereiche die einem Auskunft über den Aufbau des aktuellen Graphen geben. Darunter befindet sich der Bereich Nachbarschaft. In diesem Fenster sieht man demnach alle Nachbar-Knoten des momentan ausgewählten Knoten. Zudem bietet es Registerkarten an, in denen nur die Knoten innerhalb der Gruppe, Vorgänger oder Nachfolger angezeigt werden.

Dies kann sich bei einem Entity-Relationship Diagramm als großen Vorteil entpuppen, da mit einem Klick herausgefunden werden kann mit welchen Beziehungstypen der Entitytyp in relation steht. In der Abbildung 3.19 wird die Nachbarschaft des Entitytypen Abteilung dargestellt.

¹⁸yWorks. *yEd Graph Editor Manual*. Englisch. URL: <https://yed.yworks.com/support/manual/index.html>.

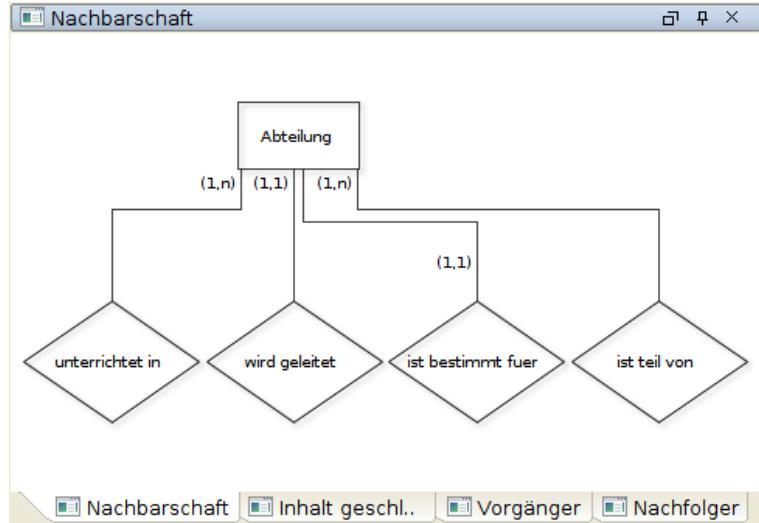


Abbildung 3.19: Nachbarschaft des Entitytypen Abteilung aus dem Datenmodell Schulinformationssystem

3.5.2.5 Struktur

→ Passet

Im wesentlichen beinhaltet dieser Bereich eine Liste mit all den Elementen die sich zum momentanen Zeitpunkt auf der Hauptoberfläche von yEd befinden. Sie ist hierarchisch geordnet, weshalb Teile des Graphen die zuerst erstellt wurden weiter oben angeführt sind als Teile die später hinzugefügt wurden.

Das Fenster Struktur ist eng mit den zuvor genannten Bereichen verbunden. Wählt man eine der Komponenten aus wird sie im Graphen markiert, die Eigenschaften dargestellt und die Nachbarschaft aufgezeigt.

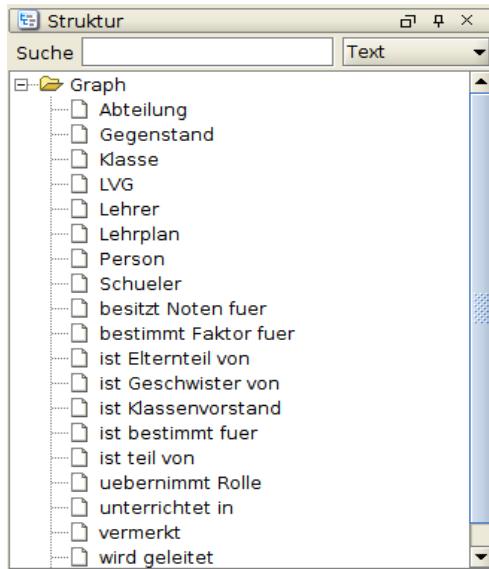


Abbildung 3.20: Struktur des ER-Diagrammes Schulinformationssystem

3.5.2.6 Editor-Bereich

→ Passet

In diesem Bereich wird der Graph dargestellt und durch Elemente aus der Palette ergänzt und verändert. Dies geschieht durch die Verwendung einer Maus oder durch drücken bestimmter Tasten auf der Tastatur. Die folgenden Aktionen und noch weiter nicht angeführte können auf diese Weise ausgeführt werden:

- Einen neuen Knoten erstellen und auswählen
- Eine Kante erstellen und auswählen
- Knoten verschieben
- Eine neuen Biegepunkt auf der Kante erstellen und verschieben
- Ein Element löschen
- Eine Beschriftung erstellen, auswählen, bearbeiten und verschieben

Die folgende Abbildung zeigt einen von yEd generierten Graphen den man ohne weiteres durch die Verwendung der eben genannten Aktionen erstellen kann.

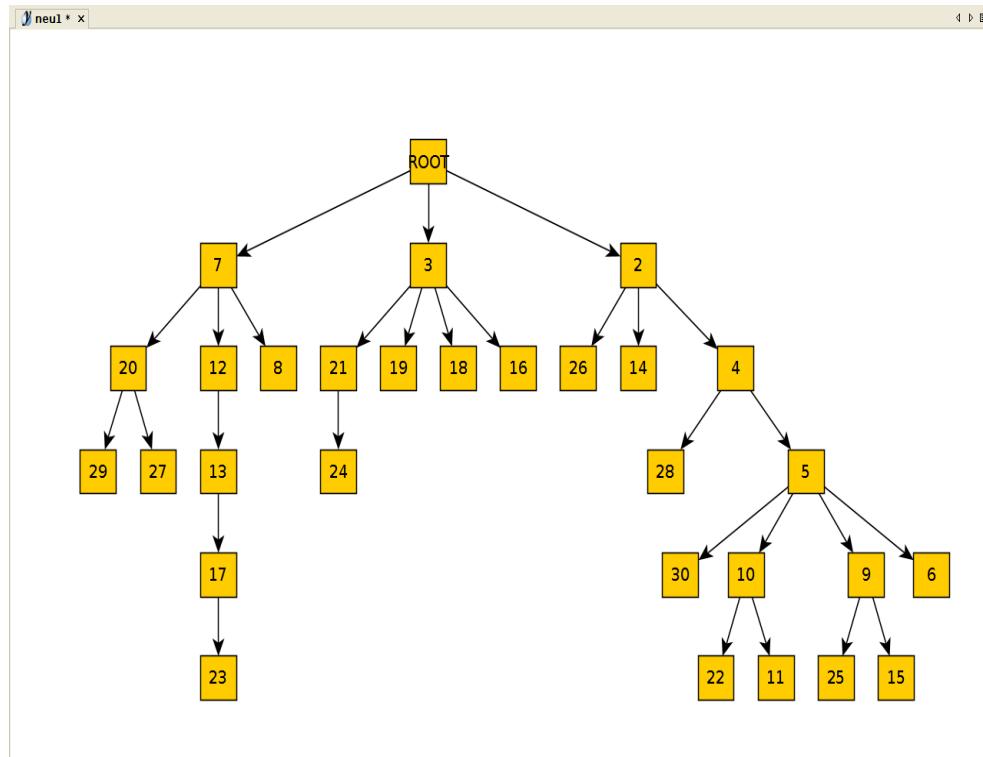


Abbildung 3.21: Beispiel Graph generiert von yEd

3.5.2.7 Übersicht

Den letzten Bereich der Oberfläche bietet die Übersicht. Hier wird der gesamte Graph dargestellt. Sollte der Editor-Bereich nicht den ganzen Graphen anzeigen wird in der Übersicht durch eine weiße Form gezeigt was gerade im Editor-Bereich angezeigt wird. Diese Form kann mit dem Mauszeiger durch anklicken und bewegen verschoben werden. Auf diese Weise kann man im Editor-Bereich navigieren.

3.5.3 Editier-Hilfen

Um den Benutzer bei der Anordnung der Elemente seines Graphen zu unterstützen, bietet yEd 3 Features.

3.5.3.1 Snap Lines

Im *yEd Graph Editor Manual* wird das Grundkonzept der „Snap Lines“ , zu deutsch Hilfs-

→Passet

linien, wie folgt beschrieben:

Snap Lines help you to interactively create a graph with an appealing layout.

Zusammenfassend lässt sich also sagen, dass die „Snap Lines“ für ein ansprechendes Layout sorgen. Diese Linien werden angezeigt während man ein ausgewähltes Element bewegt. Dabei rastet die Komponente an den Stellen ein wo es gut aussehen könnte und zeigt durch eine Linie an, weshalb es dort eingeraстet ist.

In der Abbildung 3.22 sieht man zum Beispiel, dass das ausgewählte Element an dieser Stelle einrastet weil es sich auf der selben Höhe befindet wie das Element das nebenan platziert ist.



Abbildung 3.22: „Snap Line“ hilft bei der zentralen Positionierung

Des Weiteren lassen die „Snap Lines“ einen erkennen, ob ein Knoten der zwischen 2 anderen Knoten positioniert wird sich mittig befindet oder ob ein Knoten die selbe Höhe oder Breite hat als ein anderer Knoten.

3.5.3.2 Grid

Eine weitere Hilfestellung für ein gutes Layout ist das Grid, zu deutsch Gitter, das man über die Menüleiste einschalten kann. Hat man es eingeschaltet, rasten die Elemente die man aus der Palette zieht an einem der fest vorgegebenen Punkte ein.

3.5.3.3 Orthogonale Kanten

Das letzte Feature das Abhilfe bei der händischen Anordnung der Komponenten schafft sind die orthogonalen Kanten. Hat man diese Funktion über die Menüleiste von yEd aktiviert, dann werden Kanten nur noch mit 90° Winkeln erstellt. Das heißt, dass es keine schiefen Linien mehr gibt, sondern der Editor von sich aus die Linien horizontal bzw. vertikal zeichnet bis die gewünschten Knoten miteinander verbunden sind, wie in Abbildung 3.23 dargestellt wird.

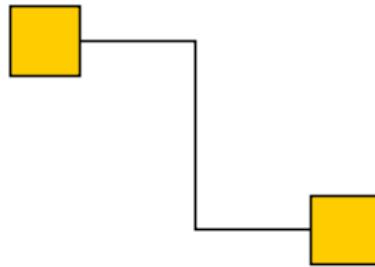


Abbildung 3.23: Beispiel einer orthogonalen Kante in yEd

3.5.4 Layout-Algorithmen

Einer der größten Vorteile die yEd besitzt, sind die Layout-Algorithmen die der Editor einem zur Verfügung stellt. Insgesamt kann man zwischen 19 verschiedenen auswählen. Diese sind unter anderem:

→ Passet

- Hierarchisch
- Organisch
- Orthogonal
 - Klassisch
 - UML-Stil
 - Kompakt
- Kreisförmig
- Baumartig
 - Gerichtet
 - Ballon
- Radial
- Serien-Parallel
- Swimlane
 - Hierarchisch
 - Organisch
 - Tabellarisch
- Flowchart
- BPMN
- SBGN
- Stammbaum
- Tree-Map
- One-Click Layout

Der Benutzer hat natürlich die freie Wahl welches Layout er für sein Diagramm haben möchte aber im yWorks, *yEd Graph Editor Manual* sind Empfehlungen gelistet welcher Algorithmus für welchen Diagrammtyp am besten geeignet ist. In den folgenden Unterkapiteln werden jene Layouts näher beschrieben die für ein ER-Diagramm geeignet sind.

3.5.4.1 Orthogonales Layout

Für Entity-Relationship-Diagramme wird das *Orthogonale Layout* empfohlen. Davon stehen 3 Varianten zur Auswahl:

- Klassisch
- UML-Stil
- Kompakt

Jede dieser Methoden verfügt über unterschiedliche Parameter die man als Benutzer festlegen kann und die anschließende Darstellung sichtlich beeinflussen. In der nachfolgenden

Abbildung 3.24 wird ein Beispiel von einem Datenmodell gezeigt, dessen Elemente mithilfe des klassischen Orthogonalen Layout angeordnet wurden.

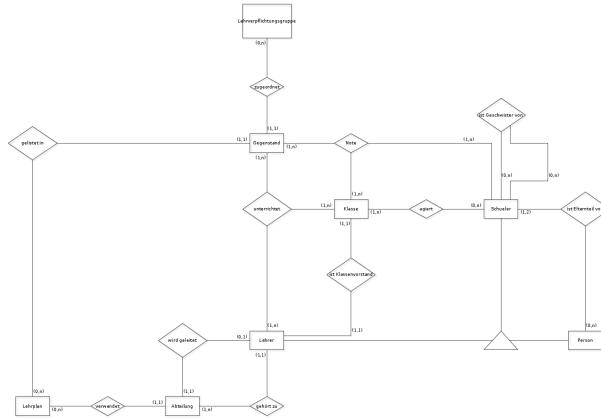


Abbildung 3.24: Beispiel für die Anordnung durch klassisches Orthogonales Layout

Wie man sehen kann sind die Kanten alle gerade weshalb es schön anzusehen und auch lesbar ist. Zeichnet man jedoch auch die Attribute in das Diagramm werden wie in Abbildung 3.25 ersichtlich, viele Kanten nah aneinander gelegt, was das Diagramm klobig erscheinen lässt.

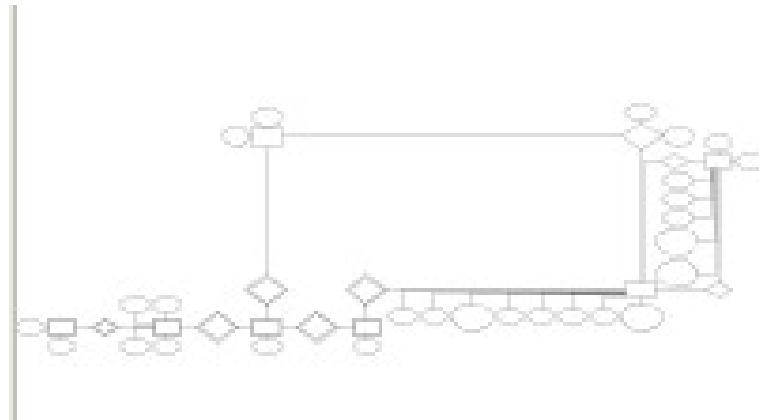


Abbildung 3.25: Orthogonales Layout mit Attributen

3.5.4.2 Kreisförmig

Im Zuge der Diplomarbeit wurde auch das *Kreisförmige Layout* näher untersucht. Wie der Name schon preisgibt, werden hierbei die Knoten in einem oder mehrere Kreise angeordnet abhängig von den Einstellungen. Bei diesem Layout werden 4 Stile unterschieden:

- BCC Kompakt
 - BCC isoliert
 - Benutzerdefinierte Gruppen

- Einzelner Kreis

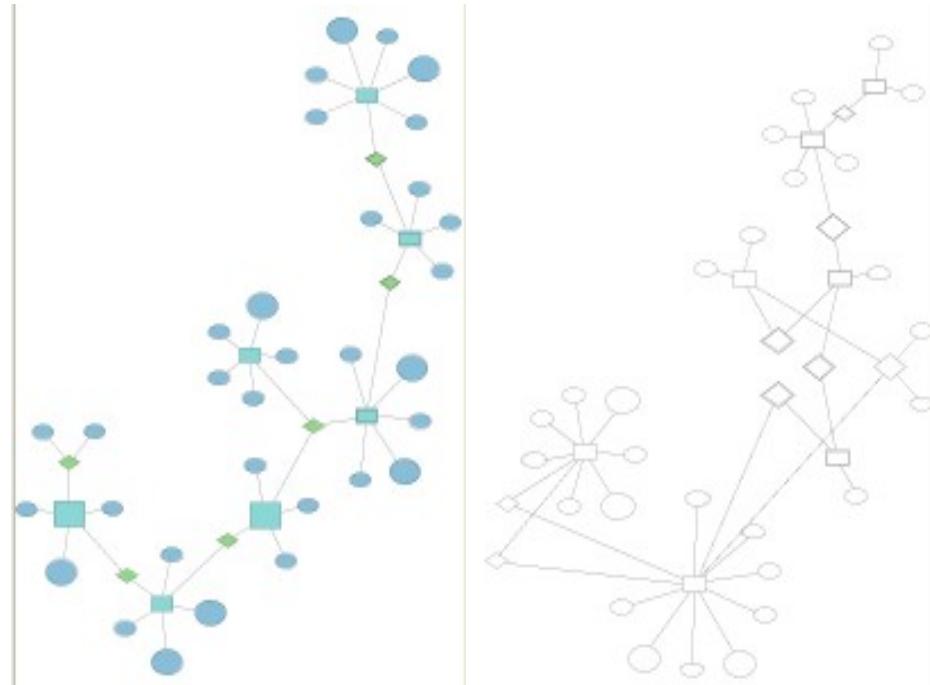
Für ein ER-Diagramm könnte man die Stile BCC Kompakt und BCC isoliert in betracht ziehen. Bei BCC Kompakt repräsentieren die Teile des Diagramms *biconnected* Komponenten. Diese Komponente wird laut *yEd Graph Editor Manual* wie folgt definiert:

A biconnected component consists of nodes that are reachable by two edge disjoint paths.

Sollte ein Knoten die Möglichkeit haben zu zwei Komponenten zu gehören wird dieser vom Algorithmus einer Komponente exklusiv zugewiesen.

BCC isoliert wendet das selbe verfahren an wie BCC Kompakt. Kommt es hierbei jedoch zu dem Fall, dass ein Knoten zu zwei Komponenten dazugehören könnte wird dieser als eigene Komponente im Diagramm dargestellt¹⁹.

Wie man an den Abbildungen 3.26(a) und 3.26(b) erkennen kann, ist es möglich ein ER-Diagramm mit diesem Algorithmus gut darzustellen bzw. es mit wenigen Handgriffen über die Editor-Oberfläche zu verschönern. Jedoch kann sich, abhängig von dem jeweiligen Datenmodell, ein Diagramm wie in Abbildung 3.27 entstehen.



(a) Datenmodell Fluggesellschaft mit BCC Kompakt (b) Datenmodell Fluggesellschaft mit BCC isoliert

Abbildung 3.26: Datenmodell Fluggesellschaft in BCC Kompakt und BCC isoliert

¹⁹yWorks, *yEd Graph Editor Manual*.

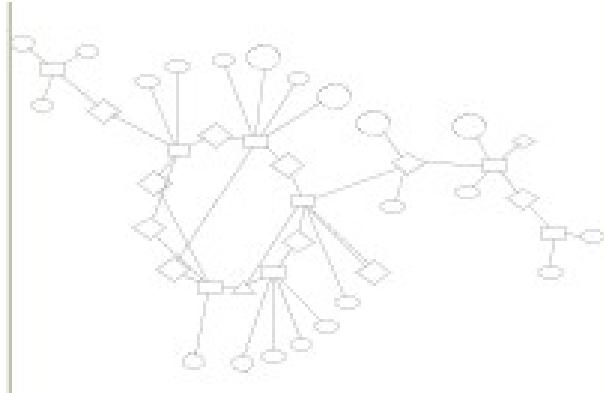


Abbildung 3.27: Datenmodell Schulinformationssystem mit BCC Kompakt

3.6 GraphML

GraphML ist ein einfaches Dateiformat das für die Erstellung von Graphen genutzt werden kann. Dabei handelt es sich um eine Grundlage die die strukturellen Eigenschaften eines Graphen beschreiben kann. Die unterstützten Grundlagen laut Group, *The GraphML File Format* sind:

- gerichtete, ungerichtete und gemischte Graphen
- Hypergraphen
- hierarchische Graphen
- Referenzen auf externe Daten
- Applikationsspezifische Attribut-Daten und
- light-weight Parser

Das Dateiformat basiert auf XML was es einfach macht damit Graphen zu generieren, archivieren und entwickeln.

3.6.1 Hintergrund

Die Entwicklung von GraphML während eines Workshops auf der Graph Drawing Symposium im Jahre 2000 in Williamsburg. Ein erster Ansatz für das Endergebnis wurde auf der Graph Drawing Symposium, 2001 in Wien vorgestellt²⁰.

Seither wurden laufend Erweiterungen erstellt und auch Anwendungen die auf diesem Dateiformat aufbauen. Eine dieser Anwendungen ist das in Kapitel 3.5 beschriebene Programm yEd. Die folgenden Kapitel beziehen sich auf das modifizierte GraphML Format für yEd.

3.6.2 Grundelemente und Aufbau

Aufgrund der Eigenschaft, dass es sich bei GraphML um ein XML-Dokument handelt ist der Aufbau übersichtlich und man kann mit nur vier Elementen einen ganzen Graphen beschreiben²¹. Zu diesen vier Elementen gehören:

²⁰GraphML Working Group. *The GraphML File Format*. 2007. URL: <http://graphml.graphdrawing.org>.

²¹Ulrik Brandes, Markus Eiglsperger und Jürgen Lerner. *GraphML Primer*. URL: <http://graphml.graphdrawing.org/primer/graphml-primer.html>.

- Das *graphml*-Element, dass das *root*-Element des Graphen bildet.
- Das *graph*-Element, dass den Graphen kennzeichnet.
- Das *node*-Element, dass einen Knoten des Graphen widerspiegelt.
- Das *edge*-Element, dass die Kanten zwischen den Knoten repräsentiert.

Innerhalb des *graph*-Elements sind die Elemente für die Knoten und Kanten des Graphen angesiedelt. Hierbei muss aber nicht auf die Reihenfolge der Elemente geachtet werden.

3.6.3 Generierung der Datei und Aufbau

Um einen GraphML-Datei für yEd zu erstellen bietet yWorks, die Entwicklungsfirma der Applikation, eine API mit dem Namen yFiles an. Diese kann man aber nur für Java, JavaFX, .NET, WPF und HTML verwenden. Weil wir als Diplomarbeitsteam uns dazu entschieden haben *Python* als Programmiersprache zu verwenden stieß man hierbei schon auf die ersten Probleme.

Jedoch konnte durch den Aufbau der Datei als XML und durch reverse Engineering, der Graph in Form von XML als Text in eine Datei geschrieben werden. Dementsprechend wurde ein Konvertierer entwickelt der die Informationen der XERML-Datei sinngemäß in eine GraphML-Datei umwandelt, damit hinterher ein vollständiges ER-Diagramm erstellt wird.

3.6.3.1 Initiierung der Datei

Damit die Datei von yEd korrekt gelesen werden kann müssen auch die entsprechenden *Namensräume* für das XML gesetzt werden. Zudem werden von yEd in der GraphML-Datei *key-Elemente* verwendet um den Knoten und Kanten innerhalb des Graphen den richtigen Typ zuweisen zu können. Bei der Initiierung wird zudem noch das Root-Element der GraphML-Datei erstellt. Hierbei ist anzugeben ob es sich bei dem Graphen um einen gerichteten, ungerichteten oder gemischten Graphen handelt. Für den gewählten Ansatz wird ein gerichteter Graph erstellt, weil dies auch die gewählte Einstellung von yEd ist bei der Erstellung eines ER-Diagramms, wie sich durch reverse Engineering herausgestellt hat.

Der XML-Code bis nach dem öffnen des *graph*-Elements sieht also wie folgt aus:

Listing 3.20: Header der GraphML-Datei mit nötigen Namensräumen

```

1 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2 <graphml xmlns="http://graphml.graphdrawing.org/xmlns"
3   xmlns:java="http://www.yworks.com/xml/yfiles-common/1.0/java"
4   xmlns:sys="http://www.yworks.com/xml/yfiles-common/markup/primitives/2.0"
5   xmlns:x="http://www.yworks.com/xml/yfiles-common/markup/2.0"
6   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
7   xmlns:y="http://www.yworks.com/xml/graphml"
8   xmlns:yed="http://www.yworks.com/xml/yed/3"
9   xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
10  http://www.yworks.com/xml/schema/graphml/1.1/ygraphml.xsd">
11 <key for="node" id="d1" yfiles.type="nodegraphics"/>
12 <key for="edge" id="d2" yfiles.type="edgegraphics"/>
13 <graph edgedefault="directed" id="G">
```

3.6.3.2 Erstellung der Knoten und Kanten

→ Passet

Die Informationen für die Erstellung der Knoten und Kanten werden aus einem „etree“ ausgelesen der zuvor mithilfe der XERML-Datei des Benutzers erstellt wird. Beim Konverter werden diese Informationen dann analysiert und in ein entsprechendes Format für die GraphML Datei umgewandelt.

3.6.3.2.1 Knoten

Beim Schreiben des XML-Codes für die Knoten wird ein Element für einen Knoten mit einer *ID* als Attribut angelegt. Mithilfe dieser ID können später die Kanten zwischen den Knoten gezeichnet werden. Innerhalb des Knotenelements sind ein *data-Element*. Diese zwei XML-Elemente gehören zu den grundlegenden GraphML-Elementen. Die spezifische Information über dem Knoten, den man am Ende im ER-Diagramm sieht, steht im *GenericNode* der ein Kindelement von dem data-Element ist.

