universität
wien

MASTERARBEIT

Titel der Masterarbeit

"Social Network for APIs"

verfasst von

Patrik Burzynski Bakk.rer.soc.oec.

angestrebter akademischer Grad

Diplom-Ingenieur (Dipl.-Ing.)

Wien, 2013

Studienkennzahl lt. Studienblatt:	A 066 926
Studienrichtung lt. Studienblatt:	Masterstudium Wirtschaftsinformatik (UG2002)
Betreuer:	o. Univ.-Prof. Dr. Prof. h. c. Dimitris Karagiannis

Schriftliche Versicherung

Ich habe mich bemüht, sämtliche Inhaber der Bildrechte ausfindig zu machen und ihre Zustimmung zur Verwendung der Bilder in dieser Arbeit eingeholt. Sollte dennoch eine Urheberrechtsverletzung bekannt werden, ersuche ich um Meldung bei mir.

Ich versichere, dass die Arbeit von mir selbständig verfasst wurde und dass ich keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Weiterhin wurde diese Arbeit keiner anderen Prüfungsbehörde übergeben.

Wien, den 3. Dezember 2013

Acknowledgement

I want to thank everybody who has supported me in finishing this master thesis, either through motivation, pointing out new and interesting developments, feedback or other types of support.

I also want to thank the reader for showing interest and (hopefully) taking time to read this master thesis.

Table of Contents

List of Abbreviations

API	Application Programming Interface
ANTLR	ANother Tool for Language Recognition
BOC	Business Objectives Consulting
CASE	Computer-aided Software Engineering
HRM	Hidden Relational Model
IM	Instant Messaging/Messenger
IHRM	Infinite Hidden Relational Model
JSON	JavaScript Object Notation
JUNG	Java Universal Network/Graph Framework
LDAP	Lightweight Directory Access Protocol
OMI	Open Model Initiative
OMR	Open Model Repository
RDF	Resource Description Framework
SN	Social Network
SUS	System Under Study
UML	Unified Modeling Language
URI	Uniform Resource Identifier
UUID	Universally Unique Identifier
WWW	World Wide Web
XML	Extensible Markup Language
XPIDL	Cross Platform Interface Description Language

List of Tables

List of Figures

1 Introduction

Millions of users participate, communicate and share knowledge every day on one or several social network platforms. In this master thesis a concept for the creation of "Social API Networks" based on social networks and its prototypical implementation will be presented. The basic idea in this work is the assumption that whereas a person or organisation is in the centre of interest in a social network, in the API network it can be replaced by an API, with some adaptations. The aim of the "Social API Network" (or API network for short) is to support both developers creating APIs and the ones using the APIs.

As an example social networks often contain friendship or acquaintance relations. These can be translated into similarity relationships between APIs, based on the thought that friends usually have some similarities, either in their interests, their view of life or other aspects. Using such a structure and a cluster detection algorithm APIs could be grouped together creating "communities" of similar APIs. A developer that uses APIs could then use those sets of similar APIs to find alternative APIs which might better fit their needs. Another use of the API network would be to identify APIs that are often used and therefore should only be changed with great care. Some further applications envisioned for the API network are described in section 3.1.

For this selected social networks like Facebook and Twitter have been consulted and used as an input to derive an appropriate structure for the domain of APIs. Furthermore literature and research findings about social network processing and analysis have been applied to utilize the information from the API network and to identify possible cases of interest for users. This is generally depicted in figure 1.1.

The API network was then realised and made available to the Open Model Initiative[1] (OMI). The prevalent meta-modelling approach of the OMI was applied on the knowledge gained from social networks to create a feasible concept. Also interfaces to the available Open Model Repository (OMR)[2] have been be created to support the persistence of an API network as well as the proposed mechanisms and algorithms were implemented. Using this implementation

[1] Accessible at `http://www.openmodel.at`, accessed 20.09.2013
[2] based on Schremser [2011], available at `http://omi-repo.dke.univie.ac.at:18080`, accessed 19.10.2013

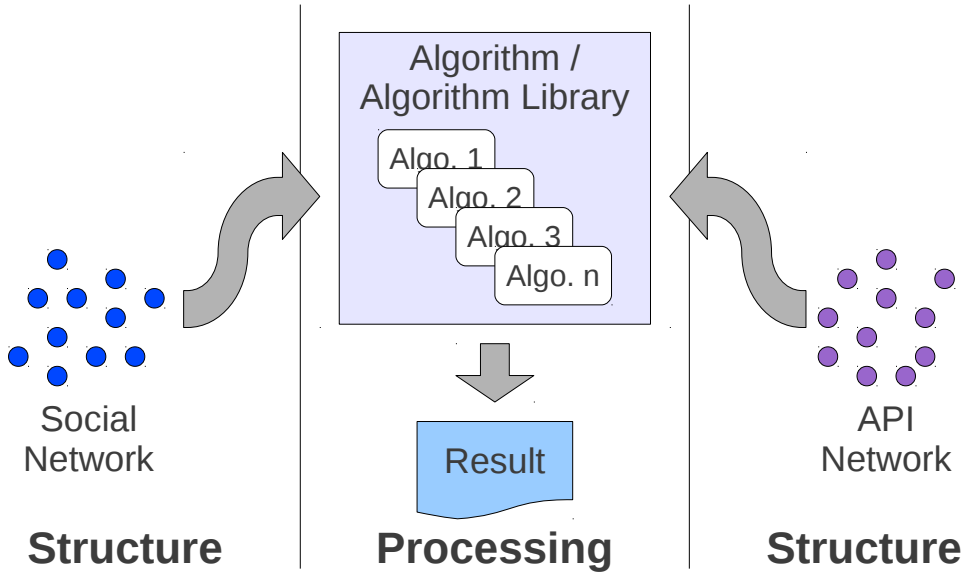

Figure 1.1: Idea of an approach for an API network

the APIs of the ADOxx toolkit[3] were used as test data in an experiment to determine if using the insights from social networks can be applied for a different domain and produce viable results.

1.1 Motivation

The motivation for this kind of idea is twofold. One is to see if and how something as popular as social networks can be applied with something that seems to be unrelated to it, in this case APIs. APIs are not social beings and they certainly don't start unexpectedly communicating with one another, this characteristic is reserved to Agents. This motivation can simply be considered as an explorative compulsion.

The other motivation is to support developers in using, documenting and familiarizing with APIs. They can be big and complex constructs where users can easily get lost and be unable to find the functionality they need. Hopefully the here presented concept for an API network will not only provide experience about combining two apparently different things, but also support developers in working with APIs by providing supportive tools which give for example a better overview of the APIs.

[3] A tool for both modelling and meta-modelling.

1.2 Thesis structure

This thesis is structured as followed: Chapter 2 contains the fundamentals about the approach in realising the API network, i.e. about meta-modelling, social networks and APIs. Chapter 3 describes the cases where the API network might be of use as well as the concept for its structure, visualization and processing. Chapter 4 illustrates the realisation of the concept. Chapter 5 showcases the results when employing the API network on the APIs of the ADOxx meta-modelling toolkit. Chapter 6 provides a short summary and an outlook. Additional material like selected source-code samples and detailed class diagrams can be found in the appendices.

2 Fundamentals

This chapter describes the necessary fundamentals on which the provided concept is based on. It starts with modelling and meta-modelling, which is used as the basic approach for creating the API network concept. Afterwards social networks will be described in further detail with a small survey of the concepts and applications employed in them. At the end an overview of what application programming interfaces (APIs) are considered is provided.