In der folgenden Strukturbeschreibung sieht man wie ein Knoten in GraphML für die Verwendung mit yEd aufgebaut ist. Man beachte jedoch, dass Attribute mit Standardwerten beim Element *NodeLabel* weggelassen wurden, weil diese ohnehin nicht beeinflusst werden.

```
Element: node
|
+-- Attribut: id
+-- Element: data
|
+-- Attribut: key
+-- Element: y:GenericNode
|
+-- Attribut: configuration
+-- Element: y:Geometry
|
+-- Attribut: heigth
+-- Attribut: width
+-- Attribut: x
+-- Attribut: y
+-- Element: y:Fill
|
+-- Attribut: hasColor
+-- Attribut: color
+-- Attribut: transparent
+-- Element: y:NodeLabel
+-- Element: y:StyleProperties
|
+-- Element: y:Property
|
+-- Attribut: class
+-- Attribut: name
+-- Attribut: value
```

→ Passet

Diese Struktur ist aber nicht immer gleich. Zum Beispiel wird das Element *StyleProperties* nur als Kindelement von Node angeführt, wenn es sich bei dem Knoten um einen schwachen

Entitytypen handelt. Weiteres ist das Attribut *color* nur dann gesetzt, wenn der Benutzer sein ER-Diagramm farbig haben möchte.

Im nachfolgenden XML-Beispiel sieht man wie der Entitytyp „Person“ aus dem Datenmodell Schulinformationssystem aussieht, wenn dieser nicht farbig dargestellt wird:

Listing 3.21: Entitytyp Person dargestellt in GraphML

```

1 <node id="n0">
2   <data key="d1">
3     <y:GenericNode configuration="com.yworks.entityRelationship.small_entity">
4       <y:Geometry height="50.0" width="90.0" x="285.0" y="135.0"/>
5       <y:Fill hasColor="false" transparent="false"/>
6       <y:NodeLabel alignment="center" autoSizePolicy="content"
7         fontFamily="Dialog" fontSize="12" fontStyle="plain"
8         height="17.96875" horizontalTextPosition="center"
9         iconTextGap="4" modelName="internal" modelPosition="c"
10        textColor="#000000" verticalTextPosition="bottom"
11        visible ="true">Person</y:NodeLabel>
12     </y:GenericNode>
13   </data>
14 </node>

```

Wie man ebenfalls anhand des Beispiels sehen kann, wird in Zeile 3 durch das Attribut *configuration* der Typ des Knoten festgelegt. Bei der Erstellung eines ER-Diagramms werden im gegebenen Sachzusammenhang zwischen 3 Typen unterschieden:

- **small_entity**
 - für einen Entitytyp
- **relationship**
 - für einen Beziehungstyp
- **attribute**
 - für ein Attribut

Abhängig vom Typ der an jener Stelle angeführt ist, wählt yEd eine andere Form um den Knoten darzustellen. Wie der Knoten aus dem Beispiel aussieht, kann man in Abbildung 3.28 sehen.

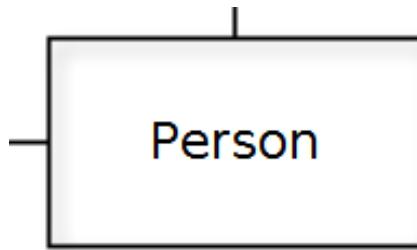


Abbildung 3.28: Entitytyp Person dargestellt in yEd

3.6.3.2.2 Kanten

Die Kanten werden in einer GraphML-Datei über ein *edge-Element* beschrieben. Dieses Element enthält genauso wie auch schon das node-Element eine ID, zusätzlich bekommt es jedoch die Attribute *source* und *target* die als Werte jene IDs der Knoten erhalten, die diese Kante zusammenhalten soll.

Wie auch schon bei den Knoten werden in der folgenden Strukturbeschreibung für die Kanten in einer GraphML-Datei all jene Attribute und Elemente beim Element *EdgeLabel* weggelassen, die mit Standardwerten initiiert sind.

```
Element: edge
|
+-- Attribut: id
+-- Attribut: source
+-- Attribut: target
+-- Element: data
|
+-- Attribut: key
+-- Element: y:PolyLineEdge
|
+-- Element: y:LineStyle
|
+-- Attribut: color
+-- Attribut: type
+-- Attribut: width
+-- Element: y:Arrows
|
+-- Attribut: source
+-- Attribut: target
+-- Element: y:EdgeLabel
```

Dadurch das der Benutzer von ERMTK sich bei der Generierung einer GraphML-Datei entscheiden kann ob er sein ER-Diagramm in der Crow-foot Notation oder in der (min,max)-Notation, fällt bei der Crow-foot Notation das Element EdgeLabel weg.

Im folgendem XML-Beispiel sieht man die Kante die den Entitytypen „Person“ mit dem Beziehungstypen „ist Elternteil von“ verbindet:

Listing 3.22: Kante zwischen einem Entitytypen und einem Beziehungstypen

```
1 <edge id="e26" source="n0" target="n19">
2   <data key="d2">
3     <y:PolyLineEdge>
4       <y:LineStyle color="#000000" type="line" width="1.0"/>
5       <y:Arrows source="none" target="none"/>
6       <y:EdgeLabel alignment="center" configuration="AutoFlippingLabel"
7         distance="2.0" fontFamily="Dialog" fontSize="12"
8         fontStyle="plain" hasBackgroundColor="false"
9         hasLineColor="false" height="17.96875"
10        horizontalTextPosition="center" iconTextGap="4"
11        modelName="custom" preferredPlacement="source_right"
12        ratio="0.5" textColor="#000000" verticalTextPosition="bottom"
13        visible="true" width="90.3203125" x="-15.16015625"
14        xml:space="preserve" y="-22.94091796875003">
```

```

15      (0,n)
16      <y:LabelModel>
17          <y:SmartEdgeLabelModel autoRotationEnabled="false" defaultAngle="0.0"
18              defaultDistance="10.0"/>
19      </y:LabelModel>
20      <y:ModelParameter>
21          <y:SmartEdgeLabelModelParameter angle="0.0"
22              distance="30.0" distanceToCenter="true" position="right" ratio="0.0"
23              segment="0"/>
24      </y:ModelParameter>
25      <y:PreferredPlacementDescriptor angle="0.0"
26          angleOffsetOnRightSide="0" angleReference="absolute"
27          angleRotationOnRightSide="co" distance="-1.0"
28          placement="source" side="right"
29          sideReference="relative_to_edge_flow"/>
30      </y:EdgeLabel>
31  </y:PolyLineEdge>
32 </data>
33 </edge>

```

Wie man in der 1. Ziele des Beispiels erkennen wird dort die Information gespeichert welche Knoten verbunden werden. Dies wird dann in yEd wie in der Abbildung 3.29 dargestellt.

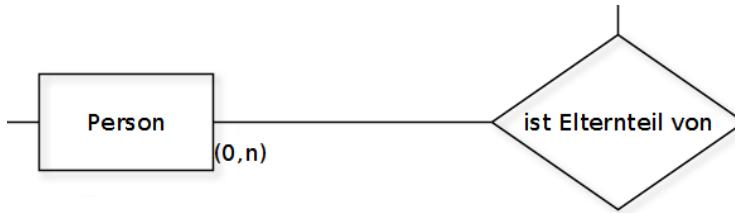


Abbildung 3.29: Entitytyp Person dargestellt in yEd

3.6.4 Python-Code für die Generierung

Um die GraphML-Datei für yEd zu erstellen, wurde die Klasse *GraphmlConverter* geschrieben die den Inhalten der XERML-Datei in das richtige Format bringt und in eine Datei schreibt.

Zu Beginn wird die Datei, die später mit yEd geöffnet werden soll erstellt. Dazu wird der Parameter *output* abgefragt, den der Benutzer angeben kann. Je nach Angaben des Benutzers wird die Datei an dem momentanen Pfad mit dem Namen *output.graphml* erstellt an dem der Benutzer den Befehl ausgeführt hat oder er gibt einen Pfad mit Namen an, andem die Datei erstellt werden soll. Der Codeausschnitt 3.23 behandelt diesen Fall:

Listing 3.23: Codeausschnitt für die Erstellung der GraphML-Datei

```

1 try:
2     if args.output is None:
3         if os.path.exists("output.graphml"):
4             os.remove("output.graphml")
5         self.out = open("output.graphml", "w+")
6     else:
7         if os.path.exists(os.getcwd() + "/" + args.output):
8             os.remove("./" + args.output)

```

```

9     self.out = open(os.getcwd() + "/" + args.output, "w+")
10    self.init_file()
11 except IOError:
12     print(Fore.RED + "There was a Problem creating the file. Please try again" + Fore.RESET)
13     return

```

Wie man in Zeile 10 von 3.23 sieht, wird dort die Funktion `init_file()` aufgerufen. In dieser Funktion wird die Datei soweit vorbereitet, dass der XML-Code für die Entity- und Beziehungstypen 3.21 und für die Kanten 3.22 hineingeschrieben werden kann. Der XML-Code der dabei hineingeschrieben wird ist in Listing 3.20 zu sehen.

→ Passet

Anschließend wird in einem XML-Baum der die Informationen aus der XERML-Datei ent-hält nach den Entity- und Beziehungstypen gesucht damit auch diese ihren Weg in die GraphML-Datei finden. Dieser Vorgang wird in dem Listing 3.24 vereinfacht dargestellt. Vereinfacht bedeutet in diesem Sinne, dass `if`-Anweisungen, `for`-Schleifen, und Definitio-nen von Variablen sowie deren Verwendung bei den Funktionsaufrufen weggelassen werden, damit die Logik des Codes im Vordergrund steht. Die komplette Darstellung des Codes kann man im Anhang finden.

Listing 3.24: Vereinfachter Code zum hizufügen der Entity- und Beziehungstypen

```

1 root = self._tree.getroot()
2 #
3 for child in root:
4     if child.tag == "ent":
5         self.create_node()
6         if args.attr:
7             for attr in child:
8                 self.create_attribute()
9 #
10    elif child.tag == "rel":
11        for part in child:
12            if part.tag == "part":
13                self.create_node()
14 #
15            if args.notation == "crowfoot":
16                # Die folgende if-Anweisung wird betreten wenn ein fuer beide Kantenenden ein Knoten
17                # definiert ist
18                if key_e != "" and key != "":
19                    self.create_edge()
20                else:
21                    print(Fore.RED + "There was an entity missing for a relationship" + Fore.RESET)
22            else:
23                # Die folgende if-Anweisung wird betreten wenn ein fuer beide Kantenenden ein Knoten
24                # definiert ist
25                if key_e != "" and key != "":
26                    self.create_chen_edge()
27                else:
28                    print(Fore.RED + "There was an entity missing for a relationship" + Fore.RESET)
29 #
30            elif part.tag == "super":
31                if not generalization_node:
32                    generalization_node = True
33                self.create_generalization_node()
34                # Die folgende if-Anweisung wird betreten wenn ein fuer beide Kantenenden ein Knoten
35                # definiert ist
36                if key_e != "" and key != "":
37                    if args.notation == "chen":
38                        self.create_chen_edge()

```

```

36         else :
37             self.create_edge()
38     else :
39         print(Fore.RED + "There was an entity missing for a relationship" + Fore.RESET)
40 #
41     elif part.tag == "sub":
42         # Die folgende if-Anweisung wird betreten wenn ein fuer beide Kantenenden ein Knoten
43         # definiert ist
44         if key_e != "" and key != "":
45             if args.notation == "chen":
46                 self.create_chen_edge()
47             else :
48                 self.create_edge()
49             else :
50                 print(Fore.RED + "There was an entity missing for a relationship" + Fore.RESET)
51     elif part.tag == "attr" and args.attr:
52         self.create_attribute()

```

Nachdem der vereinfachte Codeabschnitt 3.24 abgeschlossen ist, wird die GraphML-Datei ordnungsgemäß mit der Funktion `close_file()` geschlossen. Als letzten Schritt wird dem Benutzer noch mitgeteilt, wo sich die GraphML-Datei befindet, damit dieser sein ER-Diagramm in yEd inspizieren kann.

→ Passet

3.7 Graphviz

3.7.1 Allgemeines

Graphviz ist eine Open-Source Software mit der man Graphen darstellen kann. Entwickelt wurde es von AT&T und den Bell Labs, die es zum Ersten mal in den 1990er veröffentlichten. Die aktuellste Version von Graphviz ist die Verion 2.41 und wurde am 25.Dezember 2016 veröffentlicht. Graphviz ist dazu da aus strukturierten Daten gerichtete oder ungerichtete Graphen zu generieren.²²

Außerdem wird Graphviz in Programmen verwendet wie in z.B.:

- AsciiDoc
- PlantUML
- Trac
- Sphinx

Wobei die letzten beiden Programme im laufe des Projekts verwendet wurden.

In drei von vier Ausgabeformaten dieses Projekts kommt Graphviz zum Einsatz. In einem Ausgabeformat, um damit selbst die Modelle zu generieren und in den anderen zwei Ausgabeformaten, um an die Koordinaten zu kommen an denen die Elemente positioniert werden müssen.

3.7.1.1 Ablauf des Programms

Listing 3.25: Programmcode, um einen Graphen anzulegen

```

1
2 name = "erd"
3 set = False
4 for child in _root:
5 if child.tag == "title":
6 name = child.get("name")
7 set = True
8 if set:
9 break
10 _erd = Graph(name, filename=name, engine="sfdp")
11 self.create_erd(args, _erd, _root)

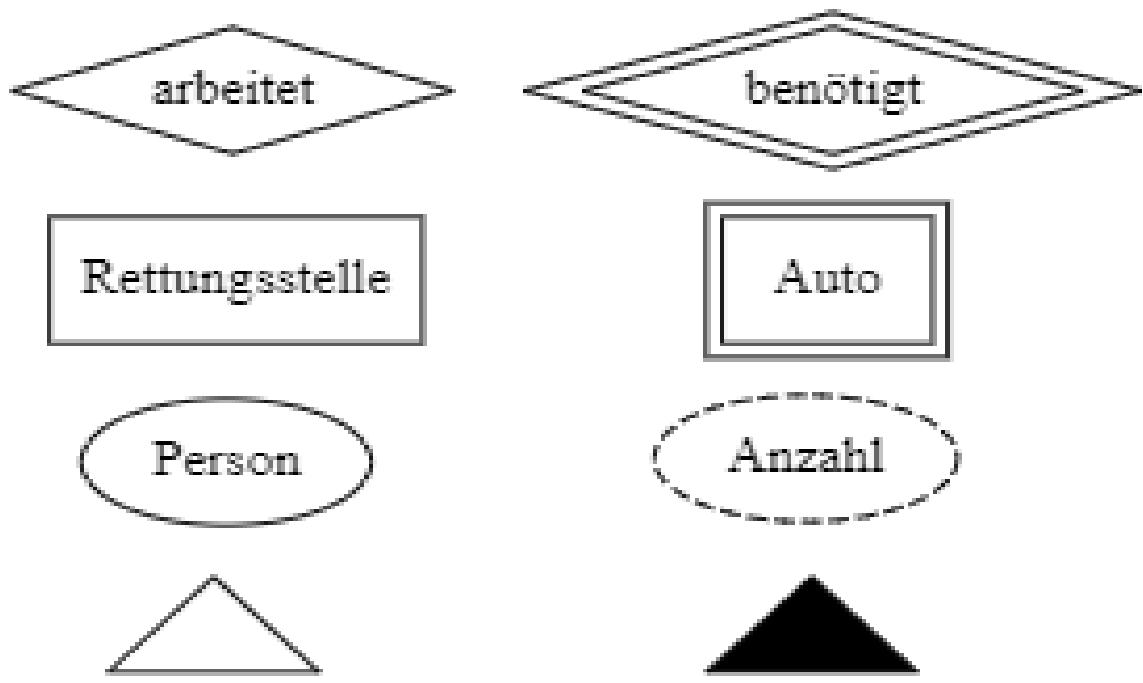
```

Im Listing 3.25 sieht man wie ein leerer Graph, der nur einen Namen und eine Engine besitzt, erstellt wird. Die Variable „set“, verhindert das mehrfache setzen vom Namen des Graphen.

Für die Darstellung eines Entity-Relationship Diagramms werden folgende Formen benötigt:

- Raute (einfach umrandet oder doppelt umrandet)
- Rechteck (einfach umrandet oder doppelt umrandet)
- Ellipse (einfach umrandet oder strichliert umrandet)
- Dreieck (schwarz ausgemahlt oder leer)

²² Graphviz - Graph Visualization Software.

**Abbildung 3.30:** Formen in Graphviz

→Fischbacher

Diese Elemente werden dann mit Linien verbunden. Diese können im speziellen Fall einer totalen Vererbung auch als doppelte Linie vorkommen.

Listing 3.26: Programmcode, um Entities zu erstellen

```

1
2 def create_ent(self, erd, name, weak):
3     if not weak:
4         erd.attr("node", shape="box", peripheries="1")
5         erd.node(name)
6     else:
7         erd.attr("node", shape="box", peripheries="2")
8         erd.node(name)

```

Um einen Entity-Typen zu zeichnen muss vorher der Name des Entity-Typen und ob er abhängig oder nicht abhängig ist bekannt sein. Außerdem muss der Graph in dem das Element sich befindet mitgegeben werden. Wie man im Listing 3.26 sieht wird durch eine if-Verzweigung darüber entschieden ob ein Rechteck mit doppeltem Rand oder einfachem Rand erstellt wird. Dementsprechend werden die Eigenschaften von „erd.attr“, mit dem Namen node angepasst.

Listing 3.27: Programmcode, um Attribute zu erstellen

```

1
2 def create_attr(self, erd, name, prime, label):
3 if prime:
4 erd.attr("node", shape="ellipse", style="dashed", label=label)
5 erd.node(name)
6 else:
7 erd.attr("node", shape="ellipse", label=label)
8 erd.node(name)

```

Bei dem Erstellen der Attribute muss zusätzlich zu Name und in welchem Graphen es sich befindet, auch bekannt sein ob es sich um einen Primär-Schlüssel handelt und zu welchem Entity-Typen das Attribut gehört. Der Parameter „label,“ ist der Name des Attributs und der Parameter „name,“ ist der Name von dem dazugehörigen Entity-Typen. Durch den Parameter „prime,“ wird in der if-Verzweigung, die man im Listing 3.27 sieht, ob es sich um einen Primär-Schlüssel handelt. Falls dem so ist wird die Ellipse mit strichliertem Rand erstellt.

Listing 3.28: Programmcode, um Beziehung zu erstellen

```

1
2 def create_rel(self, erd, name, weak, sup, disjoint):
3 if weak:
4 erd.attr("node", shape="diamond", peripheries="2")
5 erd.node(name)
6 elif sup:
7 if disjoint:
8 erd.attr("node", shape="triangle", color="black")
9 erd.node(name=name, style="filled")
10 else:
11 erd.attr("node", shape="triangle", color="white")
12 erd.node(name=name, style="filled")
13 else:
14 erd.attr("node", shape="diamond", peripheries="1")
15 erd.node(name)

```

Für die Erstellung von den Beziehungen zwischen zwei Entity-Typen muss man angeben ob es sich um eine abhängige Beziehung handelt oder nicht, außerdem muss noch angegeben werden ob eine Vererbung vorliegt und wenn ja ob sie disjunkt oder nicht disjunkt ist. Wie man im Listing 3.28 sieht wird zunächst unterschieden, ob die Beziehung einer Vererbung angehört, einem abhängigen Entity-Typen angehört oder ob sie eine gewöhnliche Beziehung zwischen zwei Entity-Typen ist. Je nachdem wird dann entweder eine Raute oder ein Dreieck erstellt. Außerdem wird darüber entschieden ob das Dreieck schwarz gefüllt ist oder weiß ist. Der Beziehung ist zu diesem Zeitpunkt nicht bekannt mit welchen Linien sie mit welchen Entity-Typen verbunden wird.

Listing 3.29: Programmcode, um Formen zu verbinden

```

1
2 def create_edgeTotal(self, erd, ent1, ent2):
3     erd.edge(ent1, ent2, peripheries="2")
4
5 def create_edgeRel(self, erd, ent1, ent2, mini, maxi):
6     s = "(" + mini + ", " + maxi + ")"
7     erd.edge(ent1, ent2, label=s, peripheries="1")
8
9 def create_edge(self, erd, ent1, ent2):
10    erd.edge(ent1, ent2, peripheries="1")

```

- Um gewöhnliche Entity-Typen mit einem normalen Beziehungs-Typen zu verbinden wird die Funktion “create edgeRel”, verwendet. Dabei werden die beiden Formen im Graphen mitgegeben, der Graph selbst und min und max Werte des Beziehungs-Typen.
- Im Falle einer Vererbungen mit einer totalen Beziehung wird die Funktion “create edgeTotal”, verwendet. Dabei werden nur die beiden Formen im Graphen mitgegeben und der Graph selbst.
- Für jeden anderen Fall wird die Funktion “create edge”, verwendet aber wird diese Funktion am häufigsten beim Verbinden der Attribute mit dem dazugehörigen Entity-Typen.

Listing 3.30: Programmcode, um Formen zu verbinden

```

1
2 erd.render(format="pdf")
3 cwd = os.getcwd()
4 print("Your ERD is successfully created in: " + cwd)

```

Nachdem der Graph und die Formen erstellt wurden und die Formen miteinander verbunden wurden. Wird das Dateiformat angegeben in welchem der Graph ausgegeben werden soll. Außerdem wird das Verzeichnis in dem die Datei gespeichert wird auf der Konsole ausgegeben.

Graphviz bietet standardmäßig eine Vielzahl an verschiedenen Datei- oder Output-Formaten an. Insgesamt sind es 55 verschiedene Formate und allein davon sind sechs Formate Variationen des „dot“, Formats. Für dieses Projekt wurden hauptsächlich das PDF-Format genutzt. Das PIC-Code-Tool verwendete jedoch das „pic-Format“, um die Koordinaten des Graphen mittels Graphviz zu ermitteln und an das Tool weiterzuleiten.²³ Zu den gängigsten Datenformaten zählen:

- dot
- bmp
- json
- pdf
- svg
- png

3.7.2.1 dot-Format

Die Ausgabe dieses Formats und der Formate gv, xdot, xdot1.2 und xdot1.4, erfolgt in dot-language. Darin stehen dann die Attribute für:

- Begrenzungsrahmen
- Position
- Breite
- Höhe
- Labelpunkt

Außerdem gibt es dann noch die Attribute „rects,, und/oder „vertices,, und noch weitere je nachdem welche Formen im Graphen angezeigt werden.

3.7.2.1.1 xdot

„xdot“, steht für „extended dot“, und ist einfach eine erweiterte Version von dot, die genauere Daten über den Graph zurück liefert als dot. Kommentare für die Lesbarkeit und für das Verständnis kann man schreiben. Weiteres gibt es auch das Attribut „xdotversion“, um Änderungen am Graphen zu erlauben.

3.7.2.1.2 xdot1.4

Mit „xdot1.4“, können Farbstrings lineare und radiale Farbverläufe codieren.

- Lineare-Form: '[' x0 y0 x1 y1 n [color-stop]+ ']
- Radiale-Form: '(' x0 y0 r0 x1 y1 r1 n [color-stop]+ ')'

Die Zahl n gibt an wie viele Color-stop es gibt. Wenn eine Form keine Farbe besitzt kann zu ihr auch keine Verbindung erstellt werden, auch wenn sie theoretisch existiert.

²³ Output Formats.

3.7.2.2 json

Dateiformate die JSON als ausgabe haben:

- json
- json0
- dot json
- xdot json

Bei Verwendung des Formats “json,, bekommt man das selbe Ergebniss wie bei dem Format “-Txdot,, nur als JSON anstelle einer Datei in DOT language. Genauso verhält es sich auch bei “json0,, und “-Tdot,,. Die Formate “dot json,, und “xdot json,, sind “json0,, und “json,, ähnlich. Jedoch verwenden diese zwei Formate nur die Input Daten und somit ist bei ihnen kein Layout durch einen bestimmten Algorithmus vorgegeben, dadurch enthalten diese Formate nur die Information über die Elemente selber. Die daraus entstehende JSON-Datei kann wie folgt aufgebaut sein wie im Listing 3.31.

Listing 3.31: Beispiel für eine JSON Datei von Graphviz

```

1
2 {
3 "nodes": [
4 {
5 "id": "n0",
6 "label": "A node",
7 "x": 0,
8 "y": 0,
9 "size": 3
10 },
11 {
12 "id": "n1",
13 "label": "Another node",
14 "x": 3,
15 "y": 1,
16 "size": 2
17 },
18 ],
19 "edges": [
20 {
21 "id": "e0",
22 "source": "n0",
23 "target": "n1"
24 },
25 {
26 "id": "e2",
27 "source": "n1",
28 "target": "n0"
29 }
30 ]
31 }
```

3.7.2.3 pdf

Das Format “pdf,, erstellt ein PDF-Dokument. Falls man Anker benutzen möchte sollte man als alternative das Format “ps2,, benutzen, da das mit diesem möglich ist und dem

Format „pdf,, ähnelt.

3.7.2.4 plain-ext

→Fischbacher

Die Formate „plain,, und „plan-ext,, geben ein Text-Dokument aus dieses besteht aus vier verschiedenen Arten von Zeilen:

- graph
- node
- edge
- stop

Die „stop,, Zeile markiert das Ende eines Graphen. Die Ausgabe besteht aus einer Zeile für den Graphen, mehreren Zeilen für Elemente, pro Element keine bis mehrere Zeilen für die Verbindungen und am Ende befindet sich ein Stop.

3.7.2.4.1 graph

Die „graph,, Zeile gibt die Breite sowie die Höhe des Graphen mit den Attributen „height,, und „width,, an. Wenn der Graph skaliert werden muss gibt es ein Attribute „scale,,. Dieses Attribute gibt an in welchem Verhältnis der Graph skaliert werden muss. Außerdem sind alle Werte nicht skaliert was bedeutet, wenn der Wert des Attributes „scale,, nicht 1.0 ist muss jede Zahl damit multipliziert werden. Standardmäßig ist der Koordinatenursprung in der linken unteren Ecke.

3.7.2.4.2 node

Die „node,, Zeilen besitzt die Attribute „name,, welches dem Element seinen Namen verleiht, ein „x,, und ein „y,, Attribute welche die Position des Elements festlegen. Außerdem wird die Breite sowie die Höhe des Element, wie bei dem „graph,, Statement mit den Attributen „height,, und „width,, angegeben. Außerdem verfügt es auch noch über die Attribute „label,, „style,, „shape,, „color,, und „fillcolor,,. Wenn das „style,, Attribute leer ist, ist der Rahmen des Elements Standardmäßig solide.

3.7.2.4.3 edge

Die zwei wichtigsten Attribute der „edge,, Zeile sind „tail,, und „head,, diese geben die zwei Namen der zu verbindenden Elemente an.

3.7.2.5 png

Das „png,, Format ist eines der unpraktischsten dafür einfachsten Formate da es lediglich ein Bild zurück liefert. Jedoch kann man bei diesem Format einstellen ob das Bild mit Antialiasing generiert werden soll oder ohne. Außerdem kann man auch noch angeben ob es mit „Indexed color,, oder mit „True color,, generiert werden soll.

Die sogenannten Engines in Graphviz geben an mit welchem Verfahren die Koordinaten der Elemente berechnet werden. Das kann von großem Vorteil sein wenn man bestimmte Ergebnisse erzielen will.²⁴

3.7.3.1 dot

dot ist in erster Linie dafür gedacht Strukturen hierarchisch abzubilden. Das bedeutet das dabei alle Kanten in etwa die selbe Richtung verlaufen, meistens von oben nach unten oder von links nach rechts. Durch diese Anordnung der Elemente kommt es selten zu Kanten Überschneidungen da sie auch in der Regel eher kurz gehalten werden. Daher wird man eher dot verwenden wenn man weiß, dass die Kanten eine Richtung besitzen.

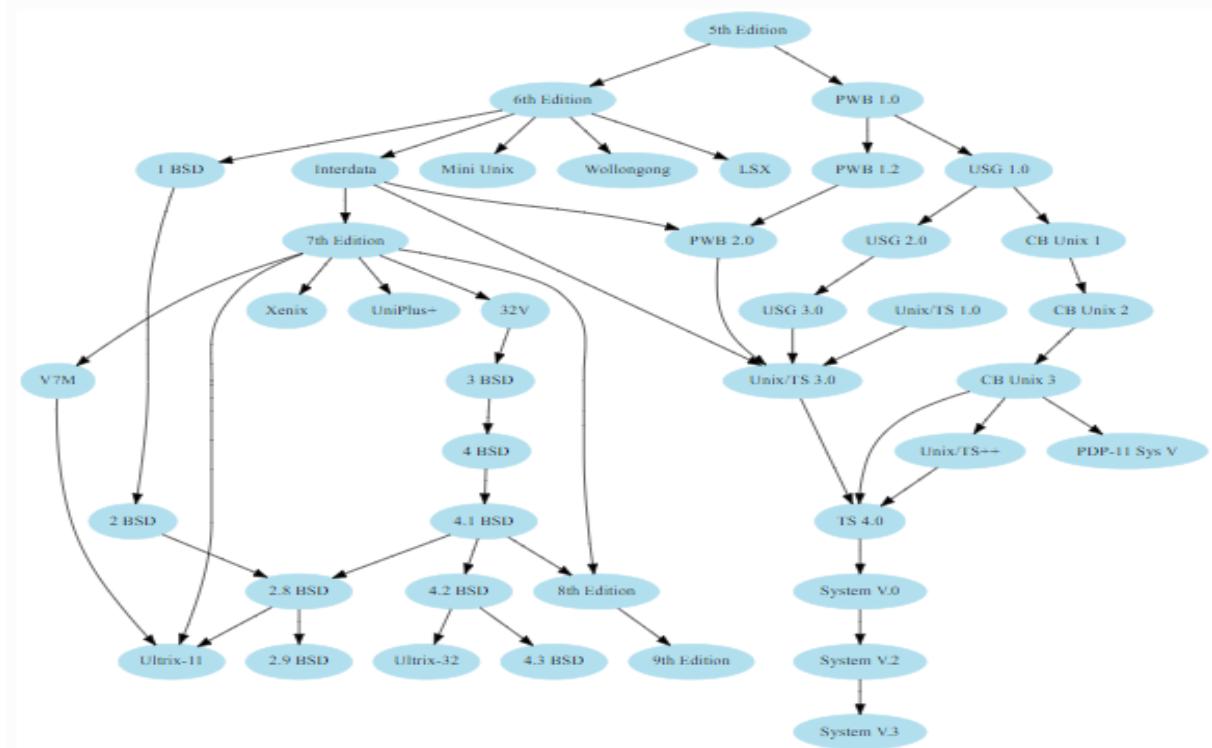


Abbildung 3.31: Typischer dot Graph

²⁴noauthor_documentation_nodate.

²⁵graphics - How to programmatically draw an organization chart?

3.7.3.2 neato

→Fischbacher

neato benutzt das sogenannte „spring model“, Layout. Neato benutzt den Kamada-Kawai-Algorithmus und ist eher dafür gedacht kleinere Graphen (um die 100 Knoten) darzustellen. Man benutzt Neato dann wenn man nichts außer der Größe des Graphen kennt. Neato führt einen Vorgang durch, beim generieren des Graphen, der äquivalent zu Multi-Dimensionaler Skalierung ist.²⁶

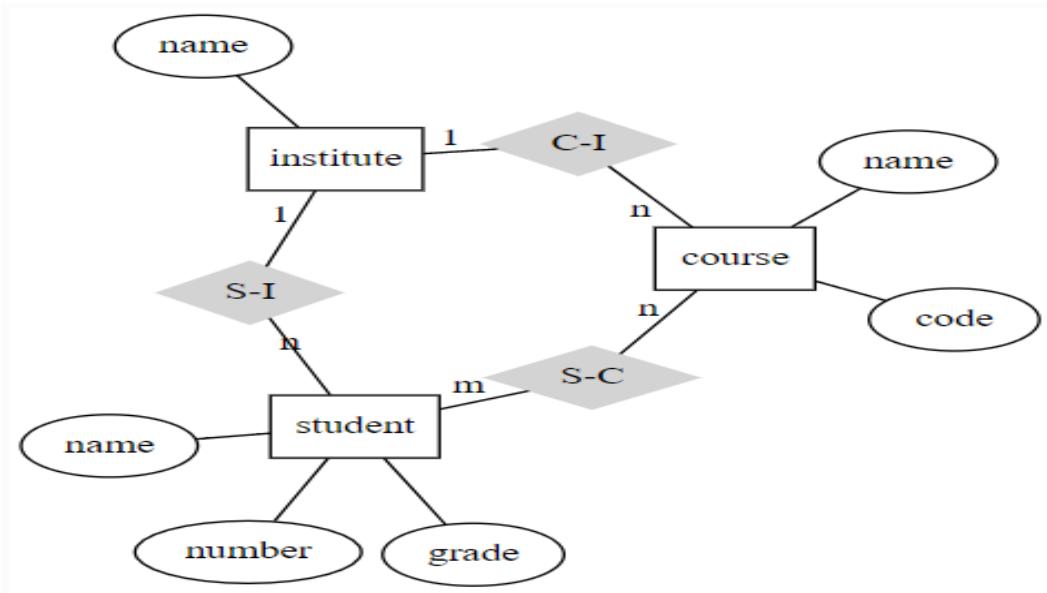


Abbildung 3.32: Typischer neato Graph

3.7.3.3 fdp

fdp benutzt genau wie neato auch das „spring model“, Layout. Jedoch wendet es ein anderes Verfahren an.

3.7.3.4 sfdp

sfdp benutzt genau wie fdp und neato auch das „spring model“, Layout. Außerdem ist sfdp eine multiscale Variante von fdp mit der man sehr große Graphen darstellen kann.²⁷

²⁶North, „Drawing graphs with NEATO“.

²⁷performance - Graphviz sfdp inferior in ubuntu comparing to mac?

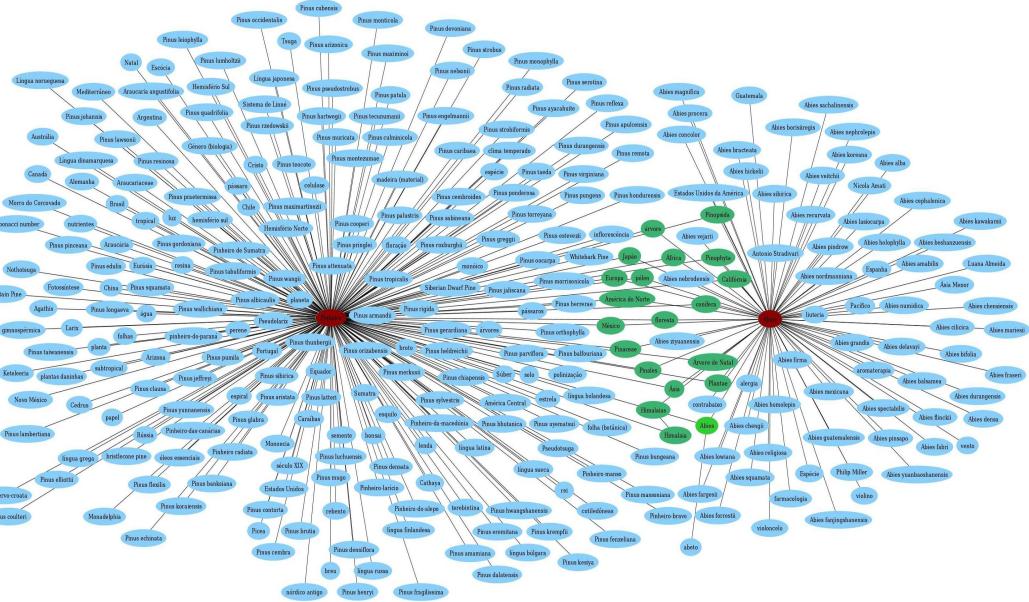


Abbildung 3.33: Typischer sfdp Graph

3.7.3.5 circo

→ Fischbacher

circo erstellt ein circuläres Layout nach Six und Tollis 99, Kauffman und Wiese 02. Dieses Verhalten verlieh ihm auch seinen Namen. Diese Art von Graphen Generierung ist dazu geeignet Zyklische Strukturen, wie man sie meist bei Telekomunikations-Netzwerken vorfindet, darzustellen.

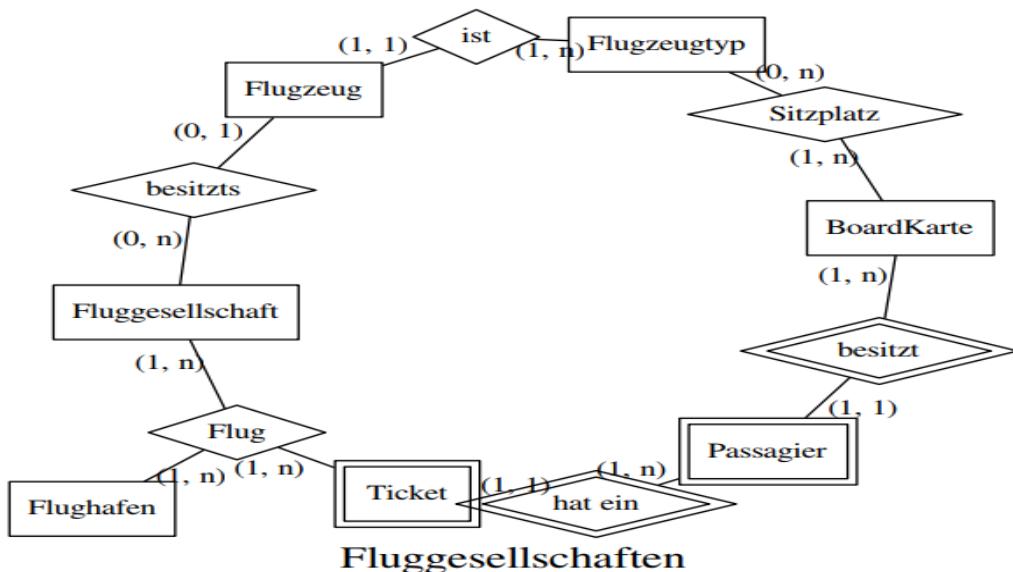


Abbildung 3.34: Typischer circo Graph

→Fischbacher

3.7.3.6 twopi

twopi erstellt ein radiales Layout nach Graham Wills 97. Dabei werden die Knoten auf konzentrischen Kreisen, abhängig von der Distanz zu einem Wurzel Knoten, platziert.

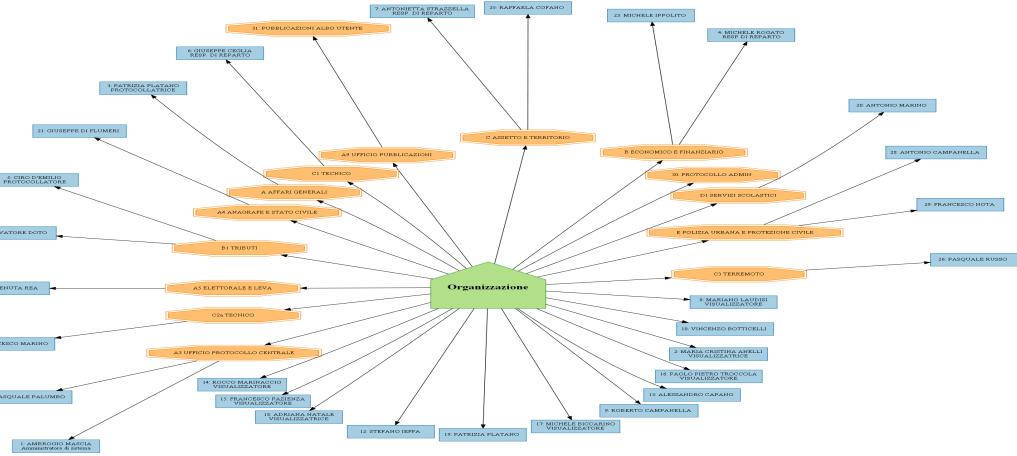


Abbildung 3.35: Typischer twopi Graph

3.7.4 Vor- und Nachteile von Graphviz

3.7.4.1 Engines

3.7.4.1.1 Vorteile

Durch sechs verschiedene Engines kann man eine Vielzahl an verschiedenen Graphen erstellen. Wenn man genug über seinen Graphen in Erfahrung gebracht hat kann man die eine Engine die für seinen Graphen am besten passt auswählen und somit ein ideales Ergebnis erzielen.

3.7.4.1.2 Nachteile

- Jedoch der große Nachteil von Engines ist das man gebunden ist an sie. Man kann Beispielsweise nicht festlegen wo ein bestimmtes Element positioniert werden soll.
- Außerdem kann man nicht bestimmen welche Größe die einzelnen Elemente haben sollen, da sie sich, je nach ihrem Inhalt, dynamisch anpassen.

3.7.4.2 Formen

3.7.4.2.1 Vorteile

Es existiert eine Vielzahl an Formen und Linien, wodurch einer Freiheit bei der Modellierung ermöglicht wird.

3.7.4.2.2 Nachteile

Ein Nachteil ist das man in seiner Freiheit eingeschränkt ist.

- Will man eine neue Form festlegen, geht das schlicht und einfach nicht.

- Will man Text in einem Element unterstrichen haben, geht auch dies nicht.
- Beim modellieren hat man die freie Auswahl über schon vorhandenen Formen, jedoch um eigene Formen dann selbst zu erstellen bedarf es viel Arbeit und selbst dann ist das Ergebnis nicht zufrieden stellend und nicht immer das selbe.
- Ein anderer Nachteil ist, dass man wenn man ein Element verändert z.B.: einen anderen Rahmen gibt, dann wird das als neuer Standard angenommen und man muss, wenn man einen Sonderfall behandelt der selten vorkommt, jedes mal wieder den alten Standard festlegen.

3.7.4.3 Dateiformate

3.7.4.3.1 Vorteile

Manche Dateiformate, wie z.B.: JSON, bieten einem die Möglichkeit nur auf die Informationen eines Graphen zuzugreifen die man auch wirklich benötigt. Das ist praktisch für Fälle wobei Graphviz nicht zur Darstellung verwendet wird sondern nur um das Layout festzulegen.

3.7.5 Vergleich von dot und neato

Die dot Engine erstellt hierarchische Graphen und die neato Engine gerichtete Graphen. In der Regel wird dot häufiger genutzt, jedoch ist neato für Entity-Relationship Diagramme weit aus besser geeignet.²⁸

→Fischbacher

3.7.5.1 Weingut

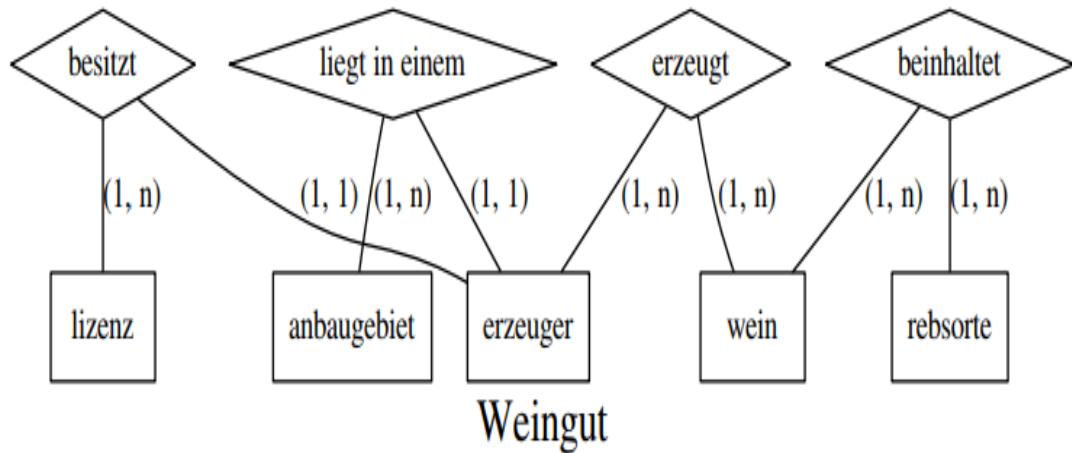


Abbildung 3.36: Weingut mit der dot Engine

Wie man an der oberen Abbildung 3.36 sehen kann wäre dot für kleinere Graphen wie hier für dem Weingut durchaus denkbar. Solange Graphen nicht zu groß und nicht mehr als

²⁸graphviz [dot and/or neato].

zwei Ebenen besitzen ist dot eine akzeptable Möglichkeit den Graph darzustellen.

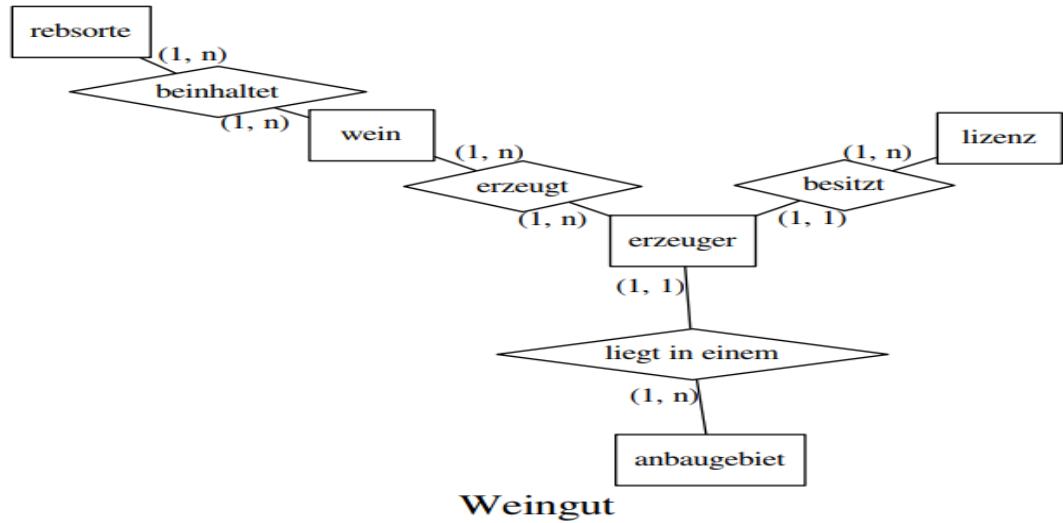


Abbildung 3.37: Weingut mit der neato Engine

Genau wie dot hat auch neato keinerlei Probleme mit kleinen Graphen wie dem Weingut.

3.7.5.2 Rettungsstelle

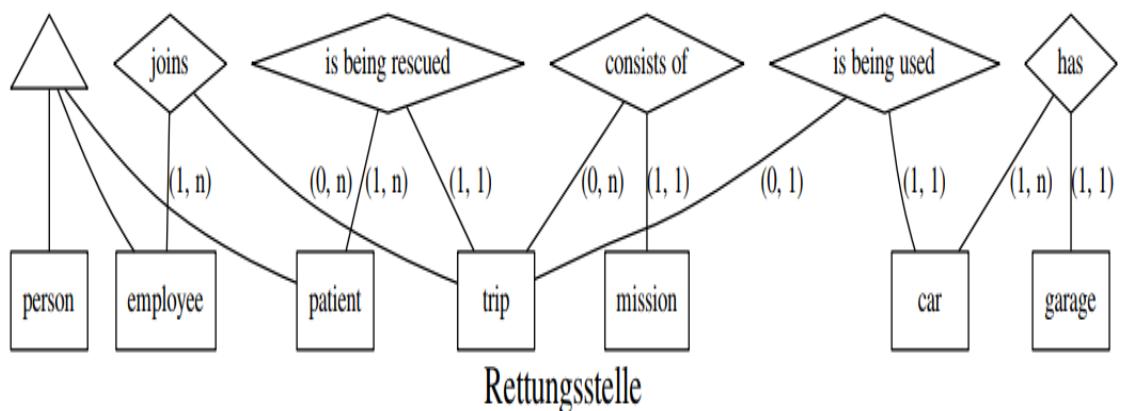


Abbildung 3.38: Rettungsstelle mit der dot Engine

→Fischbacher

Jedoch bei einem etwas größerem Graphen sieht man schon das dot zu breit wird und für Attribute keinen Spielraum mehr lassen würde.

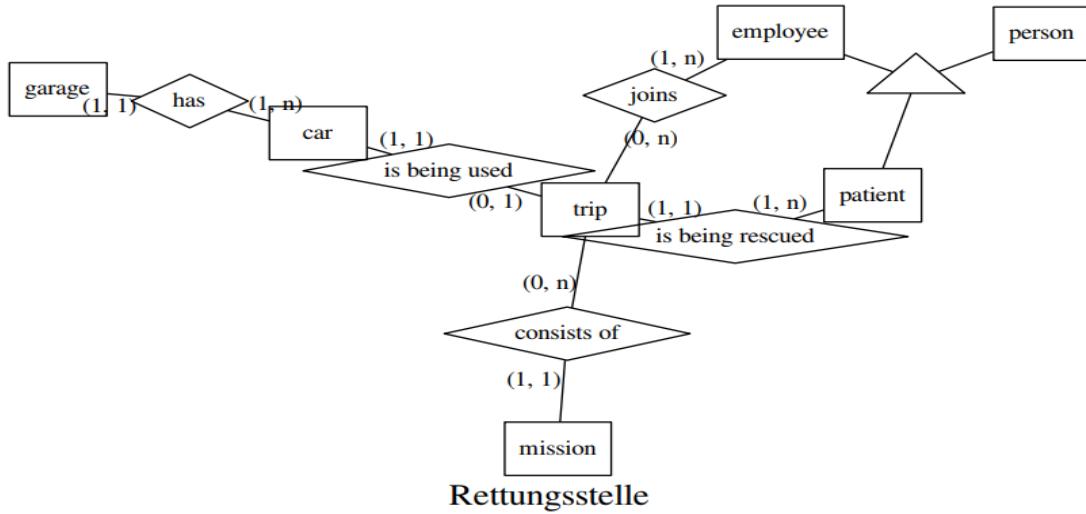


Abbildung 3.39: Rettungsstelle mit der neato Engine

Bei neato merkt man, wie hier bei der Rettungsstelle, dass noch Platz für Attribute wäre. Trotzdem merkt man das die Elemente immer näher aneinander rücken da der Graph für neato nicht die ideale Größe besitzt.