2.1 Modelling and Meta-modelling

Modelling in general is already applied in several domains to help solving more or less complex problems. In the domain of software engineering CASE tools which are based on modelling languages like UML or Entity-Relationship diagrams are employed to support the realization of programs and APIs (e.g. see Mackay et al. [2003]). In economics mathematical models are used to describe parts of reality, analyse them and base decisions on the provided result (e.g. see Elmaghraby [1978]). In chemistry simple graphical models allow focusing on the relevant parts describing the structure of a molecule in terms of its atoms and their connections.

There are many different types of models and ways they can be created through modelling, each with different areas of focus, advantages and disadvantages. The following sections will provide the basics about models, meta-models and modelling methods. This is to insights on the general domain which is used as an approach for the concept as well as the target domain of the APIs used in the provided use case. A survey of the "model" domain has been performed in Höfferer [2008], where most the information for this section has been taken from.

2.1.1 Model

As stated and presented in Höfferer [2008] "The term [model] is of polysemic nature which means that there is no single meaning attached to it but there is ambiguity." Therefore, in

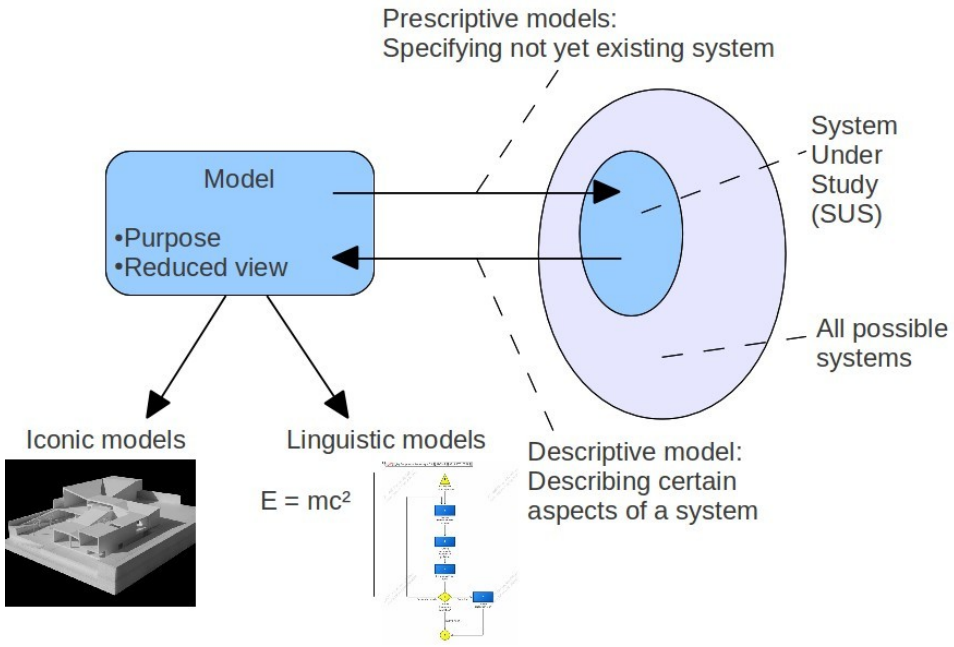

Figure 2.1: Models and SUS based on figures from Höfferer [2008]

order to prevent confusion this thesis will concentrate on models in the context of its domain (business informatics). The core characteristics can be summarized in the following points:[4]

- They represent a certain system, the System Under Study (SUS)
- They are created with an intended purpose
- They concentrate and describe only certain parts the system
- They are linguistic models (textual or graphical)

Generally a model is a depiction of a SUS with a focus on specific characteristics. On what characteristics a certain model can focus is specified by the modelling language, which is described in section 2.1.2. Additionally, models should have some goal or reason for their existence. Hence models should depict something that is or can be realistic. Two typical applications of models are communication and automated processing. Figure 2.1 provides a simple overview of what a model is. In this thesis I will be focusing on linguistic models[5].

[4] For a more detailed analysis of "Model", related terms and definitions with further descriptions please see Höfferer [2008].

[5] As stated in Höfferer [2008]: "Nearly all models used in computer science and business informatics are of the linguistic type"

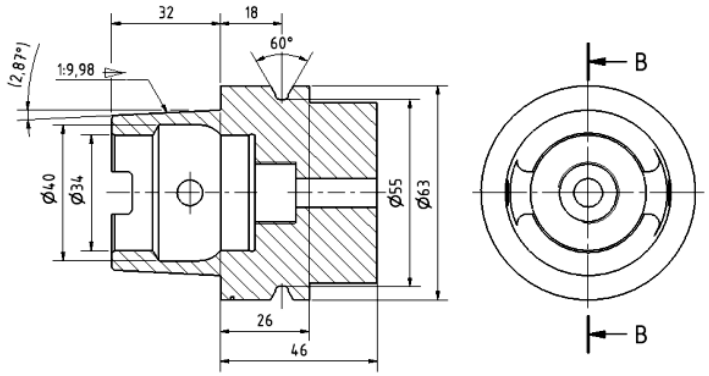

Figure 2.2: Example for a technical drawing taken from Wikipedia[6]

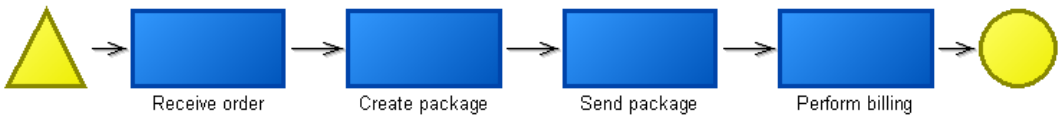

Figure 2.3: Example for a simple process created using Adonis Community Edition[7]

As for the application of models, they are often used for communication in production domains, where they describe how a certain product should look, work and be assembled. Additionally those models can be automatically process to generate a set of instructions for robots which can then create the depicted part. In a similar fashion business processes can be described and communicated to other personal to foster a uniform execution of certain functions or to support the optimisation of the depicted process. These can again be automatically processed to analyse "What-If" scenarios. Two examples for such models can be seen in figures 2.2[6] and 2.3[7].

2.1.2 Modelling Language and Meta-model

As stated before, models describe a certain system. How this system can be described is determined by the modelling language used to create the model. Similar to a natural language a modelling language describes the form (syntax) and the meaning (semantics) of the different elements that can be used in a model. It can be thought of as a set of rules or laws which have

[6] Taken from `http://en.wikipedia.org/wiki/Technical_drawing`, accessed 19.10.2013
[7] More about Adonis Community Edition can be found at `http://www.adonis-community.com/`, accessed 19.10.2013

to be obeyed. Keeping the comparison to natural languages, the syntax covers the spelling and grammar portions. For a graphical (or diagrammatic) modelling language the "spelling" would dictate how the elements should be drawn in terms of shapes, colours etc. and the "grammar" how they can be assembled to create valid models. The form or syntax could usually be checked for consistency by a computer, since the rules can be checked just against the model itself, i.e. the model has to be valid in itself.

The semantics on the other hand are a bit more complicated and for simplicity this thesis will not go further to describe what "meaning" is. However, the interested reader can find more on this topic in Höfferer [2008]. The semantics of a modelling language themselves can be described in different ways, from natural language to mathematical models. Now just as models can be valid on a syntactical level, they can also be valid on a semantic level. To a certain degree the semantically valid parts can be restricted using the grammar. For example "A process can have only one starting point." which is similar to "A sentence can have only one subject.". Grammatically this can be realized by enforcing the "starting point" (or "subject") to be only the first element.