3.7.5.3 Wahl

→Fischbacher

Wenn man zwischen dot und neato wählen müsste sollte man vorher genaue Daten über den Graphen, den man generieren will, in Erfahrung bringen. In der Regel gilt beide eigenen sich für kleinere Graphen jedoch tut sich dot mit Attributen schwer und neato mit vielen Elementen. Bei vielen Elementen kann man aber auch sfdp benutzen da es neato ähnelt und auf größere Graphen ausgelegt ist.

3.8 LibreOffice Draw

3.8.1 Einführung in LibreOffice Draw

→Homolka

Das vektorbasierte Grafikprogramm *Draw* ist Teil des *LibreOffice* Pakets und wurde von der *The Document Foundation* entwickelt. Andere Bestandteile des *LibreOffice* Pakets sind *Writer*, *Calc*, *Impress*, *Base* und *Math*. *Draw* bietet diverse Funktionalitäten um Skizzen, Poster, technische Zeichnungen, Flussdiagramme, verlustfrei skalierbare 2D-Vektorgrafiken mit 3D-Effekten, etc. zu erstellen. Der Funktionsumfang von *Draw* gliedert sich in etwa zwischen den Grafikprogrammen *Paint.NET* und *GIMP* ein. Im Gegensatz zu *Paint.NET* stehen mehrere vordefinierte 2D- und 3D-Formen zu Verfügung. Allerdings sind viele grafische Bearbeitungsfunktionen, die *GIMP* zur Verfügung stellt, nicht enthalten. Alle *LibreOffice* Programme sind auf den Plattformen *Windows*, *Linux* und *macOS* verfügbar. Darüber hinaus wird die Webapp *LibreOffice Online* angeboten, die auf einem Server installiert wird und über einen Webbrower genutzt werden kann. *Draw* verwendet für die Speicherung und Darstellung der gezeichneten Objekte ein Koordinatensystem, wobei der Punkt [0,0] sich in der linken oberen Ecke der Seite befindet.³⁰³¹

3.8.2 Entwicklung

→Homolka

Die Entwicklungsgeschichte des *LibreOffice* Pakets beginnt mit dem Projekt *StarDivision*, welches von dem Unternehmen *StarOffice* initialisiert wurde. 1999 wurde das Unternehmen von *Sun Microsystems* übernommen. Dieses gründete im darauffolgenden Jahr die Stiftung *OpenOffice.org* mit dem Hauptziel die Weiterentwicklung der Software von Firmeneinflüssen zu bewahren. Im Januar 2010 wurde das Unternehmen *Sun Microsystems* von *Oracle* übernommen. Aufgrund einiger Meinungsverschiedenheiten und unzureichender finanzieller Unterstützung entschieden sich einige der führenden *OpenOffice.org* Mitglieder im September des selben Jahres die gemeinnützige Stiftung *The Document Foundation* zu gründen. Seitdem wird die Software unter dem Namen *LibreOffice* vermarktet. Parallel dazu wird eine andere Version der Software von *Apache*, die die Weiterentwicklung von *Oracle* übernommen hat, unter dem Namen *Apache OpenOffice* vermarktet. Bis auf die fehlende 64-Bit-Version bei *Apache OpenOffice* und höhere Aktualisierungsintervalle bei *LibreOffice* bestehen nur kleine Unterschiede zwischen den zwei verschiedenen Entwicklungszweigen. Die Abbildung 3.40²⁹ zeigt den zeitlichen Verlauf grafisch.³⁰³¹

3.8.3 Generierungsvarianten

→Homolka

Um ein *Entity Relationship Diagramm* in *LibreOffice Draw* automatisiert generieren zu können, bieten sich zwei Möglichkeiten an, die sich in der Komplexität und der Umsetzung sehr stark unterscheiden. Die erste der beiden Möglichkeiten ist der Zugriff über die zu Verfügung stehende *API*. Die zweite Variante ist die Manipulation des Dateiformats.

3.8.3.1 Generierung über die LibreOffice API

3.8.3.1.1 Allgemeines

→Homolka

²⁹Dr. Andrew Davison. *Java LibreOffice Programming*. Deutsch. URL: <https://de.wikipedia.org/wiki/LibreOffice>.

³⁰*LibreOffice Wikipedia*. Englisch. URL: <https://fivedots.coe.psu.ac.th/~ad/jlop/>.

³¹*LibreOffice*. Deutsch. URL: <https://de.libreoffice.org/>.

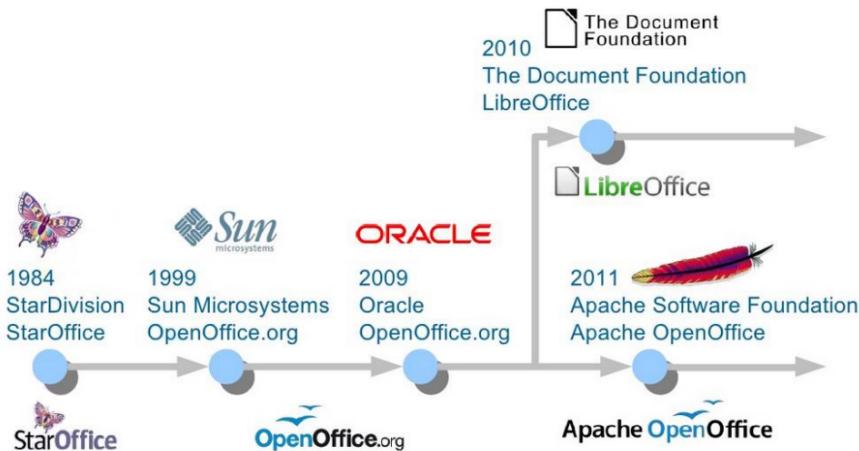


Abbildung 3.40: Timeline der Entwicklungsgeschichte

Um über die *LibreOffice Programmierschnittstelle (API)* auf ein *Draw* Dokument zugreifen zu können benötigt man das *LibreOffice Software Development Kit*. Dieses beinhaltet ein Set von Entwicklungswerkzeugen, Bibliotheken und Header-Dateien. Auf der Website der *LibreOffice API* befinden sich diverse Beispiele um die Funktionsweise und die Verwendung der *API* zu demonstrieren. Die verschiedenen Beispiele unterscheiden sich in der verwendeten Programmiersprache und dem Zielprogramm. Als Programmiersprachen stehen *Java*, *C++*, *Python*, *Basic* und das Objektsystem *Object Linking and Embedding* zur Verfügung. Bei der genaueren Betrachtung zeigt sich, dass das einzige Beispiel, dass *Draw* als Zielprogramm hat, in *Java* geschrieben ist.³²

→Homolka

Um in ein *Draw* Dokument zeichnen zu können, muss zuerst ein Dokument, eine Instanz der Klasse `com.sun.star.frame.XDesktop` mit einem Kontext der Klasse `com.sun.star.uno.XComponentContext` und einem *Komponentenlader* mit der Instanz der Klasse `com.sun.star.frame.XComponentLoader` angelegt werden. In der folgenden Grafik befindet sich ein Beispiel für jenen Code. (siehe Listing 3.32)

→Homolka

Listing 3.32: Programmcode in Java, um ein leeres *Draw* Dokument zu öffnen

```

1
2 com.sun.star.frame.XComponentLoader xCLoader;
3 com.sun.star.lang.XComponent xComp = null;
4
5 try {
6     com.sun.star.lang.XMultiComponentFactory xMCF =
7         xContext.getServiceManager();
8
9     Object oDesktop = xMCF.createInstanceWithContext
10    ("com.sun.star.frame.Desktop", xContext);
11
12    xCLoader = UnoRuntime.queryInterface
13        (com.sun.star.frame.XComponentLoader.class, oDesktop);
14    com.sun.star.beans.PropertyValue szEmptyArgs[] =

```

³² LibreOffice API. Deutsch. URL: <https://api.libreoffice.org/>.

```

15  new com.sun.star.beans.PropertyValue[0];
16  String strDoc = "private:factory/sdraw";
17  xComp = xCLoader.loadComponentFromURL
18  (strDoc, "_blank", 0, szEmptyArgs);
19
20 } catch (java.lang.Exception e) {
21     System.out.println(" Exception " + e);
22     e.printStackTrace(System.out);
23 }

```

33

→Homolka

Über einige weitere Schritte erhält man ein Objekt der Klasse `xShapes`, in das man dann die zu zeichnenden Objekte hinzufügt. (siehe Listing 3.33)

Listing 3.33: Programmcode in Java, um ein leeres *Draw* Dokument zu öffnen

```

1 xDrawDoc = openDraw(xContext);
2 try {
3     System.out.println("getting Drawpage");
4     com.sun.star.drawing.XDrawPagesSupplier xDPS = UnoRuntime
5         .queryInterface(com.sun.star.drawing.
6             XDrawPagesSupplier.class, xDrawDoc);
7     com.sun.star.drawing.XDrawPages xDPn = xDPS.getDrawPages();
8     com.sun.star.container.XIndexAccess xDPi = UnoRuntime
9         .queryInterface(com.sun.star.container.XIndexAccess.class,
10             xDPn);
11    xDrawPage = UnoRuntime.queryInterface(com.sun.star.drawing.
12        XDrawPage.class, xDPi.getByIndex(0));
13 } catch (java.lang.Exception e) {
14     System.out.println("Couldn't create document" + e);
15     e.printStackTrace(System.out);
16 }

```

32

Mit dem in Listing 3.32 und 3.33 gezeigten Programmcode öffnet sich ein *LibreOffice Draw* Dokument ohne Inhalt.

3.8.3.1.2 XERML-Daten Aufbereitung in Java

→Homolka

Um die Daten aus der *XERML*-Datei in ein für *Java* interpretierbares Format zu bringen, wird der *Java DOM Parser* verwendet. Die Daten werden immer in einer Liste des Datentyps `ArrayList<String>` mit folgendem Aufbau gespeichert:

- Für *Entity-Typen* und zugehörige Attribute: Das erste Listenelement enthält den Namen des *Entity-Typs*. Die folgenden Listenelemente enthalten die Namen der *Attribute*.
- Für *Beziehungstypen* und deren Teilnehmer: Das erste Listenelement enthält den Namen des *Beziehungstyps*. Für jeden Teilnehmer werden Listenelemente in folgender Reihenfolge hinzugefügt: Name des teilnehmenden *Entity-Typs*, Minimalwert, Maximalwert.
- Für *Beziehungsattribute*: Das erste Listenelement enthält den Namen des *Beziehungstyps*. *Beziehungsattribute* werden, falls vorhanden, dahinter angehängt.

³³ LibreOffice API. Deutsch. URL: <https://api.libreoffice.org/>.

- Für *Primary-Keys*: Das erste Listenelement enthält den Namen des *Entity-Typs*. Die darauffolgenden Listenelemente sind ein oder mehrere *Primary-Keys*.
- Für *Super-Sub-Beziehungstypen*: Das erste Listenelement enthält den Namen des *Beziehungstyps*. Das zweite und dritte Listenelement enthalten den Wert des *XERML-Attributs total* an zweiter Stelle und den Wert des *XERML-Attributs disjunkt* an dritter Stelle. Dieser kann entweder **true** oder **false** sein.

3.8.3.1.3 Darstellung der grafischen Elemente

→Homolka

Grundsätzlich folgt die Darstellung der verschiedenen grafischen Formen immer dem gleichen Schema. Zuerst wird ein Objekt, das die derzeitige Seite des Dokuments beinhaltet, angelegt. Dieses wird dann in ein Objekt der Klasse **xDrawPage** umgewandelt. Parallel dazu benötigt man ein Objekt der Klasse **XShape**, welches eine Vorlage beinhaltet. Die Vorlage bestimmt die Form des grafischen Elements.

→Homolka

Für die Erstellung der Komponenten eines *ERDs* sind folgende Klassen notwendig:

- **RectangleShape** für die Erstellung von Rechtecken um *Entity-Typen* darzustellen.
- **EllipseShape** für die Erstellung von Ellipsen um *Attribute* darzustellen.
- **PolyPolygonShape** für die Erstellung von Rauten und Dreiecken um *Beziehungs-* und *Super-Sub-Beziehungstypen* darzustellen. Die Klasse kann Formen mit einer beliebigen Anzahl an Seiten darstellen.
- **ConnectorShape** für die Erstellung von Linien, die *Entity-Typen* mit *Attributen* und *Entity-Typen* mit *Beziehungstypen* verbinden. Die Klasse **LineShape** ermöglicht es lediglich die Endpunkte der Linie an eine bestimmte Koordinate zu binden. Falls, in der Nachbearbeitung, eine Form verschoben wird, bleibt die Linie an der festgelegten Koordinate. Um diesem Problem entgegen zu wirken, wird die Klasse **ConnectorShape** verwendet, bei der man die Endpunkte der Linie an ein Objekt einer anderen Klasse binden kann.
- **XText** für die Erstellung von Textfeldern um die Namen der *Entity-Typen* und der *Attribute* anzuzeigen. Für die *min-max-Notation* sind Textfelder ebenfalls nötig.

3435

→Homolka

In allen oben gelisteten Klassen besteht die Möglichkeit, dem Objekt eine Position zuzuweisen. Wird keine Position explizit angegeben, so werden die Koordinaten [0,0] verwendet, wobei sich die Koordinaten auf den linken oberen Punkt des Objekts bezieht.

Die Funktion **setSize** der Klasse **xDrawShape** setzt die Breite und Höhe der Form fest. Standardmäßig werden die Objekte mit einem schwarzen Rand und weißer Füllung gezeichnet.

Mit der Funktion **setProperty** der Klasse **XPropertySet** kann man die Objekte in einer beliebigen Farbe darstellen. Die Farbwerte werden im Dezimalformat angegeben.

Die Listing 3.34 zeigt den Programmcode um einen *Entity-Typ* zu erstellen. Die Variablen **x** und **y** enthalten dabei die Koordinaten, wo das Rechteck gezeichnet werden soll. Die Variable **col** enthält den Dezimalwert *16711680*. Dies entspricht einer rötlichen Färbung.

³⁴ OpenOffice Developer's Guide. Englisch. URL: https://wiki.openoffice.org/w/images/d/d9/DevelopersGuide_OOo3.1.0.pdf.

³⁵ LibreOffice API. Deutsch. URL: <https://api.libreoffice.org/>.

In der Variable `text` wird der Name des *Entity-Typs* mitgegeben, der in der Schriftgröße *6pt* mittig in das Rechteck gezeichnet wird.

→Homolka

Listing 3.34: Programmcode in Java, um ein Entity-Typ mit Text in der Schriftgröße 6pt zu zeichnen

```

1
2 Object drawPages = xDrawPagesSupplier.getDrawPages();
3 XIndexAccess xIndexedDrawPages = (XIndexAccess)
4 UnoRuntime.queryInterface(XIndexAccess.class, drawPages);
5 Object drawPage = xIndexedDrawPages.getByIndex(0);
6 XMultiServiceFactory xDrawFactory = (XMultiServiceFactory)
7 UnoRuntime.queryInterface(XMultiServiceFactory.class, xComponent);
8 Object drawShape = xDrawFactory.createInstance
9 ("com.sun.star.drawing.RectangleShape");
10 XDrawPage xDrawPage = (XDrawPage)
11 UnoRuntime.queryInterface(XDrawPage.class, drawPage);
12 xDrawShape = UnoRuntime.queryInterface(XShape.class, drawShape);
13 xDrawShape.setSize(new Size(width, height));
14 xDrawShape.setPosition(new Point(x, y));
15 xDrawPage.add(xDrawShape);
16
17 XText xShapeText = UnoRuntime.queryInterface(XText.class,
18 drawShape);
19 XPropertySet xShapeProps = UnoRuntime.queryInterface
20 (XPropertySet.class, drawShape);
21
22 xShapeProps.setPropertyValue("FillColor", Integer.valueOf(col));
23
24 com.sun.star.text.XTextCursor xTCursor = xShapeText.
25 createTextCursor();
26 com.sun.star.beans.XPropertySet xTCPS =
27 UnoRuntime.queryInterface(com.sun.star.beans.XPropertySet.class,
28 xTCursor);
29 xTCPS.setPropertyValue("CharHeight", 6.0f);
30 xShapeText.insertString(xTCursor, text, false);
31 com.sun.star.beans.XPropertySet xText =
32 UnoRuntime.queryInterface(com.sun.star.beans.XPropertySet.class,
33 xShapeProps);

```

→Homolka

Das Ergebnis des Programmcodes angewendet auf das Weingut Testmodell wird in Abbildung 3.41 abgebildet:

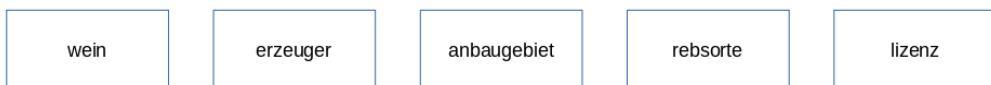


Abbildung 3.41: Erzeugte *Entity-Typen* mittels Programmcode von Listing 3.34 angewendet auf das Weingut Testmodell

→Homolka

Die Erstellung der *Attribute* erfolgt nahezu analog zu der Erstellung der *Entity-Typen*. Anstatt der Klasse `RectangleShape` ist die Klasse `EllipseShape` zu verwenden. Falls das *Attribut* ein *Primary-Key* ist, wird der Name des *Attributs* unterstrichen.

In Abbildung 3.42 sieht man die *Attribute* des *Entity-Typs* Wein:



Abbildung 3.42: Erzeugte Attribute von dem Entity-Typ Wein des Weingut Testmodells

Im Gegensatz zu Rechtecken oder Ellipsen gibt es keine vordefinierten Formen um *Beziehungstypen* durch eine Raute oder *Super-Sub-Beziehungstypen* durch ein gleichschenkeliges Dreieck darzustellen. An dieser Stelle kommt die Klasse `PolyPolygonShape` zum Einsatz. Mithilfe dieser Klasse und den Funktionen aus Listing 3.35 kann eine Form mit einer beliebigen Anzahl an Seiten erzeugt werden.

→Homolka

→Homolka

Listing 3.35: Programmcode in Java, um einen Rechteck mit beliebiger Anzahl an Eckpunkten zu erzeugen

```

1 public static XShape drawPolygon(XDrawPage slide, int x, int y, int radius, int nSides, String
      text) {
2     XShape polygon = addShape(slide, "PolyPolygonShape", 0, 0, 0, 0, text);
3     Point[] pts = genPolygonPoints(x, y, radius, nSides);
4     Point [][] polys = new Point[][] { pts };
5     setProperty(polygon, "PolyPolygon", polys);
6     return polygon;
7 }
8
9 public static XShape addShape(XDrawPage slide, String shapeType, int x, int y, int width, int
      height, String text) {
10    XShape shape = makeShape(shapeType, x, y, width, height, text);
11    if (shape != null){
12        slide.add(shape);
13    }
14    return shape;
15 }
16
17 public static XShape makeShape(String shapeType, int x, int y, int width, int height, String text)
      {
18    XShape shape = null;
19    try {
20        shape = createInstanceMSF(XShape.class, "com.sun.star.drawing." + shapeType);
21        shape.setPosition(new Point(x * 100, y * 100));
22        shape.setSize(new Size(width * 100, height * 100));
23    } catch (java.lang.Exception e) {
24        System.out.println("Unable to create shape: " + shapeType);
25    }
26    return shape;
27 }
28
29 private static Point[] genPolygonPoints(int x, int y, int radius, int nSides) {
30    if (nSides < 3) {
31        System.out.println("Too few sides; must be 3 or more");
32        nSides = 3;
33    } else if (nSides > 30) {
34        System.out.println("Too many sides; must be 30 or less");
35        nSides = 30;
36    }
37 }
```

```

38     Point[] pts = new Point[nSides];
39     double angleStep = Math.PI / nSides;
40     for (int i = 0; i < nSides; i++) {
41         pts[i] = new Point((int) Math.round(x * 100 + radius * 100 * Math.cos(i * 2 * angleStep)))
42             ,
43             (int) Math.round(y * 100 + radius * 100 * Math.sin(i * 2 * angleStep)));
44     }
45     return pts;
46 }
```

36

→Homolka

Durch den Aufruf der Funktion `drawPolygon` aus dem Listing 3.35, die die anderen Funktionen aus dieser Listing aufruft, wird ein Polygon erzeugt. Die Parameter der Funktion haben folgende Bedeutung:

- `XDrawPage slide`: beinhaltet die Seite in die das *Polygon* gezeichnet wird.
- `int x`: legt den Wert der x-Koordinate fest.
- `int y`: legt den Wert der y-Koordinate fest.
- `int radius`: legt die Größe der zu zeichnenden Form fest.
- `int nSides`: legt die Seitenanzahl der Form fest. Dieser Wert muss zwischen exklusive drei und exklusive dreißig liegen.
- `String text`: beinhaltet den Text, der in der Form angezeigt wird.

→Homolka

Das Ergebnis der Aufrufe

```

drawPolygon(xDrawPage, 10, 10, 8, 4, "erzeugt");
drawPolygon(xDrawPage, 20, 10, 8, 4, "beinhaltet");
drawPolygon(xDrawPage, 30, 10, 8, 3, "");
```

ist in Abbildung 3.43 dargestellt.

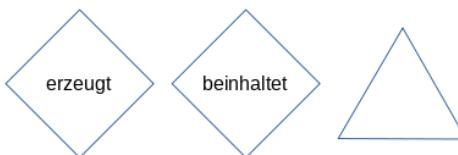


Abbildung 3.43: Zwei Beziehungstypen und ein Super-Sub-Beziehungstyp

→Homolka

Um einen *Entity-Typ* mit einem *Beziehungstyp* oder einem *Attribut* zu verbinden, wird die Klasse `XPropertySet` benötigt. Dieser Klasse wird ein Objekt der Klasse `ConnectorShape` mitgegeben. Der Programmcode aus Listing 4.1 zeigt, wie man zwei *Konnektoren* erzeugt und diese von dem *Beziehungstyp* zu zwei verschiedenen *Entity-Typen* verbindet:

Listing 3.36: Programmcode in Java, um zwei *Konnektoren* zu erzeugen und von einem *Beziehungstyp* zu zwei verschiedenen *Entity-Typen* zu binden

```

1
2 XPropertySet xConnector1PropSet = (XPropertySet)
3 UnoRuntime.queryInterface(XPropertySet.class, connector1);
```

³⁶Dr. Andrew Davison. *Java LibreOffice Programming*. Deutsch. URL: <https://de.wikipedia.org/wiki/LibreOffice>.

```

4 XPropertySet xConnector2PropSet = (XPropertySet)
5 UnoRuntime.queryInterface(XPropertySet.class, connector2);
6
7 xConnector1PropSet.setPropertyValue("StartShape", diamond);
8 xConnector1PropSet.setPropertyValue("EndShape", start_shape);
9 xConnector2PropSet.setPropertyValue("StartShape", diamond);
10 xConnector2PropSet.setPropertyValue("EndShape", end_shape);

```

Das Ergebnis aus Listing 4.1 wird in Abbildung 3.44 dargestellt.

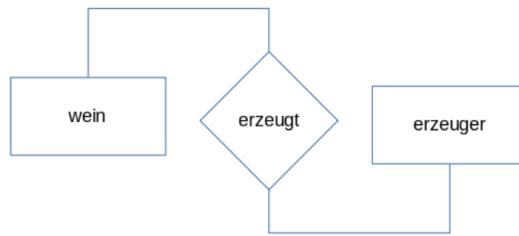


Abbildung 3.44: Ergebnis aus Listing 4

Während der Entwicklung über die *API* sind einige Probleme wie zum Beispiel das korrekte Darstellen von *Konnektoren* und *abhängigen Entity-Typen* aufgetreten, weshalb die Weiterentwicklung der Darstellung von *ERDs* über die *API* eingestellt wurde.

3.8.3.1.4 Vor- und Nachteile

→Homolka

Die *LibreOffice API* bietet eine Variante, um ein *LibreOffice*-Dokument zu bearbeiten ohne Modifikationen am Dateiformat vorzunehmen. Dies ist aber nicht der einzige Vorteil:

- Die *LibreOffice API* ist zu 100 Prozent kompatibel mit *OpenOffice*. Das bedeutet einerseits, dass das Zielprogramm sowohl ein *LibreOffice* Programm als auch ein *OpenOffice* Programm sein kann, und andererseits, dass es dem Anwendungsprogrammierer frei steht, welche Dokumentation er wählt.
- Darüber hinaus ist sowohl die *LibreOffice API* Dokumentation als auch die *OpenOffice API* Dokumentation sehr umfangreich.