To make matters more complicated the model itself can be semantically valid according to what it is modelling. However checking for this type of semantic consistency can be more difficult since knowledge about the domain is necessary, i.e. the model has to be valid in a certain domain. As an example consider two activities from figure 2.3: "create package" and "send package". The "grammar" allows to put activities in a sequence. This means that grammatically both "*'create package' then 'send package'*" and "*'send package' then 'create package'*" would be correct. However, when applying the domain of common sense to this example it can be seen that it is difficult to send the package before it is created. So while one of the two examples is both syntactically and semantically valid, the other is only syntactically valid.

It should be noted that sometimes it is possible and can make sense to create models which break some rules of the modelling language. Unfortunately automatic processing starts to be difficult if not even impossible in that case, since computerized analysis relies on assumptions which are often covered by the imposed modelling language rules. So all the user might end up with is a nice picture with some additional information.

Now of course there is a need to somehow describe the modelling language so it can be communicated and maybe even automatically process its parts. For this a meta-model can be used, which reapplies the modelling approach to the modelling language. Again this meta-model has to follow a language, a meta-modelling language, to be of real use. Also the meta-modelling language has its own rules which delimit what actually can be expressed using it. Those relations are briefly described in figure 2.4. Often a meta-model is visualized graphically and

used to describe only the structure of the modelling language, i.e. the syntax. They typically concentrate on what elements are allowed and to a certain degree how they can be connected. It is similar to using XML-Schema to describe the possible elements for a valid XML document. A simple example for a meta-model describing a very simplistic process modelling language can be seen in figure 2.5.

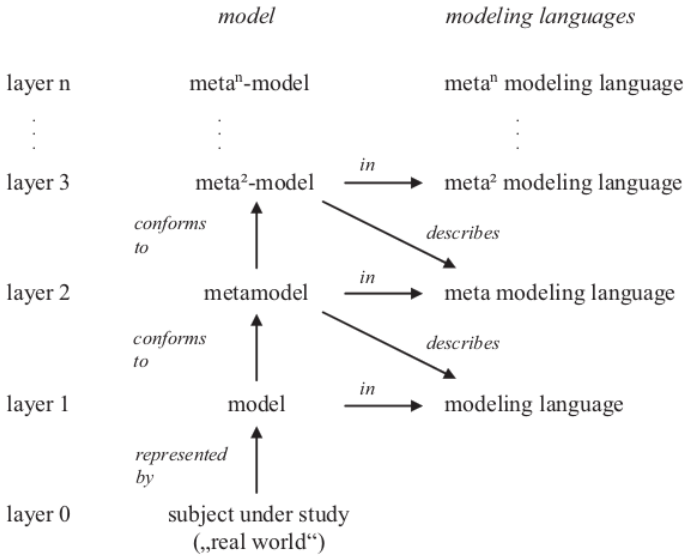

Figure 2.4: Meta-modelling hierarchy taken from Höfferer [2008]

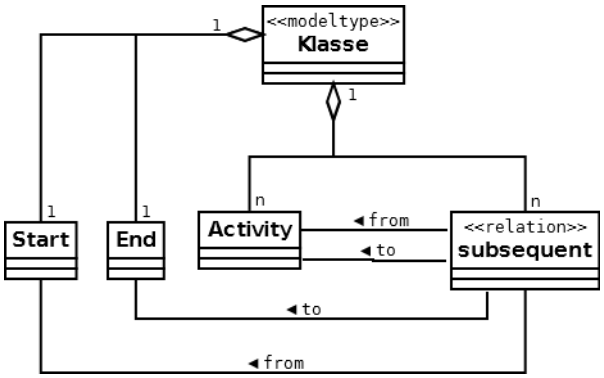

Figure 2.5: Meta-model for a simplistic process model language using UML

2.1.3 Modelling Method

A modelling method according to Karagiannis and Kühn [2002] encapsulates the modelling language and more as can be seen in figure 2.6. Here the modelling language is divided into the three parts described in the previous section: notation (the "spelling"), syntax (the "grammar") and semantics.

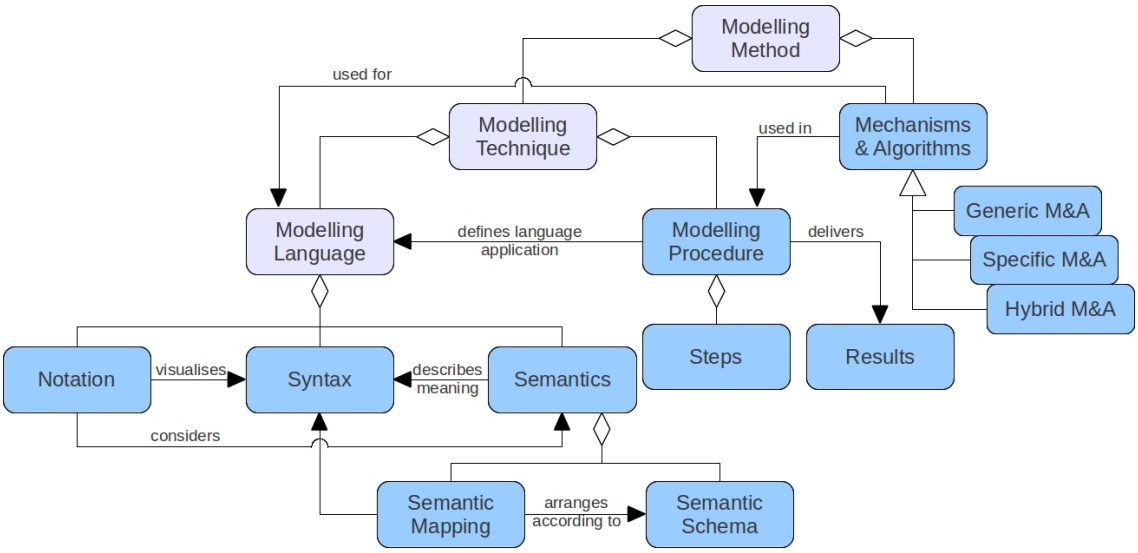

Figure 2.6: Components of a Modelling Method reproduced from Karagiannis and Kühn [2002]

This together with a modelling procedure describing how to employ a modelling language builds a modelling technique. The procedure is like a process of consecutive steps leading towards a specific result. For example those steps could describe how to extract information and how to build a business process out of it. The result would then be a valid and complete business process, if the steps have been followed correctly. The procedure has to be defined in order to have a modelling technique. However, it can be something simple to allow the user to decide which steps to take and just provide guidelines on what is recommended to reach a result.

Additionally mechanisms and algorithms can be employed in the procedure and thus together with the modelling technique finishing of the modelling method. Those algorithms can be independent from the meta-model (generic), they can be specific for a certain modelling language or they can use general concepts found throughout different modelling languages which share similarities like process languages. An example for a mechanism used in a procedure would be to use a path analysis during business process optimization to find and rate the optimization potential of certain paths.

2.2 Social Networks

With the spread of the Internet around the world into everyone's lives long distances were no longer an obstacle for connecting people and allowing to come into contact with millions of others. After some time social network platforms utilizing the World Wide Web like Facebook, Twitter and MySpace were introduced. This made it even easier to reach people both nationally and from all around the world, be it new friends, old schoolmates or potential customers.

Even before MySpace and the like there were simple networks of people connected using the Internet. Those networks were used for instant messaging (IM), like with Microsoft MSN Messenger[8] or ICQ[9]. There people could add each others' contacts to their own lists. However, unlike current social networks the main goal was to exchange instant messages between persons. Still this changed with the evolution of the applications to where they provided more services and activities and evolved into something resembling more a modern social network than a simple instant messaging program.

Several different social networks have been investigated for this master-thesis, including Facebook and Twitter. They have been mainly looked at from the perspective of a private human user, not a group or organization. Following is a short description and some facts about the examined social networks.

Facebook is one of the most popular social networks, with around 700 million users active daily, of which 80% are outside of the United States and Canada [10]. Besides providing a space for a profile for each of its users it is also possible to post information and exchange it with selected people, ones friends or the whole (Facebook) world. Its main attraction is the possibility to connect to millions of other users and staying in touch with them, exchanging news, organizing events and the like. Furthermore Facebook provides applications to "enhance" the experience, games to distract from life and APIs to create own applications.

Twitters on the other hand puts less emphasis on the users themselves and focuses more on the distribution of messages the users compose. The special thing about those messages, also called Tweets, is the length restriction. They can only contain 140 characters[11]. Twitter also provides an API which provides functionality that is also available on the web portal[12], like posting or

[8] Some history of MSN Messenger and Windows Live can be found at `http://windowsteamblog.com/windows_live/b/windowslive/archive/2010/02/09/windows-live-messenger-a-short-history.aspx`, accessed 20.09.2013

[9] which started in 1996, see `http://www.icq.com/info/story.html`, accessed 23.09.2013

[10] according to official Facebook statistics taken from `https://newsroom.fb.com/Key-Facts` accessed on 23.09.2013

[11] It is even possible to post Tweets using SMS in some countries.

[12] according to `https://dev.twitter.com/docs/api-faq#api`, accessed 24.09.2013

reading Tweets. Unfortunately the exact number of users is more difficult to assess. In 2010 Twitter had over 105 million registered users[13] and over 200 million registered users in 2011[14]. The official Twitter blog claims that each day 200 million messages called Tweets are sent[15].

Myspace allows users to present themselves to the world, providing a usually openly accessible profile page, as well as friendship declarations between its users. It allows artists, bands, comedians to showcase their work and connect to their fans or colleagues. To do this the look of the page can be customized, using different designs, fonts and colours as well as special applications like a music player or a picture gallery which are provided by MySpace.

Both Mendeley and ResearchGate focus on research networks, where researchers, their work and the connection to other researchers are considered important. While ResearchGate concentrates more on finding possible partners for collaboration and staying in touch with them, Mendeley provides a platform and tool for managing and citing scientific work, a "reference manager" as it describes itself. Both research networks are rather small compared to Facebook and Twitter with Mendeley having over 2.5 million users[16] and ResearchGate having over 3 million users[17].

The friend of a friend project or simply called FOAF follows a different approach than all other social networks. While Facebook, Twitter, Mendeley and ResearchGate all require an account and take responsibility for storing and providing the social network information i.e. being centralized, FOAF provides a decentralized approach similar the world wide web. Everybody can create a FOAF document describing themselves in a machine processable manner, stating the name, acquaintances etc. and provide it by themselves. The description is based on RDF, defining specific properties with their meaning and usually only contain a part of the social network from the view of one person. They should however contain links to other FOAF documents to be able to navigate through the social network. Since there is not one central place for all FOAF descriptions, accurate statistics about its usage are difficult to find.

As can be seen there are many different types of social networks, with different types of focuses. While Facebook and Twitter are used mainly for communication between friends, MySpace and FOAF allows presenting oneself on the Internet, whether with a custom page or with a

[13] according to `http://www.businessinsider.com/live-twitter-ceo-ev-williams-keynote-from-chirp-2010-4` , accessed 24.09.2013

[14] according to `http://socialtimes.com/200-million-twitter-accounts-but-how-many-are-active_b36952` , accessed 24.09.2013

[15] `http://blog.twitter.com/2011/06/200-million-tweets-per-day.html` accessed 24.09.2013

[16] according to `http://http://www.mendeley.com/`, accessed 24.09.2013

[17] according to `http://www.researchgate.net/`, accessed 24.09.2013

machine processable file. Mendeley and ResearchGate are focusing more on scientific or research networks for the documentation, management and creation of new scientific content.

In the following chapters the structure of the different social networks and possibilities for their visualization as well as processing will be described.

2.2.1 Social Network Structure

The general structure of a social network can be described as a graph containing and focusing on actors. These actors in turn are described using different characteristics. Which characteristics are used usually differentiates one social network from another, along with the intended target audience.

Table 2.1 shows which characteristics from the viewpoint of a person are present in the different examined social networks. To get a broader view the chosen social networks for the examination serve different purposes. Facebook and FOAF follow a very general approach, while Twitter and MySpace concentrate on publishing content. Mendeley and ResearchGate can be classified as scientifical networks, which concentrate on research and collaboration. Still there is overlapping between the different social networks which have been examined. Some very basic functionalities are missing in this table like the settings a user can set, the notification settings or security settings to protect his personal data (e.g. who is allowed to view the email address or who can view posted pictures).

Most of the concepts shown in table 2.1 should be understandable easily enough, but some peculiarities will be addressed now. Typically the email serves two purposes. It can be used to contact the person about updates and new messages, but can also be used as the identifier since it has to be unique on the Internet and therefore also in the social network. The profile picture on the other hand provides a support for a human to identify another person and is always public, unlike the photo album to which access can often be controlled through security settings. The concept of activity has been chosen to generally describe things a person participating in the network can do, like use programs, play games or organize events. Sharing content covers any type of providing information to other network participants in different forms, like text or pictures. Another example for Sharing content is the music player provided on MySpace, where bands can showcase their songs to a broad audience. Presentation and style allows to further personify the look of the own profile to other people. Donations has been derived from the FOAF “tip jar”, which allows specifying how a person can be rewarded.

Additionally there are some other unique characteristics of some social networks which were skipped in the structure table. For one there are social networks which allow connecting with

Concept	**Facebook**	**Twitter**	**MySpace**	**Mendeley**	**ResearchGate**	**FOAF**
Name	X	X	X	X	X	X
Identifier	X	X	X	X	X	X
Email	X	X	X	X	X	X
Current location	X		X	X	X	X
Time zone		X		X	X	
Hometown	X		X			
Gender	X		X	X	X	X
Birthday / Age	X		X	X	X	X
Language(s)	X	X	X			
Biography / About me / CV	X	X	X	X	X	X
Profile picture	X	X	X	X	X	X
Relationship status	X		X			
Family	X					
Friends / Contacts	X		X	X	X	X
Grouping of persons	X		X	X	X	X
Followers		X				
Work / Employment	X		X	X	X	X
Education (university, school, experience)	X		X	X	X	X
Religion / Political views	X		X			
People who inspire / Favourite quotes	X					
Interests (music, books, research, etc.)	X		X	X	X	X
Contact information (mobile, chat id, etc.)	X	X	X	X	X	X
Webpage / Weblog	X	X		X	X	X
Photo album	X		X			
Activities	X		X		X	
Sharing content	X	X	X	X	X	
Presentation / Style		X	X			
Awards / Grants				X		
Documents and Publications				X	X	X
Conferences					X	
Projects						X
Donations						X
Job offers					X	

Table 2.1: Structure of several social networks. State: July 2011

other social networks, to voluntary identify and connect the same users in both networks. For example both the Mendeley and the ResearchGate account can be linked to a Facebook account. This however is the creation of interconnection between several social networks and is outside of this thesis scope. Some social networks (e.g. Facebook, Mendeley) also provide a service to scan an existing email account for contacts and find those in the social network.

Also some social networks allow to choose an account type, like MySpace for example. On MySpace the user can choose a talent, like musician, filmmaker or comedian, which determines the account type. Those dedicated account types for different purposes allow to concentrate on different aspects and better connect the people of the network. For example a person can indicate that it is a fan of a band. Also bands and comedians can promote their gigs to the other people and use a music player to provide free samples. On MySpace the profile pages can also be used as a representation in the web (i.e. a public web-page). Another peculiar thing about MySpace was that it allowed to rank the linked friends, which could lead to some drama among a circle of friends.