37

Allerdings bringt diese Methode auch einige Nachteile mit sich:

- Die Dokumentation erstreckt sich über 1650 Seiten, was einen einfachen Einstieg in die *API* sehr schwierig macht.
- Des Weiteren ist die Dokumentation sehr unübersichtlich und unstrukturiert gestaltet.
- Die Nummerierung der Kapitel und Unterkapitel ist in der Dokumentation nicht angegeben.
- Die letzte Version der Dokumentation ist 2009 erschienen und somit veraltet.

³⁷ *OpenOffice Developer's Guide*. Englisch. URL: https://wiki.openoffice.org/w/images/d/d9/DevelopersGuide_OOo3.1.0.pdf.

- Es gibt trotz der langen Entwicklungsgeschichte von *OpenOffice* beziehungsweise *LibreOffice* wenige Anwendungsbeispiele, was ein Erlernen allein durch den Programmcode schwierig gestaltet.
- Die Generierung von *ERDs* über die *API* ist ressourcenintensiver und dauert länger als die Generierung über das Dateiformat.
- Selbst bei der einfachen Verwendung der *API* müssen oft viele Klassen und Konvertierungen zwischen Klassen erfolgen, um das gewünschte Ergebnis zu erhalten, was das Benutzen der *API* ebenfalls erschwert.
- Viele Objekte, die man in *LibreOffice Draw* händisch erzeugen kann, sind in der *API* nicht vorhanden, wie zum Beispiel ein Dreieck oder eine Raute. Diese müssen dann aufwendig durch die Klasse `PolyPolygonShape` erstellt werden, was einen höheren Programmieraufwand mit sich bringt.
- Um die *API* verwenden zu können, ist das *LibreOffice Software Development Kit* notwendig. Dieses muss in jeder Sitzung ausgeführt werden, was Performanceeinbußen mit sich bringt.

38

→Homolka

3.8.3.2 Generierung über das Dateiformat

3.8.3.2.1 Allgemeines

LibreOffice Draw verwendet das *Open Document Graphics* (ODG) Dateiformat, das 2006 als internationale Norm ISO/IEC 26300 veröffentlicht wurde. *ODG* basiert auf einem *XML-Vokabular*, dessen Elemente an den Standard HTML angelehnt sind. Eine *OpenDocument*-Datei besteht aus einer Sammlung mehrerer *XML-Dateien* und anderer Objekte wie Bilder oder Thumbnails, die zu einer Datei im ZIP-Format zusammengefasst werden.

Bei der Entpackung dieser ZIP-Datei sieht man folgende Dateien und Ordner:

```
Datei.odt
|
+-- META-INF
|   |
|   +- manifest.xml
|
+-- Thumbnails
|   |
|   +- thumbnail.png
|
+-- Pictures
|   |
|   +- picture.png
|
+-- mimetype
+-- content.xml
+-- styles.xml
+-- meta.xml
```

³⁸ *OpenOffice Developer's Guide*. Englisch. URL: https://wiki.openoffice.org/w/images/d/d9/DevelopersGuide_OOo3.1.0.pdf.

```
+-- settings.xml
```

39

Inhalt dieser Dateien:

→Homolka

- *manifest.xml*: In dieser Datei befindet sich eine Übersicht aller Dateien mit deren Pfaden und zulässigen Datentypen.
- *thumbnail.png*: Diese Datei zeigt eine Miniaturansicht der ersten Seite des Dokuments.
- Im Pictures-Ordner befinden sich alle eingebundenen Bilder des Dokuments. Falls keine Bilder eingebunden sind, existiert dieser Ordner nicht.
- *mimetype*: Diese Datei ist eine Textdatei, deren einziger Inhalt eine Zeile mit dem Typ der *LibreOffice*-Datei ist. Beim Beispiel einer *Draw*-Datei steht folgender Inhalt in der *Mimetype*-Datei:
`application/vnd.oasis.opendocument.graphics`
- *settings.xml*: In dieser Datei befinden sich sämtliche Einstellungsmöglichkeiten, die *LibreOffice Draw* anbietet. Darunter fallen zum Beispiel die Maßeinheit, Name und Setup der Drucker, die Skalierung und die Sichtbarkeit der einzelnen Ebenen.
- *content.xml*: In dieser Datei befindet sich der eigentliche Inhalt des Dokuments. Durch Betrachtung einer *Draw*-Datei ohne Inhalt ergibt sich die Baumstruktur:

```
Element: office:document-content
|
+-- Attribut: xmlns:--
+-- Element: office:scripts
+-- Element: office:font-face-decls
|
+-- Element: style:font-face
|
+-- Attribut: style:name
+-- Attribut: style:font-family
+-- Attribut: style:font-family-generic
+-- Attribut: style:font-pitch
+-- Element: office:automatic-styles
|
+-- Element: style:style
|
+-- Attribut: style:name
+-- Attribut: style:family
+-- Element: office:body
|
+-- Element: office:drawing
|
+-- Element: draw:page
|
+-- Attribut: draw:name
+-- Attribut: draw:style-name
+-- Attribut: draw:master-page-name
```

³⁹ LibreOffice Wikipedia. Englisch. URL: <https://fivedots.coe.psu.ac.th/~ad/jlop/>.

→Homolka

Der komplette Inhalt wird als *XML-Kindelement* von dem *XML-Element* `draw:page` eingefügt. In dem *XML-Element* `office:document-content` befinden sich mehrere *XML-Attribute* mit diversen *Namensräumen*, die aus Gründen der Übersichtlichkeit weggelassen wurden.

- *styles.xml*: Diese Datei beinhaltet die grafischen Informationen der Formatvorlagen für Texte und grafischen Elemente wie zum Beispiel Textfarbe, Textgröße, Füllungsfarbe, Abstand an den Seiten, Rahmenbreite, etc.
- *meta.xml*: In dieser Datei befinden sich Übersichtsinformationen der *LibreOffice*-Datei wie zum Beispiel der Name des Erzeugers, das Datum, an dem das Dokument erstellt wurde, die Sprache des Dokuments und die Bearbeitungsdauer.

40

→Homolka

Wie man in der obigen Auflistung sieht, ist das Dateiformat *ODG* komplex und daher aufwändig zu modifizieren. *LibreOffice* bietet allerdings auch die Möglichkeit eine *Draw*-Datei in dem Dateiformat *Flat Open Document Graphics (FODG)* zu speichern. In diesem Format werden die Dateien aus dem gezippten Ordner zu einer Datei zusammengefasst. Diese Variante der Speicherung ist für die Manipulation des Dateiformats am besten geeignet, da nur eine einzige Datei manipuliert werden muss. Für die Manipulation über das Dateiformat wird die Programmiersprache *Python* verwendet. Diese bringt im Zusammenhang mit der gewählten Diplomarbeit mehrere Vorteile mit sich:

- In der Regel ist es einfacher und schneller kleinere Programme wie diese Diplomarbeit in *Python* zu entwickeln.
- *Python* ist eine plattformunabhängige Sprache, die auf fast allen Betriebssystemen vorhanden ist.

3.8.3.2.2 XERML-Daten Aufbereitung in Python

→Homolka

In der Programmiersprache *Python* gibt es mehrere Optionen eine *XML-Datei* zu *parsen* und dessen Daten in ein geeignetes Format für die Weiterverarbeitung zu bringen. Im Zuge der Diplomarbeit ist die Entscheidung auf *LXML* gefallen. *LXML* ist eine *Python*-Bibliothek und bietet einen sicheren und bequemen Zugriff auf die *libxml2* und *libxslt* Bibliothek mithilfe der *ElementTree API*. Die Daten aus der *XML-Datei* werden in einem Array mit dem selben Aufbau wie in Kapitel 3.8.3.1.2 gespeichert.

3.8.3.2.3 Darstellung der grafischen Elemente

Alle Annahmen und Informationen dieses Kapitels entstammen keiner Quelle und wurden durch *Reverse Engineering* in Erfahrung gebracht.

→Homolka

Bei jeden Aufruf ein komplettes *LibreOffice* Dokument zu generieren, bedeutet viel Programmieraufwand und beeinträchtigt die Performance. Aus diesem Grund wird das zuvor erstellte leere *LibreOffice* Dokument *empty1.fodg* als Ausgangsdokument verwendet. Um das Dokument auf die Darstellung von *ERDs* zu optimieren, wurden kleinere Modifikationen vorgenommen:

- Das Dokument hat einen *Margin* von 1.9 cm.
- Sämtliche *gr*, *t* und *p* *XML-Elemente* wurden angepasst.

⁴⁰ *LibreOffice Wikipedia*. Englisch. URL: <https://fivedots.coe.psu.ac.th/~ad/jlop/>.

- Die Informationen über das Format, die Breite und die Höhe wurden entfernt.

→Homolka

Sämtliche grafische Formen lassen sich mit dem *XML-Element* `custom-shape` erzeugen. Dafür muss man das *XML-Element* als *XML-Kindelement* von dem *XML-Element* `draw:page` einfügen. Das *XML-Element* `custom-shape` benötigt ebenfalls zwei *XML-Kindelemente* um den Text in der grafischen Form anzusehen und um die grafischen Eigenschaften der Form festzulegen. Der Aufbau der *XML-Elemente* und deren *XML-Attribute* bei der Erstellung von Rechtecken für *Entity-Typen* und Ellipsen für *Attribute* sieht wie folgt aus:

```
Element: draw:page
|
+-- Element: draw:custom-shape
  |
  +-- Attribut: draw:style-name
  +-- Attribut: draw:text-style-name
  +-- Attribut: draw:id
  +-- Attribut: draw:layer
  +-- Attribut: svg:width
  +-- Attribut: svg:height
  +-- Attribut: svg:x
  +-- Attribut: svg:y
  +-- Element: draw:enhanced-geometry
    |
    +-- Attribut: svg:viewBox
    +-- Attribut: draw:mirror-horizontal
    +-- Attribut: draw:mirror-vertical
    +-- Attribut: draw:type
    +-- Attribut: draw:enhanced-path
    +-- Attribut: svg:glue-points
    +-- Attribut: draw:text-areas
    +-- Element: text:p
      |
      +-- Attribut: text:style-name
      +-- Element: text:span
        |
        +-- Attribut: text:style-name
```

→Homolka

Einige der *XML-Attribute* sind für die Erstellung eines *ERDs* unrelevant beziehungsweise werden immer mit den selben Werten gefüllt:

- Das Attribut `draw:layer` beschreibt die Ebene der Form. Das Ausgangsdokument beinhaltet folgende Ebenen: *layout*, *background*, *backgroundobjects*, *controls* und *measurelines*. Die Ebenen sind absteigend in Richtung Hintergrund geordnet. Für die Erstellung eines *ERDs* sind mehrere Ebenen nicht vorgesehen, weshalb dem *XML-Attribut* `draw:layer` immer der Wert *layout* zugeordnet wird.
- Mit dem *XML-Attribut* `svg:viewBox` wird eine fixe Pixelbreite und Pixelhöhe festgelegt. Dem Attribut wird immer der Wert 0 0 21600 21600 zugewiesen.
- Die *XML-Attribute* `draw:mirror-horizontal` und `draw:mirror-vertical` bekommen den Wert *false* zugewiesen.

- Das *XML-Attribut* `draw:enhanced-path` bekommt den Wert U 10800 10800 10800 10800 0 360 Z N.
- Das *XML-Attribut* `svg:glue-points` legt die Klebepunkte der Form fest, an denen sich *Konnektoren* anheften können. Diese haben den Wert 10800 0 3163 3163 0 10800 3163 18437 10800 21600 18437 18437 21600 10800 18437 3163.
- Das *XML-Attribut* `draw:text-areas` bekommt den Wert 3163 3163 18437 18437.

→Homolka

Der Text wird unter *LXML* mithilfe der Methode `text` bei dem *XML-Element* `text:span` festgelegt. Das *ERD* kann mit Farbe oder farblos gezeichnet werden. Bei einer Darstellung in Farbe werden die Hexadezimalwerte 89D8D6 für *Entity-Typen*, 89BED8 für *Attribute* und 93D889 für *Beziehungstypen* verwendet.

→Homolka

Sämtliche Informationen über das Aussehen der Formen werden in den *XML-Elementen* `style:style` gespeichert. Es gibt drei verschiedene Kategorien. Die `gr` *XML-Elemente* speichern die grafischen Informationen. Die *XML-Elemente* `p` und `t` speichern Informationen über die Darstellung von Text. Ein `gr` *XML-Element* könnte wie folgt aussehen:

```
<style:style style:name="gr1" style:family="graphic"
            style:parent-style-name="standard">
  <style:graphic-properties svg:stroke-width="0.03cm"
                           svg:stroke-color="#000000" draw:marker-start-width="0.245cm"
                           draw:marker-end-width="0.245cm" draw:fill="solid"
                           draw:fill-color="#ffffff"
                           draw:textarea-horizontal-align="justify"
                           draw:textarea-vertical-align="middle"
                           draw:auto-grow-height="false" fo:padding-top="0.14cm"
                           fo:padding-bottom="0.14cm" fo:padding-left="0.265cm"
                           fo:padding-right="0.265cm" style:protect="size"/>
</style:style>
```

→Homolka

Um *Entity-Typen* darzustellen, muss das *XML-Attribut* `draw:type` des *XML-Elements* `draw:enhanced-geometry` den Wert *rectangle* beinhalten. Falls die Darstellung in Farbe erwünscht ist, bekommen die *XML-Attribute* die Werte aus der Tabelle 3.1.

Element	Attribut	Wert
draw:custum-shape	draw:style-name	gr-ent
draw:custum-shape	draw:text-style-nam	P-ent
text:p	text:style-name	P1
text:span	text:style-name	T6

Tabelle 3.1: Tabelle der *XML-Attributwerte* um *Entity-Typen* farbig darzustellen

→Homolka

Ist ein Darstellen in Farbe nicht erwünscht, bekommen die *XML-Attribute* die Werte aus Tabelle 3.2

Element	Attribut	Wert
draw:custom-shape	draw:style-name	gr1
draw:custom-shape	draw:text-style-name	P8
text:p	text:style-name	P1
text:span	text:style-name	T1

Tabelle 3.2: Tabelle der *XML-Attributwerte* um *Entity-Typen* farblos darzustellen

Die Breite und Höhe des Rechtecks werden in den *XML-Attributen* `svg:width` und `svg:height` festgelegt. Die Breite ist auf 10 cm und die Höhe ist auf 6 cm festgelegt. Ein *Entity-Typ* könnte in dem Dateiformat *FODG* wie folgt aussehen:

```
<draw:custom-shape draw:style-name="gr_ent" draw:text-style-name="P_ent"
    draw:id="wein" draw:layer="layout" svg:width="10cm"
    svg:height="6cm" svg:x="159.6cm"
    svg:y="82.7320000000001cm">
    <text:p text:style-name="P1">
        <text:span text:style-name="T6">wein</text:span>
    </text:p>
    <draw:enhanced-geometry svg:viewBox="0 0 21600 21600"
        draw:mirror-horizontal="false"
        draw:mirror-vertical="false"
        draw:type="rectangle"
        draw:enhanced-path="M 0 0 L 21600 0 21600 21600 0 21600 0 0 Z N" />
</draw:custom-shape>
```

→Homolka

In Abbildung 3.45 ist links das Ergebnis aus dem oben abgebildeten *FODG* zu sehen und rechts ein *Entity-Typ* ohne Farbe.



Abbildung 3.45: Links: Farbiger Entity-Typ in *LibreOffice Draw*. Rechts: Farbloser Entity-Typ in *LibreOffice Draw*

→Homolka

Die Darstellung von *Attributen* funktioniert analog bis auf den Unterschied, dass das *XML-Attribut* `draw:type` den Wert `ellipse` beinhaltet. Falls ein Darstellen in Farbe erwünscht, bekommen die *XML-Attribute* die Werte aus Tabelle 3.3.

Element	Attribut	Wert
draw:custom-shape	draw:style-name	gr4
draw:custom-shape	draw:text-style-name	P19
text:p	text:style-name	P1
text:span	text:style-name	T6

Tabelle 3.3: Tabelle der *XML-Attributwerte* um *Attribute* farbig darzustellen

→Homolka

Ist ein Darstellen in Farbe nicht erwünscht, bekommen die *XML-Attribute* die Werte aus Tabelle 3.4.

Element	Attribut	Wert
draw:custom-shape	draw:style-name	gr1
draw:custom-shape	draw:text-style-name	P8
text:p	text:style-name	P1
text:span	text:style-name	T1

Tabelle 3.4: Tabelle der *XML-Attributwerte* um *Attribute* farblos darzustellen

Ein *Attribut* könnte in dem Dateiformat *FODG* wie folgt aussehen:

```
<draw:custom-shape draw:style-name="gr4" draw:text-style-name="P5"
    draw:id="weinname" draw:layer="layout" svg:width="10cm"
    svg:height="6cm" svg:x="182.76cm" svg:y="69.76cm">
    <text:p text:style-name="P1">
        <text:span text:style-name="T4">name</text:span>
    </text:p>
    <draw:enhanced-geometry svg:viewBox="0 0 21600 21600"
        svg:glue-points="10800 0 3163 3163 0 10800 3163 18437 10800
        21600 18437 18437 21600 10800 18437 3163"
        draw:type="ellipse" draw:text-areas="3163 3163 18437 18437"
        draw:enhanced-path="U 10800 10800 10800 10800 0 360 Z N" />
</draw:custom-shape>
```

→Homolka

Falls ein *Attribut* ein *Primary-Key* ist, wird der Name des *Attributs* unterstrichen. Folgende Verwendung von *XML-Attributen* ist dazu notwendig. Die *XML-Attribute* bekommen die Werte aus Tabelle 3.5, falls ein Darstellen in Farbe erwünscht ist.

Element	Attribut	Wert
draw:custum-shape	draw:style-name	gr4
draw:custum-shape	draw:text-style-nam	P5
text:p	text:style-name	P1
text:span	text:style-name	T4

Tabelle 3.5: Tabelle der *XML-Attributwerte* um *Primary-Key-Attribute* in Farbe darzustellen

→Homolka

Falls die Darstellung in Farbe nicht erwünscht ist, bekommen die *XML-Attribute* die Werte aus Tabelle 3.6.

Element	Attribut	Wert
draw:custum-shape	draw:style-name	gr1
draw:custum-shape	draw:text-style-nam	P7
text:p	text:style-name	P1
text:span	text:style-name	T4

Tabelle 3.6: Tabelle der *XML-Attributwerte* um *Primary-Key-Attribute* farblos darzustellen

→Homolka

In Abbildung 3.46 ist links außen ein farbiges *Attribut* zu sehen, links mittig ein farbiges *Primary-Key-Attribut*, rechts mittig ein *Attribut* ohne Farbe und rechts außen ein farbloses *Primary-Key-Attribut*:

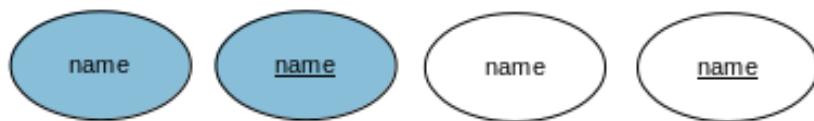


Abbildung 3.46: Links außen: Farbiges *Attribut*. Links innen: Farbiges *Primary-Key-Attribut*. Rechts innen: Farbloses *Attribut*. Rechts außen: Farbloses *Primary-Key-Attribut*

→Homolka

Für die Darstellung von Rauten, um *Beziehungstypen* darzustellen, benötigt das *XML-Attribut* `draw:type` den Wert `diamond`. Falls ein Darstellen in Farbe erwünscht, bekommen die *XML-Attribute* die Werte aus Tabelle 3.7.

Element	Attribut	Wert
draw:custum-shape	draw:style-name	gr-rel
draw:custum-shape	draw:text-style-nam	P-rel
text:p	text:style-name	P1
text:span	text:style-name	T6

Tabelle 3.7: Tabelle der *XML-Attributwerte* um *Beziehungstypen* farblos darzustellen

→Homolka

Ist die Darstellung in Farbe nicht erwünscht ist, bekommen die *XML-Attribute* die Werte aus Tabelle 3.8.

Element	Attribut	Wert
draw:custum-shape	draw:style-name	gr1
draw:custum-shape	draw:text-style-nam	P8
text:p	text:style-name	P1
text:span	text:style-name	T1

Tabelle 3.8: Tabelle der *XML-Attributwerte* um *Beziehungstypen* farblos darzustellen

Konnektoren lassen sich mit dem *XML-Element* `draw:connector` erzeugen. Um die *Konnektoren* mit einer Start- und Endform verbinden zu können und Text auf dem *Konnektor* darzustellen, wird folgender Aufbau des Elements benötigt:

```
Element: draw:page
|
+-- Element: draw:connector
  |
  +-- Attribut: draw:style-name
  +-- Attribut: draw:text-style-name
  +-- Attribut: draw:id
  +-- Attribut: draw:layer
  +-- Attribut: draw:type
  +-- Attribut: start-shape
  +-- Attribut: end-shape
  +-- Attribut: svg:d
  +-- Attribut: svg:viewBox
  +-- Element: text:p
    |
    +-- Attribut: text:style-name
  +-- Element: text:span
    |
    +-- Attribut: text:style-name
```

→Homolka

Die *XML-Attribute* `draw:start-shape` und `draw:end-shape` legen dabei die Start- und Endform fest, wobei sie als Wert die ID der zu verbindenden Form bekommen. Der Text,

der auf dem *Konnektor* steht, wird wie bei einem *Entity-Typen* festgelegt. Ein *Konnektor*, der von einem *Entity-Typ* zu einem *Beziehungstyp* führt, könnte in dem Dateiformat *FODG* wie folgt aussehen:

```
<draw:connector draw:style-name="gr2" draw:text-style-name="P3"
    draw:layer="layout" draw:type="line"
    draw:start-shape="beinhaltet"
    draw:end-shape="wein"
    svg:d="M21575 305012125 750"
    svg:viewBox="0 0 2126 751">
<text:p text-style-name="P1">
    <text:span text-style-name="T1">(1, n)</text:span>
</text:p>
</draw:connector>
```

→Homolka

Bei *Konnektoren* zwischen einem *Entity-Typ* oder *Beziehungstyp* und einem *Attribut* wird kein Text benötigt und daher entfallen die *XML-Elemente* `text:p` und `text:span`.

In Abbildung 3.48 wird die Beziehung zwischen den *Entity-Typen* Anbaugebiet und Erzeuger aus dem Weingut Testmodell.



Abbildung 3.47: Beziehung zwischen den *Entity-Typen* Anbaugebiet und Erzeuger aus dem Weingut Testmodell

→Homolka

Die komplexeste Form bei der Erstellung von *ERDs* ist ein *Super-Sub-Beziehungstyp*, der als gleichschenkliges Dreieck dargestellt wird. Der Aufbau der *XML-Elemente* sieht für diese Form wie folgt aus:

```
Element: draw:page
|
+-- Element: draw:custom-shape
  |
  +-- Attribut: draw:style-name
  +-- Attribut: draw:text-style-name
  +-- Attribut: draw:id
  +-- Attribut: draw:layer
  +-- Attribut: svg:width
  +-- Attribut: svg:height
  +-- Attribut: svg:x
  +-- Attribut: svg:y
  +-- Element: draw:enhanced-geometry
    |
    +-- Attribut: svg:viewBox
```

```

+-- Attribut: draw:mirror-horizontal
+-- Attribut: draw:mirror-vertical
+-- Attribut: draw:type
+-- Attribut: draw:enhanced-path
+-- Attribut: svg:glue-points
+-- Element: draw:equation
|
+-- Attribut: draw:name
+-- Attribut: draw:formula

```

→Homolka

Das *XML-Element* `draw:equation` wird dabei achtmal verwendet. (siehe Tabelle 3.10)

Element	Wert von <code>draw:name</code>	Wert von <code>draw:formula</code>
Erstes Element <code>draw:equation</code>	f0	\$0
Zweites Element <code>draw:equation</code>	f1	\$0 /2
Drittes Element <code>draw:equation</code>	f2	?f1 +10800
Viertes Element <code>draw:equation</code>	f3	\$0 *2/3
Fünftes Element <code>draw:equation</code>	f4	?f3 +7200
Sechstes Element <code>draw:equation</code>	f5	21600-?f0
Siebtes Element <code>draw:equation</code>	f6	?f5 /2
Achtes Element <code>draw:equation</code>	f7	21600-?f6

Tabelle 3.9: Werte von den `draw:equation` Elementen

→Homolka

Ein *Super-Sub-Beziehungstyp* mit den Werten aus Tabelle 3.10 wird in Abbildung 3.48 gezeigt.

Falls der *Beziehungstyp disjunkt* ist, bekommt der Maximalwert der Beziehung zu dem *Super-Typen* den Wert 1 und das Dreieck ist *weiß* gefüllt. Falls der *Beziehungstyp nicht disjunkt* ist, bekommt der Maximalwert der Beziehung zu dem *Super-Typen* den Wert der Anzahl der *Sub-Typen* und das Dreieck ist *schwarz* gefüllt.