Furthermore some social networks like Facebook provide instant messaging as part of their platform, coming full circle with the start of "connected people" through ICQ or MSN. Also Twitter allows to post open messages directed at another person, however those can be read by everybody.

2.2.2 Social Network Visualisation

Since social networks are used by individual humans the presentation also concentrates on providing the information from that point of view. A big overview of a graph is usually not provided to the user, instead only the detailed profile pages are used. Searches can be performed generally resulting in a list with all the fitting entries. These are very similar to what search sites like Google or Bing provide. However instead of providing a content excerpt the selected profile picture is provided for humans (or a selected or default icon for other entities). One such result from Facebook can be seen in 2.7 where a search for "Markus" has been performed.

Since it is a social network, the typical approaches for visualizing networks can be employed. However since the proper visualisation of networks can be a thesis topic in itself this thesis will not go into further details.

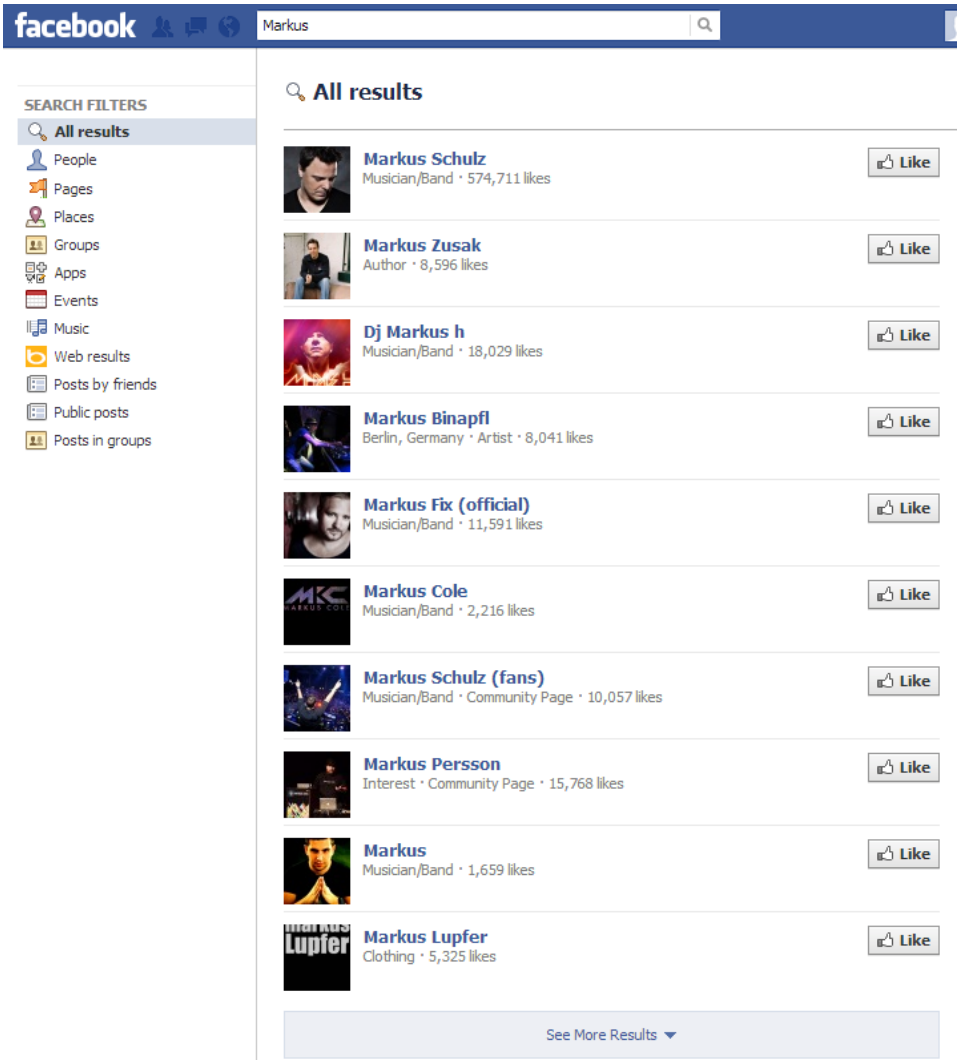

Figure 2.7: Search for “Markus” performed in Facebook

2.2.3 Social Network Processing

From the users point of view the provided functionality of the social network focuses on further connecting the people. Therefore a lot of the visible results of processing are recommendations, for friends, for events or other things. Additionally popular social network platforms also provide information services, notifying the users whenever something of interest might have happened. Another thing which draws people to social networks are also the provided applications or games therein, e.g. FarmVille on Facebook. Also when looking at the statistics provided by social networks they have to process the activity of the persons, i.e. when a user actually uses the social network.

However since a social network still can be seen as a network or graph of people (and other entities) with relations to one another they can be processed as such. Still for each of those processing executions there is the question of what the added value of generating new information is. One of the first processing applications for social networks (or sometimes networks in general) found in literature was the grouping or clustering of the people according to some criteria. Clauset et al. [2004], Newman [2006], Branting [2010] and Xu et al. [2010] address this type of processing. Another application was the prediction of friendships based on the grouping of people. In the case of Zheleva et al. [2010] the groups indicated families.

The before mentioned processing possibilities for social networks have been used as inspiration for the envisioned applications described in section 3.1. Also several of the processing applications for social networks found in the literature are described in more detail in the presented concept found in section 3.2.3.

2.3 Application Programming Interfaces

Today it is almost impossible to avoid the concept of an *Application Programming Interface* or short API when developing applications, tools or extensions. Most programmers understand what is meant, even though everybody might have a slightly different comprehension of what an API really is. Definitions in the world wide web are often distinct from one another, since they usually consider API under a different aspect. For instance PC Magazine [2011] defines an API as “A language and message format used by an application program to communicate with the operating system or some other control program such as a database management system (DBMS) or communications protocol”. Computerworld [2000] describes an API as “a set of standardized requests, . . . , that have been defined for the program being called upon. . . . In essence, a program’s API defines the proper way for a developer to request services from that

program.". On the other hand How stuff works [2011] describes an API as "a set of programming instructions and standards for accessing a Web-based software application or Web tool". This definition takes a more modern approach towards web service orientation. In contrast to this Erl [2008] compares a service contract with its available operations to a (traditional) API, but does not use API as a synonym for it. In Hansen and Neumann [2005] the API is the interface through which a software component provides functionality. Software components being defined as a piece of software that provides set functionality through well-defined interfaces (their API). Also in this definition software components are reusable and can be exchanged by other compatible components. Compatible meaning they provide the same interfaces and functionality.

However all of these definitions have the concept of interface as a common baseline. Nevertheless an API differs from a general interface by its main intention, to allow other applications to access and communicate with a service, an operating system or a simple program. Interface itself on the other hand has a more broader meaning which also covers (graphical) user interfaces and private interfaces inside an application which are not accessible from outside. Therefore some form of documentation on how to use an API is vital and should be available to developers. Otherwise nobody except the creator would know how to make use of it. Furthermore APIs need to be carefully maintained, considering backwards compatibility and side-effects, if any occur.

But why invest time and effort into the development, maintenance and support of an API? A fitting quote is provided by PC Magazine [2011]: "Building an application with no APIs, says Josh Walker, an analyst at Forrester Research Inc. in Cambridge, Mass., 'is basically like building a house with no doors. The API for all computing purposes is how you open the blinds and the doors and exchange information' ". An API allows others to reuse functionality from one application and therefore provide additional value, but at the same time using an API makes an application depend on it and its implementation(s).

Many different APIs exist today as can be seen in the following list containing some available APIs:

Windows API - also known as Win32-API, provides functions for creating graphical user interfaces, accessing system resources, handling input devices and much more[18].

Linux Kernel API - an API provided by the Linux kernel[19].

Cocoa API - Apples API for creation of native Mac OSX and iOS applications[20].

[18] `http://msdn.microsoft.com/en-us/library/cc433218%28VS.85%29.aspx` accessed 24.09.2013
[19] `http://www.kernel.org/doc/htmldocs/kernel-api/` accessed 24.09.2013
[20] `http://developer.apple.com/technologies/mac/cocoa.html` accessed 24.09.2013

IBM i5/OS API - allows access to the IBM® i5/OS operating system and provides functionality for many different tasks[21].

Java API - an API for the Java Virtual Machine, providing functionality for graphical user interface, data processing, network connections and more[22].

Eclipse Platform API - used for the development of plug-ins for Eclipse[23].

ADOxx API - API(s) provided by the ADOxx platform for accessing and manipulating its model and object repository and adapting the GUI.

Protégé API - allows extending and reusing parts and functions from Protégé, like class tree visualization, mouse events and the like[24].

Facebook API - provides interfaces to both integrate Facebook into own websites and creating of applications for Facebook[25].

Twitter API - allows access to data on Twitter, about its users, their tweets (messages) as well as a search functionality[26].

Flickr API - offers access to Flickr data, the photos and their metadata[27].

Windows Live API - an API for the Windows Live service[28].

Google APIs - several APIs provided by Google for Search, Maps, Google Documents, etc.[29]

Geolocation API - an API to "access geographical location information associated with the hosting device", currently a proposed recommendation from W3C[30].

OpenGL API - used for creation of hardware accelerated computer graphics both 3D and 2D.[31]

As can be seen there are APIs not just for different problems, but also on different levels. While some are more general when developing an application, like the Windows API covering many different aspects, others are more specific like the Twitter API concentrating on retrieval of their social network data. Also the APIs are structured differently, while some follow a procedural

[21] `http://pic.dhe.ibm.com/infocenter/iseries/v6r1m0/topic/apiref/api.htm` accessed 24.09.2011
[22] `http://docs.oracle.com/javase/6/docs/api/` accessed 24.09.2013
[23] `http://help.eclipse.org/indigo/topic/org.eclipse.platform.doc.isv/reference/api/index.html` accessed 24.09.2013
[24] `http://protege.stanford.edu/doc/dev.html#api` accessed 24.09.2013
[25] `http://developers.facebook.com/` accessed 24.09.2013
[26] `https://dev.twitter.com/docs` accessed 24.09.2013
[27] `http://www.flickr.com/services/developer/` accessed 24.09.2013
[28] `http://msdn.microsoft.com/en-us/windowslive/ff621314` accessed 24.09.2013
[29] `https://developers.google.com/products/` accessed 24.09.2013
[30] `http://www.w3.org/TR/geolocation-API/` accessed 24.09.2013
[31] `http://www.opengl.org/documentation/` accessed 24.09.2013

approach providing functions or function prototypes to call, like the OpenGL API, others are more oriented towards objects which in turn provide functions, like the Cocoa API or the Java API. Furthermore there are several different ways of using the listed APIs. For example the Windows API can be used from a C++ program, while some parts of the Twitter API can be accessed as REST Services and others through an HTTP stream [32].

Some bigger APIs can also be divided into smaller categories. For instance the overview of the Windows API [33] divides the provided functionality into seven categories: *a*) Administration and Management, *b*) Diagnostics, *c*) Graphics and Multimedia, *d*) Networking, *e*) Security, *f*) System Services and *g*) Windows User Interface. The IBM® i5/OS API is divided into even more categories, 50 to be precise[34]. The Programmable Web[35] provides a catalogue of different APIs which can be accessed from the Web and contains 10000 entries (as of 24.09.2013) assigned to one of 68 different categories. The website also provides a catalogue of web mashups[36] which employ one or several APIs from their catalogue.

I personally started programming with C++, PHP and Java. This influences my view of APIs, since I came into contact with them by using programming libraries. These libraries encapsulate some functionality and I was able to accessed it using the API they provided.

In many cases the code for the used libraries was also available as open-source. Unlike open-source however APIs do not require the source code of the application to be provided to the customers, users and developers. This allows to keep control and the responsibility over the integral parts of the application as well as their encapsulation, while still providing some degree of customisation and reuse. Furthermore APIs provide an easier access to the functionality than open-source, since they do not require to hunt through thousands of lines of code to find the part that a developer requires. Instead the developer has to hunt through (hopefully) a shorter documentation. On the other hand an open-source application allows fine-tuning and to deal with how it actually works instead of taking the functionality for granted. This also helps with debugging, since a developer does not need to deal with a black box. Still an open-source application can have APIs to get the best of both worlds, as is the case of the Linux Kernel.

In order to be able to create a concept and an implementation for an API network, the definition of what an API actually is still has to be provided. For the purpose of this work a simple definition of API based on the found information will be used: *An Application Programming*

[32] The Streaming API, User Streams and Site Streams use this type of access. A simple HTTP request is send to the server and the connection is kept open for as long as required.
[33] `http://msdn.microsoft.com/en-us/library/aa383723%28v=VS.85%29.aspx` accessed 24.09.2013
[34] for a complete list please refer to `http://pic.dhe.ibm.com/infocenter/iseries/v6r1m0/topic/apis/aplist.htm` accessed 24.09.2013
[35] `http://www.programmableweb.com/apis`, accessed 24.09.2013
[36] A web page or application combining data and/or functionality from two other web sources.

Interface defines everything required to allow the delegation of functionality. It should cover both the technical parts as well as the necessary information for the human developer to use it. This definition is only used here in the context of electronic data processing.

3 “API-Social Network”

Using the information from chapter 2 the concept of the API network will be based on the modelling method approach where the designated system under study is APIs. Therefore the model (or models) created from the here presented concept are graphs or networks of APIs, just as the model(s) in a social network are the network (or graph) of people.

In this concept the focus is on the specification of the syntax, notation, semantic (see subsections 3.2.1 and 3.2.2) and processing (see section 3.2.3). The user interface and proper depiction are however less focused on since covering them properly would require content enough for an additional master thesis. Additionally the possible procedures will be brought closer to the reader through the envisioned applications which are described in this chapter. However since it is tried to keep the concept simple the general procedure can be described in three steps: 1) enter data (manually, using a script, etc.), 2) execute mechanisms and 3) act according to results. Since the concept is based on the modelling method approach, it will also consider for the implementation the results from Schremser [2011] which are available on the Open Model Initiative (more in section 4.1 and 4.2).

Even though the API network presented in this thesis is based on observations of social networks, there is no denying that an API differs from a person. For instance when contacting a person he/she may not reply, either because he/she is currently unavailable or doesn't want to talk. When contacting an API on the other hand there will be a result in a mostly understandable reply even when it is unavailable. So communication with APIs is much less ambiguous than communication with people. However the down side of this is that there is often only one correct way how to communicate with an API. Because of this also some parts specific for APIs are part of the concept.

The following section of this chapter describe the “API-Social Network”, or simply API network. First the envisioned applications for such a network are presented and afterwards a concept fitting to those is described with its structure, a possible visualisation and the general algorithms so it can be processed.

3.1 Envisioned Applications for API Networks

Before creating a concept for an API network, it is important to think about its uses and what benefit it could provide. Therefore several applications for employing an API network are described in this chapter. Each of the here presented cases contains a general description of the idea and thoughts, a short example, the requirements or preconditions for the employment and the expected result. The provided examples do not always portray the exact reality.