Falls der *Beziehungstyp total* ist, bekommt der Minimalwert der Beziehung zu dem *Super-Typen* den Wert 1 und der *Konnektor* zu dem *Super-Typ* ist strichliert, da die eigentliche Darstellung mit zwei parallel verlaufenden Linien unter *Draw* nicht möglich ist. Falls der *Beziehungstyp partiell* ist, bekommt der Minimalwert des *Konnektors* zu dem *Super-Typen* den Wert 0 und der *Konnektor* zu dem *Super-Typ* ist durchgezogen. Die *Beziehungskardinalität* zu den *Sub-Typen* ist immer (1, 1).

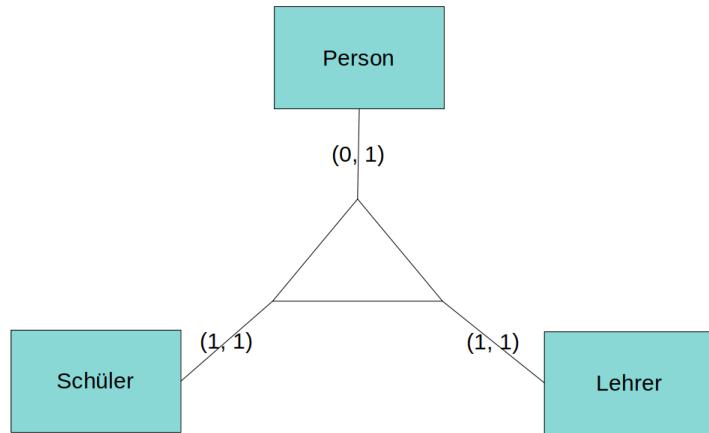


Abbildung 3.48: Disjunkter und partieller Super-Sub-Beziehungstyp erstellt mit den Werten aus Tabelle 3.10 angewendet auf das Testmodell SIS

→Homolka

Abhängige Entity-Typen und Identifizierende Beziehungstypen werden mit einer doppelten Umrahmung dargestellt. Dafür werden zwei Rechtecke in das XML-Element `draw:g` als XML-Kindelemente eingefügt. Die Breite und Höhe des zusätzlichen Rechtecks beträgt 9 cm beziehungsweise 5 cm. Zusätzlich sind die x- und y-Koordinate um 0.5 cm zu verschieben, um den Abstand zwischen den beiden Rechtecken auf jeder Seite gleich zu halten. Ein abhängiger Entity-Typ könnte in dem Dateiformat *FODG* wie folgt aussehen:

```

<draw:g draw:style-name="gr7" draw:id="kveranst">
    <draw:custom-shape draw:style-name="gr1" draw:text-style-name="P8"
        draw:layer="layout" svg:width="10cm" svg:height="6cm"
        svg:x="130.0cm" svg:y="32.62cm">
        <text:p text-style-name="P1">
            <text:span text-style-name="T1"/>
        </text:p>
        <draw:enhanced-geometry svg:viewBox="0 0 21600 21600"
            draw:mirror-horizontal="false" draw:mirror-vertical="false"
            draw:type="rectangle"
            draw:enhanced-path="M 0 0 L 21600 0 21600 21600 0 21600 0 0 Z N"/>
    </draw:custom-shape>
    <draw:custom-shape draw:style-name="gr_ent"
        draw:text-style-name="P_ent" draw:id="kveranst"
        draw:layer="layout" svg:width="9cm" svg:height="5cm"
        svg:x="130.5cm" svg:y="33.12cm">
        <text:p text-style-name="P1">
            <text:span text-style-name="T6">kveranst</text:span>
        </text:p>
        <draw:enhanced-geometry svg:viewBox="0 0 21600 21600"
            draw:mirror-horizontal="false" draw:mirror-vertical="false"
            draw:type="rectangle"
  
```

```

    draw:enhanced-path="M 0 0 L 21600 0 21600 21600 0 21600 0 0 Z N"/>
</draw:custom-shape>
</draw:g>

```

→Homolka

Das *Primary-Key-Attribut* des *abhängigen Entity-Typen* wird strichliert unterstrichen. Dafür benötigen die *XML-Attribute* aus Tabelle 3.10 jene Werte, falls das Darstellen in Farbe erwünscht ist.

Element	Attribut	Wert
draw:custum-shape des äußeren Rechtecks	draw:style-name	gr1
draw:custum-shape des äußeren Rechtecks	draw:text-style-nam	P8
text:p des äußeren Rechtecks	text:style-name	P1
text:span des äußeren Rechtecks	text:style-name	T1
draw:custum-shape des inneren Rechtecks	draw:style-name	gr-ent
draw:custum-shape des inneren Rechtecks	draw:text-style-nam	P-ent
text:p des inneren Rechtecks	text:style-name	P1
text:span des inneren Rechtecks	text:style-name	T6

Tabelle 3.10: Tabelle der *XML-Attributwerte* um *Primary-Key-Attribute* von *abhängigen Entity-Typen* in Farbe darzustellen

→Homolka

Ist die Darstellung in Farbe nicht erwünscht ist, bekommen die *XML-Attribute* die Werte aus Tabelle 3.11.

Element	Attribut	Wert
draw:custum-shape des äußeren Rechtecks	draw:style-name	gr1
draw:custum-shape des äußeren Rechtecks	draw:text-style-nam	P8
text:p des äußeren Rechtecks	text:style-name	P1
text:span des äußeren Rechtecks	text:style-name	T1
draw:custum-shape des inneren Rechtecks	draw:style-name	gr1
draw:custum-shape des inneren Rechtecks	draw:text-style-nam	P8
text:p des inneren Rechtecks	text:style-name	P1
text:span des inneren Rechtecks	text:style-name	T1

Tabelle 3.11: Tabelle der *XML-Attributwerte* um *Primary-Key-Attribute* von *abhängigen Entity-Typen* farblos darzustellen

→Homolka

Die Abbildung 3.50 zeigt einen *identifizierenden Beziehungstyp* und einen *abhängigen Entity-Typen* mit dessen *Primary-Key-Attribut*.

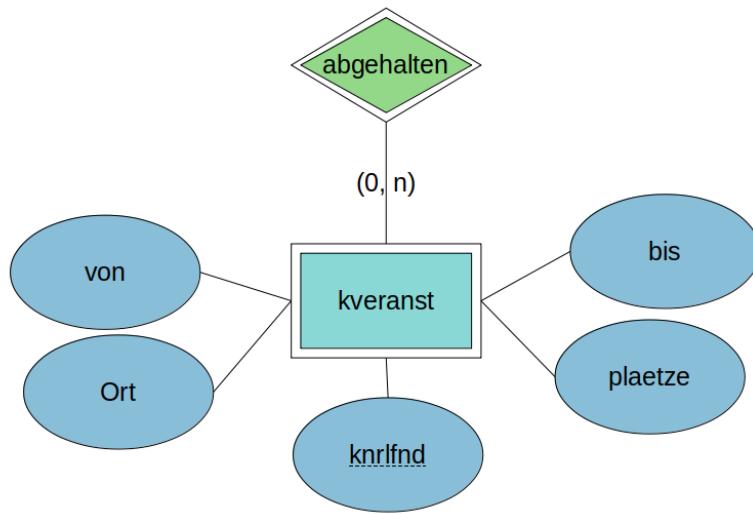


Abbildung 3.49: Identifizierender Beziehungstyp und ein abhängigen Entity-Typ mit dessen Primary-Key-Attribut

3.8.3.2.4 IDs

→Homolka

Eine zentrale Rolle in der Generierung von *ERDs* über das Dateiformat spielen die IDs der Formen. Ohne diese wäre es nicht möglich *Konnektoren* an *Entity-Typen*, *Attributen* oder *Beziehungstypen* zu verankern. Da eine ID eindeutig sein muss, reicht der Name eines *Attributes* oder eines *Beziehungstyps* meist nicht aus, da dieser meist öfter als einmal vorkommen kann. Aus diesem Grund werden die IDs aus mehreren Faktoren zusammengesetzt:

- Die ID eines *Entity-Typs* ist dessen Name. Dieser muss eindeutig sein und reicht daher aus.
- Die ID eines Attributs setzt sich aus dem Namen des zugehörigen *Entity-Typs* und dem Namen des Attributs zusammen.
- Die ID eines *Beziehungstyps* ist dessen Name. Dieser muss eindeutig sein und reicht daher aus.
- Die ID eines *Beziehungsattributs* setzt sich aus dem Namen der zugehörigen Beziehung und dem Namen des Attributs zusammen.
- Die ID einer *Super-Sub-Beziehung* setzt sich aus dem Namen der *Super-Sub-Beziehung* und dem Suffix *super-sub* zusammen.

3.8.3.2.5 Vor- und Nachteile

→Homolka

Die Erstellung eines *ERDs* in *LibreOffice Draw* über das Dateiformat bringt folgende Vorteile mit sich:

- Durch *Reverse Engineering* lässt sich sehr schnell die Bedeutung der einzelnen *XML-Elemente* und *XML-Attribute* herausfinden, was ein schnelles Einlesen in das Dateiformat ermöglicht.
- Die Erzeugung einer *XML-Datei* in *Python* ist sehr performant im Vergleich zu der Verwendung der *LibreOffice API* (siehe Abschnitt 3.8.3.3).

Allerdings bringt diese Variante auch einige Nachteile mit sich:

- Die Modifikation des Dateiformats ist sehr heikel. Ein falsches Einrücken oder das Erzeugen eines nicht vorhandenen *XML-Attributs* reicht, um die *FODG*-Datei nicht mehr öffnen zu können.

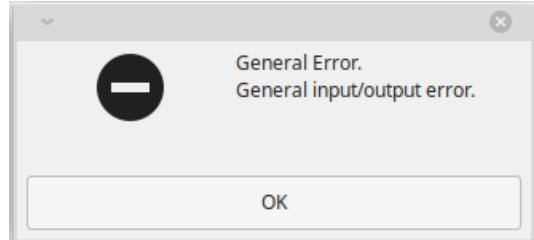


Abbildung 3.50: Fehlermeldung in *LibreOffice Draw*, falls das Dateiformat ungültig ist.

- Es gibt keine Dokumentation, in der das Dateiformat beschrieben steht.

3.8.3.3 Performancevergleich

→Homolka

Bei der Generierung der Testmodelle ist ein klarer Performanceunterschied erkennbar. Um diesen Unterschied zu veranschaulichen, wurde jedes Testmodell in beiden Generierungsvarianten generiert und dabei die benötigte Zeit gemessen. Um mögliche Ausreißer zu eliminieren, wurde jedes Testmodell fünfmal pro Generierungsvariante generiert und danach wurde der Durchschnitt gebildet. Die Testmodelle wurden in Farbe und mit Attributen generiert. Die Ergebnisse des Vergleichs sind in Tabelle 3.12 veranschaulicht.

Testmodell	Benötigte Zeit API	Benötigte Zeit Dateiformat
AAA	29,61 Sekunden	0,52 Sekunden
Fußball	25,02 Sekunden	0,31 Sekunden
Kinokette	28,50 Sekunden	0,46 Sekunden
Mondial	93,79 Sekunden	12,03 Sekunden
Rettungsstelles	32,95 Sekunden	0,72 Sekunden
SF	40,32 Sekunden	0,16 Sekunden
SIS	29,83 Sekunden	0,45 Sekunden
Tankstelle	25,99 Sekunden	0,36 Sekunden
Weingut	18,86 Sekunden	0,15 Sekunden

Tabelle 3.12: Benötigte Zeit jedes Testmodells in den beiden Generierungsvarianten.

→Homolka

Folgende Soft- beziehungsweise Hardware wurde für die Generierung der Testmodelle verwendet:

- OS: Linux Mint virtualisiert mittels Oracle VM VirtualBox
- CPU: Intel Core i5-6600k @ 3,9GHz
- GPU: GeForce GTX 980 Ti
- RAM: 16GB @ 2666MHz

3.8.4 Layout

→Homolka

Um ein aufwendiges Nachbearbeiten des *ERDs* zu verhindern, wird das *ERD* mit einem Layout erzeugt. Als Layoutalgorithmus wurde *Graphviz* verwendet. (siehe Kapitel 3.4.4)

3.9 Git

→Fischbacher

Bei einem Projekt mit diesem Umfang ist es von Vorteil jegliche Art von Versionsverwaltung zu verwenden. Außer Git hätten sich noch andere Versionsverwaltungssysteme angeboten wie z.B.:⁴¹

- Mercurial
- Darcs
- Fossil
- Bazaar

Jedoch wurde für dieses Projekt gezielt Git verwendet. Git ist unter den Versionverwaltungssystemen das gängigste und deswegen existiert dafür auch der meiste Support. Außerdem verwendet Trac, die Projektorganisations-Web-App die in diesem Projekt verwendet wurde, Git deswegen waren die anderen Möglichkeiten nicht mit Git gleichwertig.

Durch gezieltes anlegen von Branches wurde es ermöglicht jedem Entwickler eine vollkommen von den Anderen unabhängige Entwicklungsumgebung zu bieten. Diese sogenannten Feature-Branches ermöglichen es, unabhängig vom derzeitigen Entwicklungsstands des Projekts, es zu jederzeit eine funktionierende Version der Software am Master-Branch vorlag.

Die Daten lagen zu jeder Zeit auf dem Server, den der Auftraggeber bereitgestellt hat, vor und ermöglichte dadurch eine einfachen Zugriff. Mit einem unabhängigen Server ist es den Entwicklern leicht gefallen von zu Hause und in der Schule immer am neusten Stand zu sein.

⁴¹ *Git*.

3.10 Trac

→Fischbacher

Trac ist eine Software die bei der Entwicklung und Organisation des Projekts eine große Hilfe ist. Dadurch wird ein Interface für Versionsverwaltung geboten, sowie ein für Scrum ideales Ticketsystem. Außerdem bietet es einen Aktivitätsüberblick und einen Überblick über den aktuellen Stand des Projekts in Form von einer Roadmap.⁴² Durch das Ticketsystem kann man mit Leichtigkeit die verschiedenen Aufgaben verteilen und feststellen ob sie fertiggestellt wurden.

The screenshot shows a Trac ticket interface. At the top, it says '#6 new User Story'. To the right, it indicates the ticket was opened 5 months ago. The main title of the ticket is 'UD002 Als Anwender möchte ich dass mein generiertes ERD mit einem gut lesbaren Layout generiert wird'. Below the title, there are several fields: 'Reported by' (Nicolas Homolka), 'Priority' (1), 'Component' (Diplomarbeit), 'Keywords', 'Owned by' (Product Backlog DA), 'Milestone', 'Version', and 'Cc'. Under the 'Description' section, the text reads: 'Das generierte ERD soll in einem gut lesbaren Layout ausgegeben werden'. There is also a 'Reply' button next to the description.

Abbildung 3.51: Ticket im Trac

Trac unterstützt WikiFormatting, mithilfe dieses Merkmals kann man eine angenehme und leicht leserliche Struktur erschaffen.

Davon war das erstellen von Links auf interne Ressourcen sowie auf externe Ressourcen eine der meist verwendeten Funktionen.

Links	<code>http://trac.edgewall.org</code> <code>WikiFormatting (CamelCase)</code>	<code>→ http://trac.edgewall.org</code> <code>WikiFormatting (CamelCase)</code>
TracLinks	<code>wiki:WikiFormatting</code> , <code>wiki:"WikiFormatting"</code> <code>#1 (ticket)</code> , <code>[1] (changeset)</code> , <code>{1} (report)</code> <code>ticket:1</code> , <code>ticket:1#comment:1</code> , <code>comment:1:ticket:1</code> <code>Ticket [ticket:1]</code> , <code>[ticket:1 ticket one]</code> <code>Ticket [[ticket:1]], [[ticket:1 ticket one]]</code>	<code>wiki:WikiFormatting</code> , <code>wiki:"WikiFormatting"</code> <code>#1 (ticket)</code> , <code>[1] (changeset)</code> , <code>{1} (report)</code> <code>ticket:1</code> , <code>ticket:1#comment:1</code> , <code>comment:1:ticket:1</code> <code>Ticket #, ticket one</code> <code>Ticket #, ticket one</code>

Abbildung 3.52: Dokumentation für Links im Trac

⁴² The Trac Project.

3.11 Sphinx

Das Werkzeug *Sphinx* wird verwendet, um eine Dokumentation zu erstellen. Diese wird dabei in gutem Layout automatisch generiert. Der Benutzer kann nach der Erstellung noch händische Änderungen vornehmen, falls er noch weitere Informationen zu der Dokumentation hinzufügen möchte oder der generierte Inhalt nicht korrekt ist.

→Prinz

Sphinx wurde ursprünglich für die *Python Dokumentation* erstellt und bietet einige Möglichkeiten, eine Dokumentation für Softwareprojekte in verschiedenen Sprachen zu generieren⁴³.

Die erstellte Dokumentation kann als Dokumenttypen folgende Formate annehmen³¹:

- HTML
- LaTeX
- ePub
- Texinfo
- Manual pages
- plain text

3.11.1 Generierung der Dokumentation

Um die Dokumentation automatisch generieren zu können, muss das Skript `sphinx-quickstart` ausgeführt werden. Dieses Skript erstellt ein *Source*-Verzeichnis und die Datei `conf.py`, in der die sinnvollste Konfiguration angegeben wird. Diese Konfigurationsdatei wird anhand der Angaben erstellt, die in dem Befehl `sphinx-quickstart` eingegeben wurden. Zum Beispiel stellt der Befehl die Frage, ob *autodoc* verwendet werden soll. Diese Frage ist mit *JA* zu beantworten, weil sonst die Dokumentation nicht automatisch generiert werden kann³¹.

→Prinz

Sobald dieser Befehl vollständig ausgeführt wurde, wird eine weitere Datei `index.rst` erstellt. Der Zweck dieses Dokuments ist im Grunde, dass ein Deckblatt und ein Inhaltsverzeichnis generiert werden³¹.

Um Inhalte hinzuzufügen, können einige Features von *reStructuredText* verwendet werden. Zum Beispiel kann als Richtlinie *toctree* verwendet werden. *toctree* kann verglichen werden mit *Markup*, jedoch ist *toctree* um einiges vielseitiger³¹.

Um Dokumente in die Dokumentation beziehungsweise Einträge in das Inhaltsverzeichnis hinzuzufügen müssen die jeweiligen Dateien in dem *toctree* angegeben werden. Ein *toctree* mit zwei Elementen sieht zum Beispiel wie folgt aus³¹:

→Prinz

```
.. toctree...
:maxdepth: 2

usage/installation
usage/quickstart
```

Um die ganze Dokumentation zu erstellen, muss der Befehl

⁴³Georg Brandl. „Sphinx Documentation“. In: (), S. 397.

```
sphinx-build -b html sourcedir builddir
```

ausgeführt werden. Der Parameter `-b` gibt an, in welchem Format die Dokumentation erstellt werden soll. In dem obigen Beispiel hat die Dokumentation den Dateitypen *HTML*. Das Argument `sourcedir` setzt den Ordner, wo die zu generierende Dokumentation enthalten ist und `builddir` gibt an, wo die generierte Dokumentation gespeichert werden soll.⁴⁴

Die Dokumentation kann jedoch auch mit Hilfe einer anderen Möglichkeit generiert werden. Das Skript `sphinx-quickstart` erzeugt zusätzlich noch eine *Makefile* und ein Dokument `make.bat`. Um das *Makefile* auszuführen wird folgende Anweisung benötigt³²:

```
make html
```

Der Parameter `html` gibt an, dass der Typ der Dokumentation eine *HTML*-Datei ist. Falls jedoch nur der Befehl `make` ohne weitere Parameter angegeben wird, wird eine Hilfe mit allen möglichen Zieldokumenten in dem gewünschten Ordner ausgegeben³².

3.11.2 Aufbau der Dokumentation

→Prinz

Das Grundgerüst der Dokumentation für das Projekt wurde mit Hilfe von Sphinx automatisch generiert. Weitere Informationen wie unter anderem Beispielaufrufe müssen per Hand geschrieben werden.

Zu Beginn der Dokumentation befindet sich ein Deckblatt mit allen Autoren. Anschließend bietet ein Inhaltsverzeichnis eine kleine Übersicht an. Nach diesem beginnt Kapitel 1 mit einer kurzen Anleitung, wie das Projekt installiert werden kann.

In Kapitel 2 der Dokumentation werden alle Befehle aufgelistet, wobei jeder Befehl über eine eigene Sektion verfügt. In jeder dieser Sektionen werden ebenfalls alle gültigen Parameter sowie mindestens ein Beispielaufruf angegeben.

Das Projekt verfügt über zwei unterschiedliche Hauptfunktionalitäten. Diese teilen sich in das Generieren eines *Entity Relationsip Diagrammes* und in das Erzeugen von *Data Manipulation Language*-Kommandos auf. In der Dokumentation werden zuerst die Befehle aufgezählt, die für das Erstellen eines *ERDs* verwendet werden können. Zu diesen Befehlen gehören:

→Prinz

- `erdgenerate`
- `open`
- `exit`
- `shell`
- `bye`
- `list`
- `erdfocus`

⁴⁴Georg Brandl. „Sphinx Documentation“. In: (), S. 397.

- `blockdiagram`

Nach den bereits aufgelisteten Befehlen für die Erstellung eines *ERDs* werden alle Befehle zur Generierung von *DML*-Kommandos aufgezählt:

→Prinz

- `dmlgenerate`
- `dmlform`
- `config`
- `basex`
- `oracle`
- `postgresql`
- `sqlite`
- `sqlserver`
- `mysql`

Die Sektion, die den Befehl `erdgenerate` beschreibt, sieht wie folgt aus:

2.3.1 erdgenerate

Generate an ERD from XERML Modell

```
ermkt erdgenerate [-h] [-i INPUTFILE] [-n NOTATION] [-t TYP] [-a]
                  [-c] [-g] [-p] [-d] [-v] [-l LOC] [-s] [--auto]
```

Named Arguments

<code>-i, --inputfile</code>	Inputfile
<code>-o, --output</code>	Outputfile
<code>-n, --notation</code>	Takes a value to define the notation
<code>-t, --typ</code>	Attributes with types are displayed in the ERD
<code>-a, --attr</code>	The ERD displays Attributes
<code>-c, --color</code>	The ERD is colored
<code>-g, --graphml</code>	The Output-Type is a GraphML File
<code>-p, --pic</code>	The Output-Type is a PIC File
<code>-d, --draw</code>	The Output-Type is a Libre Office Draw File
<code>-v, --viz</code>	The Output-Type is a Graphviz File
<code>-l, --loc</code>	Define the output language
<code>-s, --show</code>	Shows generated Diagram in Programm
<code>--auto</code>	ERD generated with default options

EXAMPLES

```
ermkt erdgenerate -i sis.xerml.xml -o sis.graphml -g
ermkt erdgenerate -o sis.xerml.xml -o sis.graphml -n crowfoot -g
```

Kapitel 4

Testmodelle

4.1 Überblick

Bei dem Projekt gab es 9 Testmodelle die wären:

- Fußball
- Kinokette
- Mondial
- Rettungsstelle
- Schulungsfirma
- Schulinformationssystem
- Tankstellenkette
- Weingut

Jedes dieser Testmodelle stellt eine Datenbank da. Alle 9 mussten in Form einer XERML-Datei erstellt werden um sie dann mit den vier verschiedenen Programmen zu verarbeiten. Darin sollten alle Variationen vorhanden sein z.B. Vererbungen, Abhängige-Typen usw.. Diese XERML-Dateien können mit einem Tool eines anderen Projekts automatisch generiert werden. Sie bestehen in der Regel aus drei verschiedenen Dateien.

- Grunddatei mit der Endung "xerml.xml", in dieser Datei steht der grobe Aufbau des Modells.
- Sprachdatei wo verschiedene Übersetzungen enthalten sind mit der Endung "xerml.lo.xml".
- Typdatei die die einzelnen Attribute einer Entity beschreibt mit der Endung "xerml.ty.xml".

4.2 Schulinformationssystem

Das Testmodell „Schulinformationssystem“ soll das Datenmodell für eine HTL abbilden. Dabei waren folgenden Punkte aus dem Skriptum Hillebrand, „Datenbanken und Informationssysteme“ zu berücksichtigen:

→ Passet

1. Die abteilungsweise Gliederung einer HTL ist wiederzugeben.
2. Jeder Lehrer ist einer Abteilung als Stammabteilung zugeordnet, kann aber auch in Klassen anderer Abteilungen unterrichten. Jede Abteilung wird von einem Lehrer als Abteilungsvorstand geleitet.
3. Für jede Ausbildungsform der Abteilungen ist im Lehrplan festgehalten, welche Gegenstände in welchen Jahrgängen, in welchem Ausmaß (Theorie- und Übungsstunden) unterrichtet werden müssen.
4. Die Klassen eines Schuljahres werden von den Lehrern in den einzelnen Gegenständen in einem bestimmten Stundenausmaß (Theorie- und Übungsstunden) unterrichtet.
5. Jeder Schüler wird mit einer Semester- und einer Jahresnote pro Klasse und Gegenstand beurteilt. In dem System sollen diese Informationen für mehrere Schuljahre festgehalten werden können.
6. Die Klassenvorstände der verschiedenen Klassen sollen feststellbar sein, ebenso von welchem Schüler welche Funktionen (Klassensprecher, Kassier, etc.) ausgeübt werden oder wurden.
7. Die Entlohnung der Lehrer erfolgt nicht nach gehaltenen Stunden, sondern nach gehaltenen Werteinheiten: jeder Gegenstand ist einer bestimmten Lehrverpflichtungsgruppe (LVG) (I bis VI) zugeordnet. Für jede LVG ist ein Faktor (1,167 bis 0,75) festgelegt, der zur Umrechnung von Stunden in Werteinheiten herangezogen wird.
8. Für jeden Schüler ist ein Erziehungsberichtiger verantwortlich (sofern der Schüler nicht eigenberechtigt ist). Wenn Geschwister die Schule besuchen, soll dies ebenfalls ermittelt werden können.

Diese Punkte ließen auf die folgenden Entitytypen schließen:

- | | |
|--|--|
| <ul style="list-style-type: none"> • Person • Lehrer • Schüler • Abteilung | <ul style="list-style-type: none"> • Gegenstand • Klasse • Lehrverpflichtungsgruppe • Lehrplan |
|--|--|

Um das Datenmodell vollständig abzubilden sind auch Beziehungen zwischen den Entiytypen nötig. Um zum Beispiel den 2. Punkt der Aufgabe zu realisieren, wurden folgende Beziehungstypen definiert:

Listing 4.1: XERML-Definition von Punkt 2 der Angabe

```

1 <!-- Punkt 2 -->
2 <rel to="gehört zu">
3   <part ref="Lehrer" min="1" max="1"/>
4   <part ref="Abteilung" min="1" max="n"/>
5 </rel>
6
7 <rel to="wird geleitet" from="leitet">
8   <part ref="Abteilung" min="1" max="1"/>
9   <part ref="Lehrer" min="0" max="1"/>
10 </rel>

```

Die komplette Umsetzung des Datenmodells befindet sich im Anhang.

4.3 Rettungsstelle

→Fischbacher

Die Rettungsstelle soll den Aufbau einer echten Rettungsstelle darstellen und dabei veranschaulichen welche Vorgänge bzw. Ressourcen miteinander in Beziehung stehen. Das Datenmodell der Rettungsstelle besteht aus sieben Entity-Typen:

- Einsatz oder auch mission
- Fahrt oder auch trip
- Person oder auch person
- Angestellter oder auch employee
- Patient oder auch patient
- Auto oder auch car
- Garage oder auch garage

Zwischen diesen Entity-Typen existieren sechs Relationen und jede dieser Relationen kann man mit je zwei Sätzen beschreiben.

- Ein Einsatz besteht aus keiner oder mehreren Fahrten.
Eine Fahrt ist Bestandteil genau eines Einsatzes.
- Ein Mitarbeiter nimmt an keiner oder mehreren Fahrten teil.
Bei einer Fahrt ist mindestens ein Mitarbeiter oder mehrere Mitarbeiter beteiligt.
- Ein Patient wird bei genau einem Einsatz gerettet.
Bei einer Fahrt wird mindestens ein Patient oder mehrere Patienten gerettet.
- Ein Auto wird bei mindestens einer Fahrt oder mehreren Fahrten verwendet.
Bei einer Fahrt wird genau ein Auto verwendet.
- Ein Auto hat mindestens eine Garage oder mehrere Garagen.
In einer Garage steht genau ein Auto.
- Patient ist eine Person mit einer SVN, KVA, Einsatzbeschreibung, Kennzeichen, Einsatznummer.
Angestellter ist eine Person mit einer Mittarbeiternummer, Rang, Schulung, Kennzeichen, Einsatznummer.

4.4 Fußball

→ Prinz

Für die Erstellung des Datenmodells *Fußball* wurden folgende Annahmen getroffen:

- Jede Mannschaft hat einen eindeutigen Namen, ein bestimmtes Gründungsjahr und ist an einer bestimmten Adresse beheimatet.
- Zu jeder Mannschaft gehören Fußballspieler.
- Ein Spieler kann durch die SVNr identifiziert werden. Weiters hat ein Spieler einen Namen, eine Wohnadresse, ein Geburtsdatum und eine Position, an der er spielt.
- Die Mannschaften beteiligen sich an Spielen.
- Die Spiele können durch die Adresse des Stadions, dem Tag und der Uhrzeit eindeutig festgelegt werden.
- Pro Spiel werden die beteiligten Mannschaften sowie der Schiedsrichter und das Ergebnis gespeichert.
- Falls das Spiel zu einem Turnier gehört, soll diese Information ebenfalls gespeichert werden.
- Die Anzahl der Tore, die in einem Spiel geschossen wurden, sollen gespeichert werden.
- Ein Schiedsrichter verfügt über die gleichen Daten wie ein Spieler außer dass er keine Spielposition hat. Dafür wird bei dem Schiedsrichter das Datum der Schiedsrichterprüfung und die Berechtigungsklasse gespeichert.
- Jedes Turnier hat eine eindeutige Nummer, einen Namen, ein Beginn- und Enddatum und die beteiligten Mannschaften gespeichert.

Anhand all dieser Annahmen entsteht folgendes Datenmodell:

→ Prinz

```
<erm version="0.2">

<!-- Front Matter -->

<title name="Fussball"/>
<title name="soccer" lang="en"/>

<!-- Entity-Types -->

<ent name="mannschaft">
    <attr name="name" prime="true"/>
    <attr name="gründungsjahr"/>
    <attr name="adresse"/>
</ent>

<ent name="person">
    <attr name="svnr" prime="true"/>
    <attr name="name"/>
    <attr name="wohnadresse"/>
    <attr name="geburtsdatum"/>
</ent>

<ent name="spieler">
    <attr name="spielposition"/>
</ent>
```

```

<ent name="spiel">
    <attr name="spielort" prime="true"/>
    <attr name="datum" prime="true"/>
    <attr name="mannschaft_heim"/>
    <attr name="mannschaft_ausw"/>
    <attr name="schiedsrichter"/>
    <attr name="ergebnis"/>
</ent>

<ent name="turnier">
    <attr name="nummer" prime="true"/>
    <attr name="name" prime="true"/>
    <attr name="beginndatum"/>
    <attr name="enddatum"/>
    <attr name="mannschaften"/>
</ent>

<ent name="schiedsrichter">
    <attr name="datum_prüfung"/>
    <attr name="berechtigungsklasse"/>
</ent>

<ent name="tore">
    <attr name="anzahl_tore"/>
</ent>

```

Die *Entities* stehen wie folgt in Beziehung miteinander:

→Prinz

```

<rel to="ist ein">
    <super ref="person" total="false" disjoint="true"/>
        <sub ref="spieler"/>
        <sub ref="schiedsrichter"/>
</rel>

<rel to="spielt bei">
    <part ref="spieler" min="1" max="1"/>
    <part ref="mannschaft" min="1" max="n"/>
</rel>

<rel to="spielt mit bei">
    <part ref="mannschaft" min="1" max="n"/>
    <part ref="spiel" min="1" max="n"/>
</rel>

<rel to="gehört zu">
    <part ref="spiel" min="0" max="1"/>
    <part ref="turnier" min="1" max="n"/>

```

```

</rel>

<rel to="pfeift bei">
    <part ref="schiedsrichter" min="1" max="1"/>
    <part ref="spiel" min="1" max="1"/>
</rel>

<rel to="hat geschossen">
    <part ref="tore" min="0" max="n"/>
    <part ref="spieler" min="0" max="n"/>
</rel>

<rel to="wurden geschossen">
    <part ref="tore" min="0" max="n"/>
    <part ref="spiel" min="0" max="n"/>
</rel>
</erm>

```

→ Prinz

4.5 Weingut

→ Homolka

Im Umfang der Diplomarbeit ist die Erstellung einer *XERML*-Testmodells von jedem Diplomanden beinhaltet, um die zu entwickelnde Software mit den Modellen testen zu können. Gefordert ist das Modell eines Weinguts, in dem verschiedene Erzeuger diverse Weine anbieten. Jeder Erzeuger liegt in einem Anbaugebiet und hat eine Lizenz, die es dem Erzeuger ermöglicht eine bestimmte Menge Wein zu erzeugen. Die Weine bestehen jeweils zu einem bestimmten Anteil aus verschiedenen Rebsorten.

4.5.1 Die Hauptdatei

→ Homolka

Bei der Modellierung der Hauptdatei *weingut.xerml.xml* kommt es zu mehreren Varianten der *Beziehungen* oder *Attribute*. Um das *XERML* so realistisch und einfach wie möglich zu halten, wurden folgende Annahmen getroffen:

- Der Name des Weins ist der *Primary-Key*, da es in der Realität nicht mehrere Weine mit dem selben Namen gibt.
- Der Name und die Adresse des Erzeugers bilden den *Primary-Key*, da es keine Erzeuger mit demselben Namen und derselben Adresse geben kann.
- Der Name und die Region des Anbaugebiets ist der *Primary-Key*, da es nicht mehrere Anbaugebiete mit demselben Namen in derselben Region geben kann.
- Der Name einer Rebsorte ist eindeutig und daher der alleinige *Primary-Key*.
- Die Lizenznummer einer Lizenz ist eindeutig und daher der alleinige *Primary-Key*.
- Ein bestimmter Erzeuger kann mehrere Weine erzeugen und ein bestimmter Wein kann von mehreren Erzeugern erzeugt werden. Ein Erzeuger muss allerdings mindestens einen Wein erzeugen um als Erzeuger zu gelten. Ein bestimmter Wein muss nicht zwingend hergestellt werden.
- Ein bestimmter Wein muss mindestens eine Rebsorte enthalten. Eine Rebsorte muss nicht zwingend in einem Wein enthalten sein. Jede Rebsorte ist zu einem gewissen Anteil in % in einem Wein enthalten.

- Ein bestimmter Erzeuger besitzt genau eine Lizenz. Eine Lizenz kann allerdings auch von mehreren Erzeugern besessen werden. Sie muss aber mindestens von einem besessen werden.
- Ein bestimmter Erzeuger muss in mindestens einem Anbaugebiet liegen. In einem bestimmten Anbaugebiet muss nicht zwingend ein Erzeuger anbauen. Es können allerdings mehrere darin anbauen.

→Homolka

Aufgrund dieser Annahmen ergibt sich folgendes *XERML*:

```
<erm version="0.2">

<title name="Weingut"/>

<ent name="wein">
  <attr name="name" prime="true"/>
  <attr name="farbe"/>
  <attr name="jahrgang"/>
  <attr name="restsüße"/>
  <attr name="erzeuger"/>
</ent>

<ent name="erzeuger">
  <attr name="name" prime="true"/>
  <attr name="adresse" prime="true"/>
  <attr name="lizenz"/>
  <attr name="menge"/>
  <attr name="anbaugebiet"/>
</ent>

<ent name="anbaugebiet">
  <attr name="name" prime="true"/>
  <attr name="region" prime="true"/>
  <attr name="land"/>
</ent>

<ent name="rebsorte">
  <attr name="name" prime="true"/>
  <attr name="farbe"/>
</ent>

<ent name="lizenz">
  <attr name="lizenznummer" prime="true"/>
  <attr name="menge"/>
</ent>

<rel to="erzeugt">
  <part ref="erzeuger" min="1" max="n"/>
  <part ref="wein" min="0" max="n"/>
</rel>
```

```

<rel to="beinhaltet">
    <part ref="wein" min="1" max="n"/>
    <part ref="rebsorte" min="0" max="n"/>
    <attr name="anteil"/>
</rel>

<rel to="besitzt">
    <part ref="erzeuger" min="1" max="1"/>
    <part ref="lizenz" min="1" max="n"/>
</rel>

<rel to="liegt in einem">
    <part ref="erzeuger" min="1" max="n"/>
    <part ref="anbaugebiet" min="0" max="n"/>
</rel>

</erm>

```

4.5.2 Die Sprachdatei

Die Sprachdatei *weingut.xerml.lo.xml* beinhaltet eine englische Übersetzung der in der deutschen Sprache geschriebenen Hauptdatei.

→Homolka

4.5.3 Die Typdatei

Die Typdatei *weingut.xerml.ty.xml* beinhaltet die Datentypen aller *Attribute*. Die *Attribute* Jahrgang, Lizenznummer und Lizenz haben den Datentyp **Integer**. Die *Attribute* Restsüsse, Menge und Anteil haben den Datentypen **float**. Die anderen *Attribute* besitzt den Datentyp **char**.

→Homolka

Kapitel 5

Ergebnis

5.1 Diagramm mit GraphML

5.1.1 Aufbau der GraphML-Datei

→ Passet

Die Elemente innerhalb der GraphML-Datei sind in der selben Reihenfolge wie sie in der XERML-Datei des Benutzers stehen. Bei unseren Testmodellen sind zuerst die Entitytypen und hinterher die Beziehungstypen gelistet. Aus diesem Grund sind in der GraphML-Datei zuerst die Entitytypen und Beziehungstypen als *node*-Elemente. Zwischen den Elementen die die Beziehungstypen repräsentieren sind die *edge*-Elemente enthalten um die Knoten miteinander zu verbinden.

5.1.2 Beispiel eines ER-Diagramms in yEd

In der nachfolgenden Abbildung 5.1 sieht man ein Beispiel für ein ER-Diagramm in yEd. Das Diagramm wurde vollständig durch das Tool erzeugt. Einzig die endgültige Anordnung der Elemente wurde durch die Layout-Algorithmen von yEd übernommen.

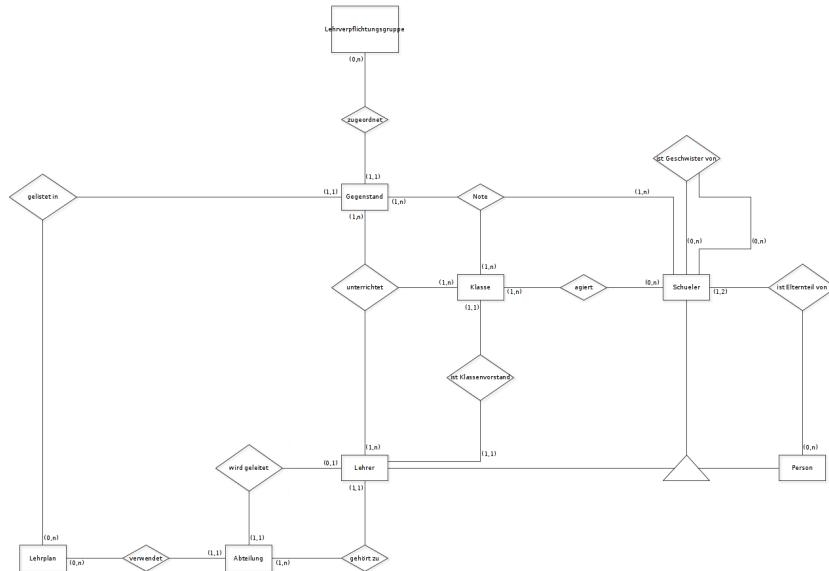


Abbildung 5.1: ER-Diagramm des Datenmodells Schulinformationssystem

5.2 Ergebnis der Darstellung mit Graphviz

→Fischbacher

Die Erstellung von Entity-Relationship Diagramme mittels Graphviz hat sich am besten mit den Engines sfdp und neato realisieren lassen. Bei Folgenden Abbildungen wurden Absichtlich die Attribute nicht mit generiert, da Graphviz dazu neigt die Attribute übereinander zu stapeln wodurch der Graph unleserlich wird.

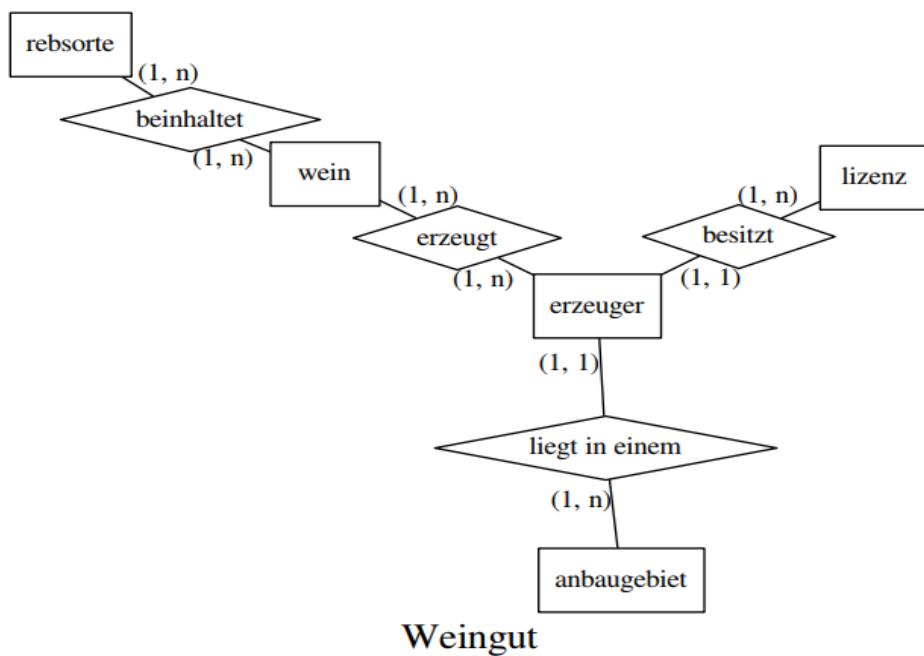


Abbildung 5.2: Weingut Entity-Relationship Diagramm erstellt mittels Graphviz

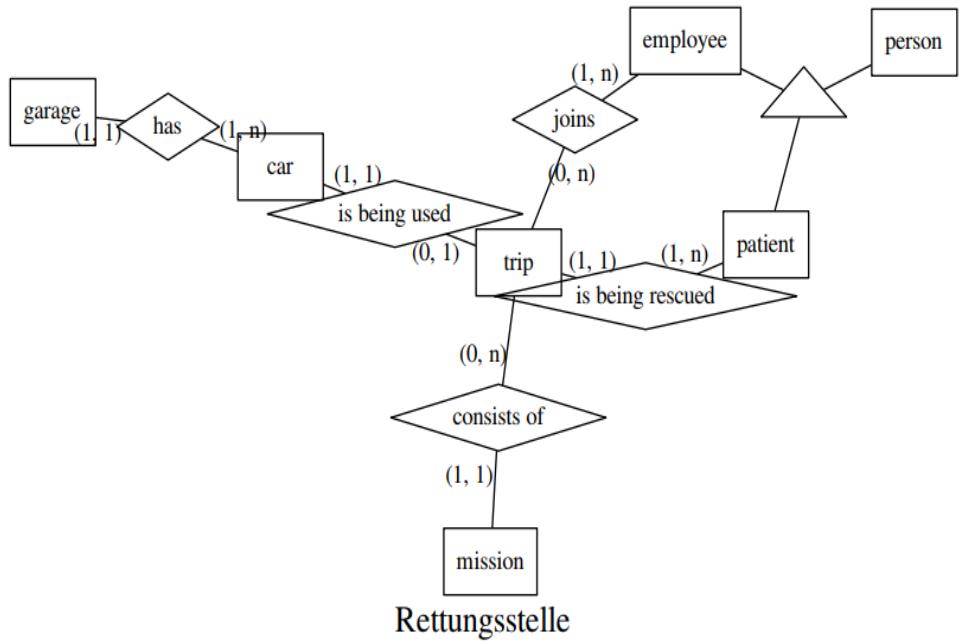


Abbildung 5.3: Rettungsstelle Entity-Relationship Diagramm erstellt mittels Graphviz

→Fischbacher

An den Abbildung 5.2 und Abbildung 5.3 sieht man das sich Graphviz bei kleineren Graphen durchaus geeignet hat.

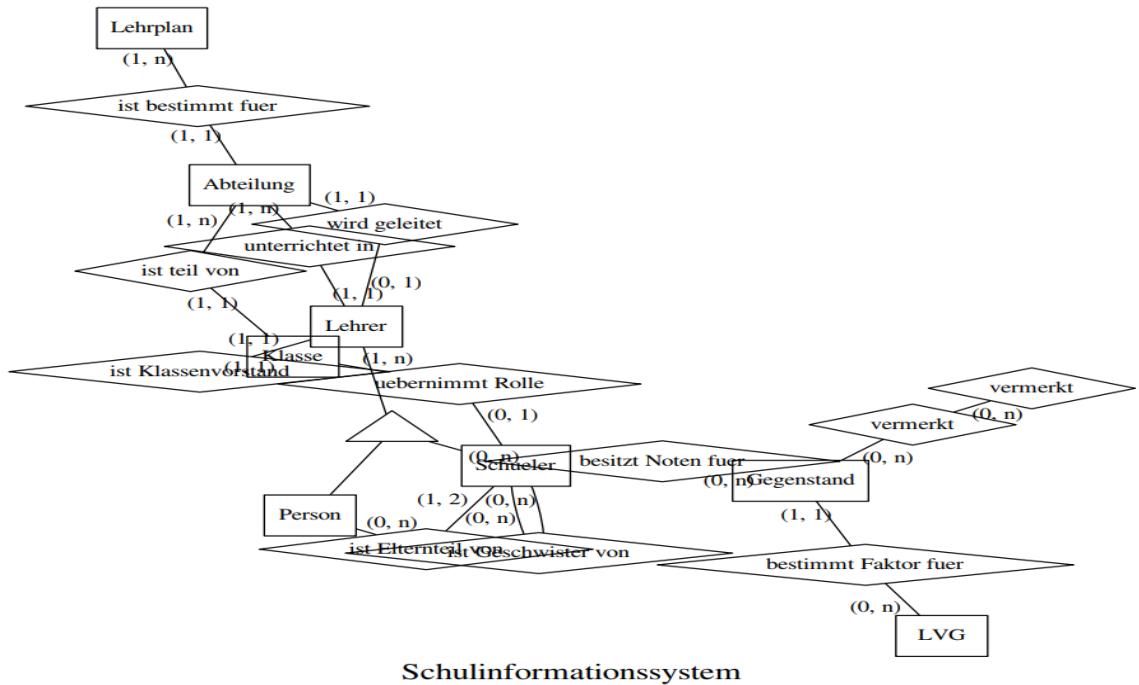


Abbildung 5.4: Schulinformationssystem Entity-Relationship Diagramm erstellt mittels Graphviz

Jedoch sieht man an den Abbildung 5.4 das ab einer gewissen Anzahl an Elementen Graphviz nicht mehr eignet, da es die Elemente wegen Platz mängels, immer enger aneinander positioniert, wodurch der Graph unleserlich wird.

5.3 Diagramm mit PIC-Code

5.3.1 Aufbau des PIC-Codes

→ Prinz

Der Inhalt des generierten *PIC*-Codes verfügt über eine gewisse Struktur. Zu Beginn des Dokuments werden erstmals alle *Entities* und *Beziehungstypen* definiert und gezeichnet. Diese sind noch mit keinem anderen Element verbunden. Sobald alle *Entity*-Typen und *Beziehungstypen* erstellt wurden, werden die Verbindungen zwischen den einzelnen Objekten gezeichnet. Der letzte Teil des Codes generiert die *Attribute* und verbindet diese mit den jeweiligen *Entity*-Typen.

5.3.2 Problemfälle bei dem erstellten ERD

Das erzeugte *Entity Relationship Diagramm* verfügt über gewisse Mängel. Diese entstehen zum Beispiel, indem sich gewisse Linien überschneiden. In diesem Fall besteht die Möglichkeit, dass der Inhalt der `min`- und `max`-Werte schwer lesbar wird. Abbildung 5.5 veranschaulicht dieses Problem.



Abbildung 5.5: min- und max-Werte schwer lesbar

→ Prinz

Des weiteren kann es vorkommen, dass *Attribute* sich überschneiden. Diese Überschneidungen können dazu führen, dass der Name eines *Attributes* teilweise nicht mehr lesbar ist. Dieses Problem wird in Abbildung 5.6 gezeigt.

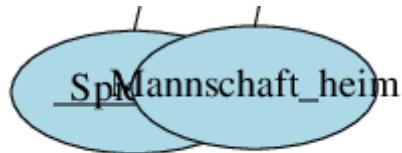


Abbildung 5.6: Name der Attribute schlecht lesbar wegen Überschneidung

Attribute können am Rand der Grafik gezeichnet werden, sodass diese nur teilweise in dem angezeigtem Bereich liegen. Diesen Problemfall veranschaulicht Abbildung 5.7.