3.1.1 Application: Documentation and Overview

One of the more general applications of an API network would be for documentation purposes and to provide an overview of the available APIs. The overview can help finding possible applications to hand over certain tasks as well as to show the available functionality provided by the applications. This would help answering the questions "Which application can help me?" and "What can this application help me with?" and could either be directed towards humans or even towards machines. The later would allow an automatic discovery for agents to find and access functionality on demand. Also the network can be used to dive deeper into the documentation of each API to help a developer employ it correctly. A network description of APIs can also provide numeric measures about each API, like the amount of different versions or how many APIs one actor is responsible for.

Example:

Name	Repository
Signature	Ado:IAdoRepository
Version	1.0
Super-class	Ado:IAdoRepositoryObject
Documentation	Interface of the repository, where models and objects are stored. It also provides events associated with the managed objects.

Preconditions:

- APIs with identifiers
- Noteworthy characteristics of APIs
- Classification of APIs by application that provides them
- Search/Discovery functionality

Expected Result: Informed developers and users of APIs and easier access to a programs functionality. Additional measures, like the amount of provided functions or APIs available from the same location or website, could also be provided.

3.1.2 Application: Infer transitive inheritances

An API network can be used to infer additional relations for each API, for instance all inheritances based on the existing ones. This inference is similar to determining family membership in family trees or social networks, where parent relations can be used to determine the ancestors or siblings of persons. This application only make sense for APIs using a description language allowing inheritance and when the APIs make use of it, for instance when they are describing the objects with their structure and functionality.

Transitive inheritances are of interest for identifying the origin of properties or functions and providing simple overviews of the hierarchy as a tree. The level of the hierarchy also provides a clue about how generic or specific the API is in relation to the other APIs. Furthermore inheritance is also a dependency since changes in an API influence all APIs which inherit from it.

An Example for transitive dependencies can be seen in the following table and in figure 3.1.

Signature	Ado:IAdoInstance
Version	1.0
Super-class	Ado:IAdoRepositoryObject
Inferred Super-classes	Ado:IAdoRepositoryObject, Ado:IAdoIDObject, ...
Inferred Sub-classes	Ado:IAdoAttributeInstance

Preconditions:

- APIs with identifiers
- Inheritance relations between APIs
- Specific API as starting point (optional)

Expected Result: An overview of all the inheritance relations providing information of the super- and sub-relationships between APIs in form of inheritance hierarchies. This type of analysis might be of interest for both the whole graph as well as for single APIs, indicating which parts are inherited from where. This is useful to find possible types for an abstract interface and when polymorphism is available as well.

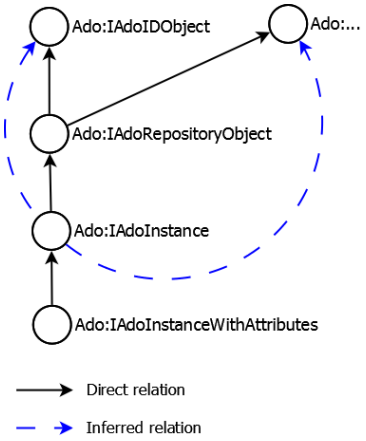

Figure 3.1: Example for inferred inheritances

3.1.3 Application: Infer transitive dependencies

The network can also be used to infer transitive dependencies based on the existing ones. Again it usually does not make sense to try extracting dependencies when looking only at simple functions as APIs since the documentation for an API function usually avoids stating what other functions it calls. However, when APIs follow an object-oriented approach and are descriptions of a possible object, dependencies can found by looking at the required function parameters and return values, since those can reference other APIs and practically create a dependency. Furthermore inheritance can both pass on dependencies and be itself a dependency.

The transitive dependency is of interest since changes of a fundamental API can influence not just the APIs directly applying it, but the ones using those as well. Therefore the length of the shortest dependency path might be of interest. However there is still a difference between direct and transitive dependency, since transitive dependencies are a weaker link for influencing the APIs. The longer the dependency chain and hence the distance between two APIs, the less likely it is that unexpected behaviour occurs. The total number of incoming dependencies can of course also be used as a measure for how important an API is.

An Example for transitive dependencies can be seen in the following table and in figure 3.2.

Preconditions:

- APIs with identifiers
- Existing dependency relations between APIs
- Specific API as starting point (optional)

Signature	Ado:IAdoInstanceWithAttributes
Version	1.0
Dependent on	Ado:IAdoEnumerator Ado:IAdoAttributeInstance Ado:IAdoRelationInstance, ...
Inferred Dependencies	Ado:IAdoCoreAttributeValue Ado:IAdoWritableVariant Ado:IAdoRelationEndpointInstance, ...

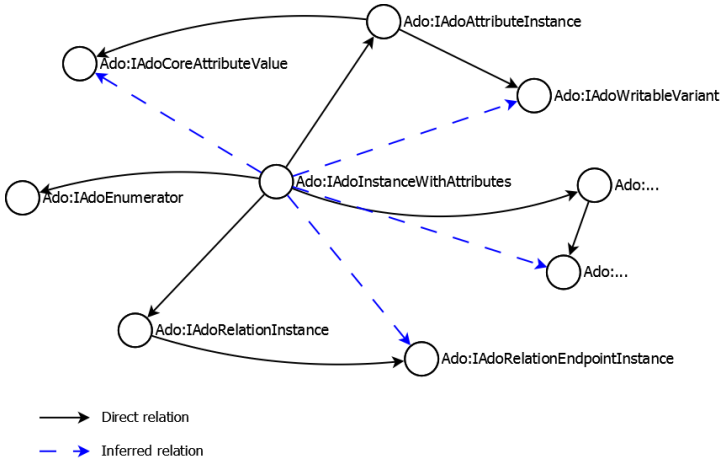

Figure 3.2: Example for inferred dependencies

Expected Result: An overview of an APIs dependencies on other APIs, as well as providing a link in the other direction showing all APIs dependent on an API. This type of analysis might be of interest for both the whole graph, as well as for single APIs, when a developer is thinking about changing a specific API.

3.1.4 Application: Calculate similarity

This application is based on friendship recommendations often provided by social networks to further increase the information available about a person. When friends are recommended this is usually based on some form of similarity like having the same friends, attending the same school or having the same workplace. Similarity between APIs could be determined in a similar fashion using the available information to calculate how similar two APIs are. The similarity would be for a certain aspect, like functionality or affiliation to a certain application. One thing which should not be forgotten is that in social networks only recommendations are provided which the user has to manually verify or ignore. Therefore the similarity is used as a support

for the user and should allow the explicit verification by a human.

For an example with similarity between APIs see figure 3.3.

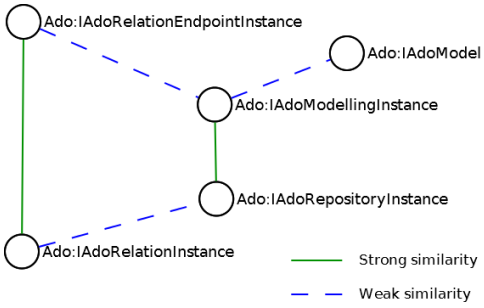

Figure 3.3: Example for similarities between APIs

Preconditions:

- APIs with identifiers
- Characteristics of APIs to be used to calculate similarity (depends on the aspect which is of interest, like the provided functionality)

Expected Result: The similarities between two or several APIs for certain factors. These similarities can then be used to find alternative APIs with similar functionality or some other characteristic which is of interest. Furthermore similar APIs could be used to further assist in understanding an API.

3.1.5 Application: Similarity Clustering

Organizing the APIs of the network by putting them in clusters according to their similarity relations would help finding several APIs with a similar aspect. For this the manually provided similarities and maybe even automatically calculated similarities could be used. They should however all be for the same or a related aspect. The idea is to have fewer clusters for a certain aspect than APIs in the network. This would provide users with a more manageable amount of elements, offering a better overview.