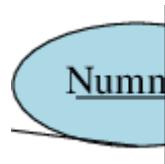


Abbildung 5.7: Attribut liegt nur teilweise im sichtbaren Bereich

→ Prinz

5.3.3 Ist PIC-Code für das Projekt geeignet?

5.3.3.1 Generierung der Objekte des ERDs

PIC verfügt über viele einfache Mittel, um ein *Entity Relationship Diagram* zu generieren. Auf Grund der bereits vordefinierten Objekte lässt sich das Zeichnen des Grundgerüstes des *ERDs* relativ leicht gestalten. Bei der Implementierung einer Raute, die PIC von Beginn an nicht bekannt ist, besteht ebenfalls kein großer Aufwand, wie in Abbildung 3.3 bereits erwähnt wurde.

5.3.3.2 Zeichnen der Beziehungen

Weiters vereinfacht das Arbeiten mit Labels die Implementierung der Verbindungen zwischen den Elementen. Zuerst werden alle Boxen, Ellipsen und Rauten gezeichnet und sobald alle Elemente erstmals vorhanden sind, werden die Verbindungen zwischen den Objekten generiert. Auf Grund der Label ist dies kein Problem mehr, da auf die unterschiedlichen Elemente leicht zugegriffen werden kann (siehe Abbildung 3.9).

5.3.3.3 Skalierung der Objekte

Die Elemente können in *PIC* leicht über das Argument *scale* skaliert werden. Dies ist hilfreich, wenn die Größe des zu zeichnenden Diagrammes bereits bekannt ist. Jedoch kann nicht automatisiert skaliert werden. Das heißt, dass die Größe des *ERDs* zuerst festgelegt werden muss. Da die Koordinaten im vorhinein schon erzeugt werden, ist dies kein Problem, jedoch bringt diese Methode einen größeren Implementierungsaufwand mit sich.

5.3.3.4 Zeichnen von Rauten/Dreiecken in Farbe

Im Gegensatz zu den Boxen und Ellipsen kann das farbige Zeichnen von Rauten und Dreiecken nicht umgesetzt werden, weil diese Elemente im Endeffekt nur Linien sind und auf die Position innerhalb der Linien nicht zugegriffen werden kann. Die unsichtbare Box, in der die Raute beziehungsweise das Dreieck gezeichnet werden, kann zwar farbig erstellt werden, jedoch wird dann nicht nur der Bereich innerhalb der Raute oder des Dreiecks gefärbt, sondern die ganze Box.

5.3.3.5 Vergleich mit anderen Varianten

→Prinz

Im Vergleich zu den anderen Varianten *Graphviz*, *Graphml* und *Libre Office Draw* verfügt die Sprache *PIC* über weniger Möglichkeiten für die Implementierung und Generierung eines *Entity Relationship Diagrammes*. Beispielsweise kann das erzeugte Diagramm im Nachhinein nicht mehr verändert werden, da es in einer *PDF*-Datei oder in ein Bild gespeichert wird. Bei den Varianten *GraphML* und *Libre Office Draw* können einzelne Elemente nachträglich noch per Hand verschoben werden. Daher ist *PIC* im Vergleich zu den anderen Implementierungsvarianten am Wenigsten geeignet für dieses Projekt, jedoch gelingt es trotzdem, ein übersichtliches *ERD* zu erzeugen.

5.3.3.6 Fertiges ERD

In Abbildung 5.8 wird das mittels *PIC*-Code generierte *Entity Relationship Diagramm* von dem Datenmodell *Fußball* mit *Attributten* und Farbe dargestellt.

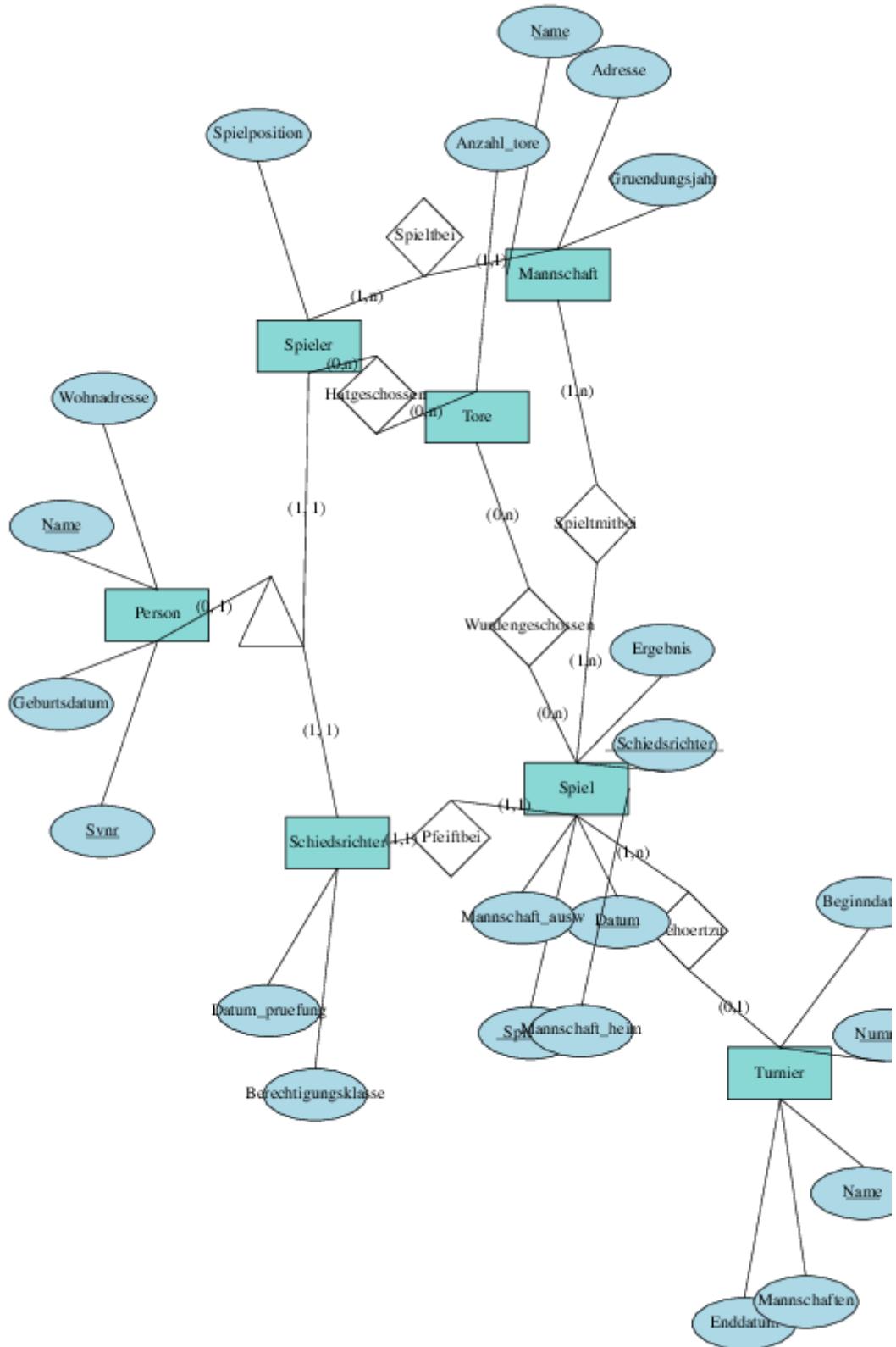


Abbildung 5.8: Mittels PIC erstelltes ERD für das Datenmodell Fußball

5.4 Diagramm mit LibreOffice Draw

Ein *ERD* wird in *LibreOffice Draw* korrekt und vollständig erzeugt. Alle Sonderfälle wie unter anderem *Abhängige Entity-Typen*, *Super-Sub-Beziehungstypen* und *mehrwertige Beziehungen* werden bei der Generierung aller Datenmodelle (siehe Kapitel 4) in der erwünschten Form dargestellt.

→Homolka

5.4.1 Probleme bei der Darstellung

Bei der Erstellung eines *ERDs* sind folgende Probleme aufgetreten:

- In *LibreOffice Draw* gibt es keine vorhandene Möglichkeit die *Krähenfußnotation* zu implementieren, da keine vordefinierte Formen für die Elemente dieser Notation vorhanden sind. Ein manuelles Erzeugen der Elemente ist komplex und aufwendig.
- Bei dem Layout, dass mittels *graphviz-layout* generiert wird, kann es zu, je nach Datenmodell, ein oder mehreren Überschneidungen kommen. Ein solcher Vorfall beeinträchtigt die Lesbarkeit des *ERDs*. Die Abbildung 5.9 zeigt eine Überschneidung der Konnektoren bei dem Datenmodell *SIS*.

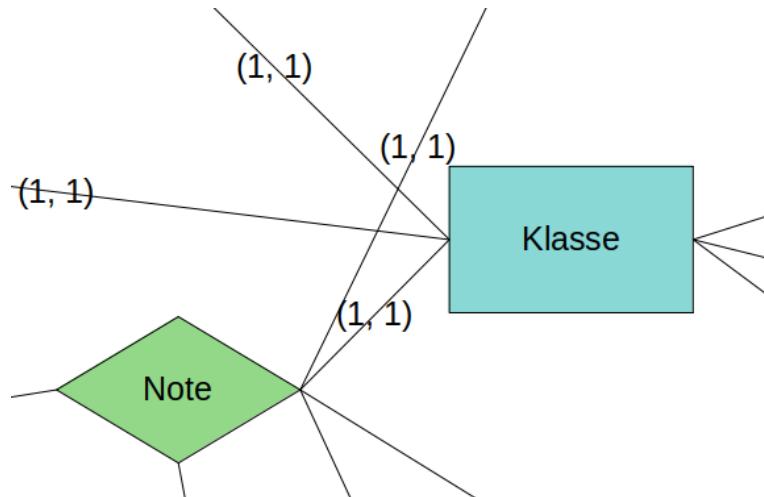


Abbildung 5.9: Überschneidung der Konnektoren bei dem Datenmodell *SIS*

5.4.2 Beispiel eines ER-Diagramms in LibreOffice Draw

In der Abbildung 5.10 wird das *ERD* des Datenmodells *Weingut* gezeigt. Dieses wurde vollständig durch das Tool erzeugt und wurde händisch **nicht** nachbearbeitet.

→Homolka

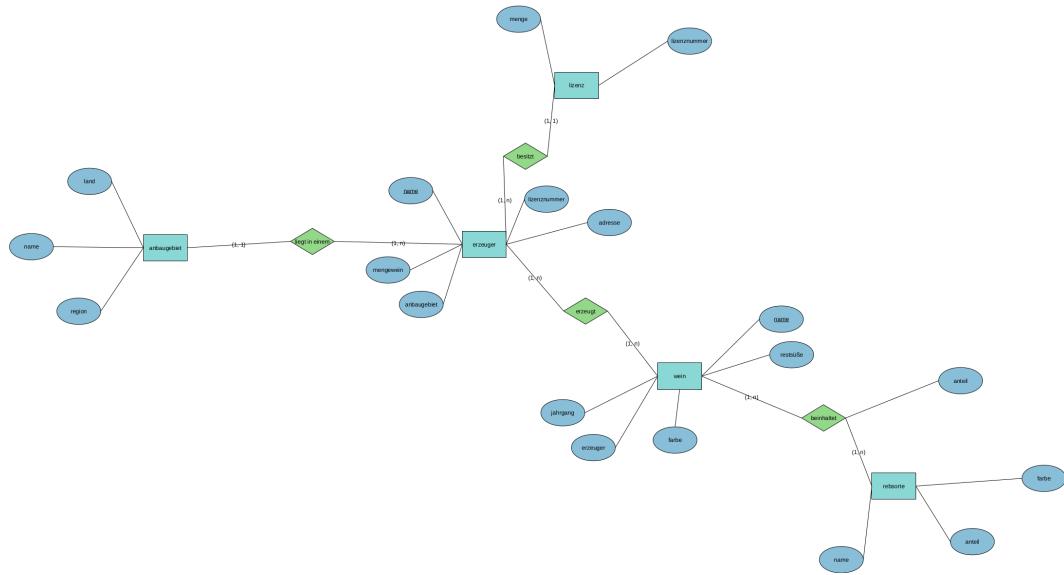


Abbildung 5.10: ER-Diagramm des Datenmodells *Weingut*

Teil V

Schlussfolgerung

Kapitel 6

Schluss

6.1 Ziele

Das Ziel der Diplomarbeit war es ein Toolkit zu erstellen, das dem Benutzer das händische bzw. digitale Zeichnen von Entity-Relationship Diagrammen abnimmt. Diese sollten außerdem ein sehr gutes Layout besitzen damit nur noch kleine händische Korrekturen vorgenommen werden müssen.

6.1.1 Welche Ziele wurden erreicht?

Das Toolkit kann für jede der vereinbarten Ausgabeformate ein Entity-Relationship Diagramm zeichnen und das in wenigen Sekunden. Beim Erstellen eines Layouts gab es anfänglich Probleme. Diese wurden jedoch für alle Ausgabeformate überwunden.

6.2 Zukunft des Tools

Die automatische Generierung und auch die Steuerung der Darstellung durch die Eingabe von Parametern ist fertiggestellt weshalb das Tool als fertig implementiert anzusehen ist. Abhängig von den Programmen yEd und Libre Office Draw sind eventuelle Wartungsarbeiten einzuplanen, falls diese Applikationen grundlegende Strukturen ihrer Dateien ändern.

6.3 Verwendung

6.3.1 Hilfe

Um die Hilfe aufzurufen kann man einen der beiden folgenden Befehle verwenden:

- “ermtk –help,,
- “ermtk -h,,

```
fisch@fischbacher:~/Documents/Dipl/p185b01/datenmodelle/weingut/xml$ ermtk -h
usage: ermtk [-h] [-v]
              {erdgenerate,open,close,exit,shell,LOD,bye,list,erdfocus,blockdiagram,dllgenerate,dmlgenerate,dmlform,config}
              ...
Runs Help Command

optional arguments:
  -h, --help            show this help message and exit
  -v, --version          show program's version number and exit

sub-commands:
  {erdgenerate,open,close,exit,shell,LOD,bye,list,erdfocus,blockdiagram,dllgenerate,dmlgenerate,dmlform,config}
    sub-command help
      erdgenerate        Generate an ERD from XERML Modell
      open               Opens a XERML-File to use it in the Shell
      close              Closes a XERML-File if it is not needed anymore
      exit               Exit the shell
      shell              Enters the shell
      LOD                LibreOffice Draw
      bye                Exit the shell
      list               Generate an ERD from XERML Modell
      erdfocus           Generate an ERD focusing on specified Entities
      blockdiagram       Generate an blockdiagram of the relational model
      dllgenerate        Convert an XERML Modell into DDL-Commands
      dmlgenerate        Generate example data in form of DML-Commands
      dmlform            Generate an entry form for typ in example data
      config             Configure database connection attributes
```

Abbildung 6.1: Ausgabe vom help Befehl

6.3.2 Programm Aufruf

Das die ERMTK-Repl Shell gestartet wird muss man den Befehl “ermtk shell,, eingeben. Man merkt das man sich in der ERMTK-Repl Shell befindet, an dem Prompt “ermtk>,,

```
fisch@fischbacher:~/Documents/Dipl/p185b01/datenmodelle/weingut/xml$ ermtk shell
ermtk> open weingut.xerml.xml
-> XERML-File: weingut.xerml.xml sucessfully parsed
```

Abbildung 6.2: Aufruf der ERMTK-Repl Shell und öffnen einer Datei

Um dann eine Datei zu öffnen muss man den Befehl “open <inputfile>,, wie in Abbildung 6.2 dargestellt, verwenden.

6.3.3 “erdgenerate,,

```
fischbacher@fischbacher:~/Documents/Dipl/p185b01/datenmodelle/sf/xml$ ermtk shell
ermtk> open sf.xerml.xml
-> XERML-File: sf.xerml.xml sucessfully parsed
ermtk (sf.xerml.xml)> erdgenerate -h
usage: erdgenerate [-h] [-i INPUTFILE] [-o OUTPUT] [-n NOTATION] [-t TYP] [-a]
                   [-c] [-g] [-p] [-d] [-v] [-l LOC] [-s] [--auto]

Generates ERD Diagram

optional arguments:
  -h, --help            show this help message and exit
  -i INPUTFILE, --inputfile INPUTFILE
                        Inputfile
  -o OUTPUT, --output OUTPUT
                        Outputfile
  -n NOTATION, --notation NOTATION
                        Takes a value to define the notation Example:
                        --notation crowfoot
  -t TYP, --typ TYP    Attributes with types are displayed in the ERD
  -a, --attr            The ERD displays Attributes
  -c, --color           The ERD is colored
  -g, --graphml         The Output-Type is a GraphML File
  -p, --pic             The Output-Type is a Pic-File
  -d, --draw            The Output-Type is a LibreOffice Draw File
  -v, --viz             The Output-Type is a Graphviz File
  -l LOC, --loc LOC    Define the output language
  -s, --show            Shows generated Diagramm in Programm
  --auto                ERD generated with default options
```

Abbildung 6.3: Ausgabe der hilfe zum Befehl erdgenerate

Nachdem man eine Datei geöffnet hat kann man mittels “erdgenerate,, sich ein Entity-Relationship Diagramm generieren lassen, dabei hat man die Auswahl zwischen vier verschiedenen Varianten:

- -g, -graphml um den Graphen mittels Graphml zu generieren.
- -p, -pic um den Graphen mittels PIC-Code zu generieren.
- -d, -draw um den Graphen mittels LibreOffice Draw zu generieren.
- -v, -viz um den Graphen mittels Graphviz zu generieren.

Außerdem hat man die Möglichkeit mit dem Parameter “-c,, oder “-color,, sich den den Graphen farbig zu generieren lassen und mit dem Parameter “-a,, oder “-attr,, ob man Attribute auch generiert haben will. Die ERMTK-Repl Shell kann man jederzeit mit den Befehlen “exit,, oder “close,, verlassen.

→Fischbacher

Kapitel 7

Technische Ergänzungen

7.1 Testmodelle

7.1.1 Schulinformationssystem

7.1.1.0.1 Pfad: `../datenmodelle/sis/xml`
`sis.xerml.xml` Grunddatei
`sis.xerml.lo.xml` Sprachdatei
`sis.xerml.ty.xml` Typdatei

7.1.2 Rettungsstelle

7.1.2.0.1 Pfad: `../datenmodelle/rettungsstelle/xml`
`rettungsstelle.xerml.xml` Grunddatei
`rettungsstelle.xerml.lo.xml` Sprachdatei
`rettungsstelle.xerml.ty.xml` Typdatei

7.1.3 Fußball

7.1.3.0.1 Pfad: `../datenmodelle/fussball/xml`
`fussball.xerml.xml` Grunddatei
`fussball.xerml.lo.xml` Sprachdatei
`fussball.xerml.ty.xml` Typdatei

7.1.4 Weingut

7.1.4.0.1 Pfad: `../datenmodelle/weingut/xml`
`weingut.xerml.xml` Grunddatei
`weingut.xerml.lo.xml` Sprachdatei
`weingut.xerml.ty.xml` Typdatei

7.2 Python-Code

7.2.1 GraphML

7.2.1.0.1 Pfad: `../Projekt/`

`Graphml_converter.py` . Programmcode für die Generierung von einer GraphML-Datei

Autoren Index

- Group, GraphML Working, 42
Hillebrand, Kurt, 7, 93
Hills, Ted, 8, 9
Kudraß, Thomas, 6, 7
North, Stephen C, 58
Prabhu, C.S.R, 7
yWorks, 39, 41

Literatur Index

- IntegratedDevelopmentEnvironments - Python Wiki*, 12
PyCharm: the Python IDE for Professional Developers by JetBrains, 12
Datenbanken und Informationssysteme, 4, 5, 7, 93
Drawing graphs with NEATO, 58
Features - PyCharm, 12, 13
General Python FAQ — Python 3.7.3rc1 documentation, 11
Git, 87
graphics - How to programmatically draw an organization chart?, 57
Graphviz - Graph Visualization Software, 50
graphviz [dot and/or neato], 61
History and License — Python 3.7.3rc1 documentation, 11
NoSQL and SQL Data Modeling, 8, 9
Object - Oriented Database Systems : Approaches and Architectures, 7
Output Formats, 54
performance - Graphviz sfdp inferior in ubuntu comparing to mac?, 58
Previous Releases - PyCharm, 12
Taschenbuch Datenbanken, 3, 6, 7, 14
The GraphML File Format, 42
The Trac Project, 88
yEd Graph Editor Manual, 38, 39, 41

Literatur

- ausarbeitung1.pdf*. URL: <http://wwwgis.informatik.uni-kl.de/archiv/wwwdvs.informatik.uni-kl.de/courses/seminar/WS0203/ausarbeitung1.pdf> (besucht am 04.04.2019).
- Brandes, Ulrik, Markus Eiglsperger und Jürgen Lerner. *GraphML Primer*. URL: <http://graphml.graphdrawing.org/primer/graphml-primer.html>.
- Brandl, Georg. „Sphinx Documentation“. In: (), S. 397.
- Davison, Dr. Andrew. *Java LibreOffice Programming*. Deutsch. URL: <https://de.wikipedia.org/wiki/LibreOffice>.
- Features - PyCharm*. JetBrains. URL: <https://www.jetbrains.com/pycharm/features/> (besucht am 19.03.2019).
- General Python FAQ — Python 3.7.3rc1 documentation*. URL: <https://docs.python.org/3/faq/general.html#why-is-it-called-python> (besucht am 14.03.2019).
- Git*. In: *Wikipedia*. Page Version ID: 185356278. 3. Feb. 2019. URL: <https://de.wikipedia.org/w/index.php?title=Git&oldid=185356278> (besucht am 19.03.2019).
- graphics - How to programmatically draw an organization chart?* Stack Overflow. URL: <https://stackoverflow.com/questions/8983834/how-to-programmatically-draw-an-organization-chart> (besucht am 19.03.2019).
- Graphviz - Graph Visualization Software*. URL: <https://www.graphviz.org/> (besucht am 04.04.2019).
- Graphviz - Graph Visualization Software*. URL: <https://www.graphviz.org/> (besucht am 19.03.2019).
- graphviz [dot and/or neato]*. URL: <http://www.adp-gmbh.ch/misc/tools/graphviz/index.html> (besucht am 03.04.2019).
- Group, GraphML Working. *The GraphML File Format*. 2007. URL: <http://graphml.graphdrawing.org>.
- Hillebrand, Kurt. „Datenbanken und Informationssysteme“. Skriptum das im DBI-Unterricht an der HTBLuVA Wiener Neustadt verwendet wird.
- Hills, Ted. *NoSQL and SQL Data Modeling*. Englisch. 1. Aufl. Basking Ridge, NJ 07920 USA: Technics Publications, 2016. URL: <https://technicspub.com/nosql-modeling/>.
- History and License — Python 3.7.3rc1 documentation*. URL: <https://docs.python.org/3/license.html> (besucht am 14.03.2019).
- IntegratedDevelopmentEnvironments - Python Wiki*. URL: <https://wiki.python.org/moin/IntegratedDevelopmentEnvironments> (besucht am 19.03.2019).
- Kudraß, Thomas. *Taschenbuch Datenbanken*. Deutsch. 2. Aufl. München: Carl Hanser Verlag, 2015. URL: <https://www.hanser-fachbuch.de/buch/Taschenbuch+Datenbanken/9783446435087>.
- LibreOffice*. Deutsch. URL: <https://de.libreoffice.org/>.
- LibreOffice API*. Deutsch. URL: <https://api.libreoffice.org/>.
- LibreOffice Wikipedia*. Englisch. URL: <https://fivedots.coe.psu.ac.th/~ad/jlop/>.

- Making Pictures With GNU PIC.* URL: <https://www.complang.tuwien.ac.at/doc/groff/html/pic.html> (besucht am 04.04.2019).
- North, Stephen C. „Drawing graphs with NEATO“. In: (2004), S. 11.
- OpenOffice Developer's Guide.* Englisch. URL: https://wiki.openoffice.org/w/images/d/d9/DevelopersGuide_OOo3.1.0.pdf.
- Output Formats.* URL: https://graphviz.gitlab.io/_pages/doc/info/output.html#d:dot (besucht am 03.04.2019).
- performance - Graphviz sfdp inferior in ubuntu comparing to mac?* Stack Overflow. URL: <https://stackoverflow.com/questions/9952097/graphviz-sfdp-inferior-in-ubuntu-comparing-to-mac> (besucht am 19.03.2019).
- Prabhu, C.S.R. „Object - Oriented Database Systems : Approaches and Architectures“. Englisch. In: 3. Aufl. PHI Learning Pvt. Ltd., 2005. Kap. 2.2.4. URL: https://books.google.at/books?id=DBkM4XjKKw4C&hl=de&source=gbs_navlinks_s.
- Previous Releases - PyCharm.* JetBrains. URL: <https://www.jetbrains.com/pycharm/download/previous.html> (besucht am 19.03.2019).
- PyCharm: the Python IDE for Professional Developers by JetBrains.* JetBrains. URL: <https://www.jetbrains.com/pycharm/> (besucht am 19.03.2019).
- The Trac Project.* URL: <https://trac.edgewall.org/> (besucht am 19.03.2019).
- yWorks. *yEd Graph Editor Manual.* Englisch. URL: <https://yed.yworks.com/support/manual/index.html>.