For an example of API clusters based on similarity see figure 3.4.

Preconditions:

- APIs with identifiers

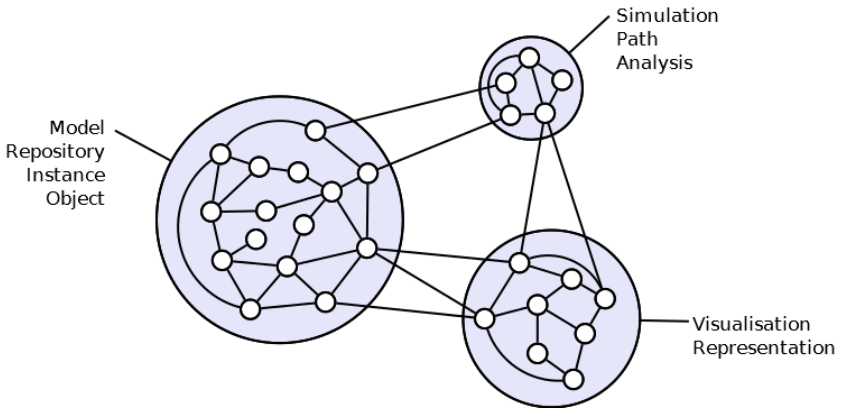

Figure 3.4: Example for labelled clusters based on API similarities

- Similarity relations between APIs

Expected Result: Clusters containing APIs which are similar to one another. Such groups of similar APIs could help in finding alternatives to known APIs or find an API for a certain task in the first place by providing a general group. For instance one cluster could contain all APIs for data access, while another would contain all APIs for data manipulation and a third one for visualization. The clusters could be labelled to indicate something general about the contained APIs.

3.1.6 Application: Dependency Clustering

This application is related to the clustering using the similarity relations, though unlike the previous case here the clusters would contain APIs with many dependencies between them. Analysing these clusters could help in identifying parts with high (poor) coupling where redesign might be appropriate. Furthermore the amount of clusters could provide a general measure for the coupling of the APIs. For this only direct (not transitive) dependency relations make sense.

For an example of API clusters based on dependency see figure 3.5.

Preconditions:

- APIs with identifiers
- Dependency relations between APIs

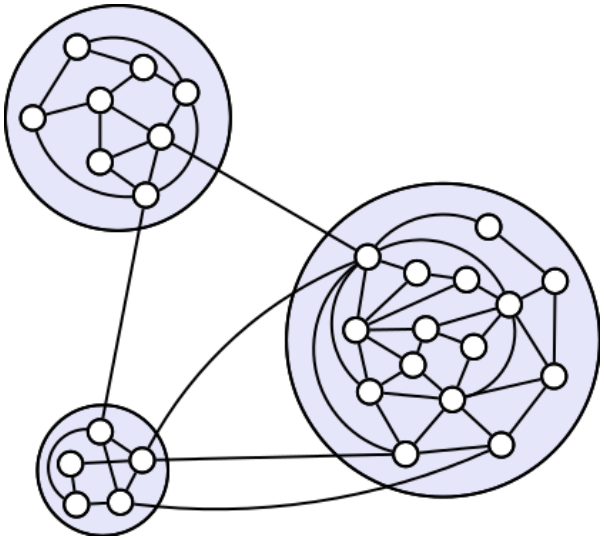

Figure 3.5: Example for clusters based on API dependencies

Expected Result: Clusters containing APIs with a high dependency between one another. This could help identifying parts with many dependencies in between and possibly poor coupling, especially if a cluster contains APIs providing different types of functionality which should better be separated.

3.1.7 Application: Analyse evolution of APIs over time

When introducing the concept of versioning to the API network, this can be used to track the evolution of the individual APIs. The change of properties and relations can be analysed over time, showing the API network as “living” network and some of its history. For this either the evolution of an individual API could be shown as a version tree or a snapshot of the whole network or a part of it at a certain time.

An overview of the whole network could help identify APIs which rely on other outdated APIs or resources. If an API has some kind of relation, like inheritance or dependency, to another part in the network which is marked as obsolete then it should be updated. Additionally the evolution of an API could provide some hints about how active the development is, where APIs with more changes in their characteristics would be considered more active. Also the evolution of the involved people can be of interest and for a set of APIs this could be visualized in a similar fashion as in the Gource project[37].

[37] Which can be found at `http://code.google.com/p/gource/`, accessed 26.09.2013

Signature	Ado:IAdoModellingInstance
Version	2.4
Evolution	
Version 1.0	Initial version
Version 1.1	* Added new read only attribute "nameAttributeInstance"
Version 1.2	* Removed "getAttributeInstance" * Added "getAttributeInstanceWithAttributeName"
Version 2.0	* Added "getAllRelations" * Added "getAllRelationsFromDS" ...

An example for the evolution of an API can be seen in the following table.

Preconditions:

- APIs with identifiers
- Different versions of the same API with different characteristics
- Specific API as starting point (optional)

Expected Result: An overview of how things have changed or evolved throughout the versions, especially regarding the relations to other APIs in the network. A kind of movie on how the relations between APIs change over time could be shown. Could also be used to automatically create change logs and inform stakeholders about changes.

3.1.8 Application: Determine important APIs

When facing many different APIs it is useful to have a prioritization in order to know on which of them to concentrate. An example in reality can be found in production, where certain tasks can have a higher priority than others to maximize profit, like prioritizing the creation of a more lucrative product instead of another one.

A measure could be used to determine the importance of an API. This could then either be used to visualize the APIs using a colour code or provide a sorted list. An example for the colours would be to use red for the most important "hot" APIs and blue or green for the less important "cold" APIs. However there can be different approaches for how to measure the importance. One would be to prioritize APIs on which many others depend (i.e. using the direct and inferred dependencies). Another one would be to get execution information and log how often an API is called and base the measure on this. Even a mixture of both is plausible

where dependencies between APIs also state how important one API is for the other or how many times one depends on the other.

For an example depicting the importance of APIs based on dependency is shown in figure 3.6.

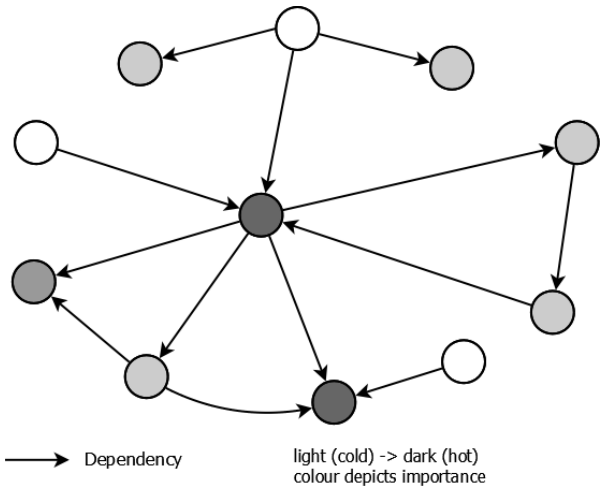

Figure 3.6: Example showing importance based on dependency as colours

Preconditions:

- APIs with identifiers
- API characteristics on which the importance/priority can be based

Expected Result: Identified importance or priority for the APIs based on the measures, allowing to identify “high priority targets” for development or other tasks. Also if the used measure is based on the employment of the APIs by the “consumers” it can be used to determine what directions of application development are more important than others.

3.1.9 Other Applications

This subsection will cover some possible applications for an API network which might be considered ’playful’ or be useful for marketing purposes. Their added value is considered debatable which is the reason for grouping them in this section. They will not follow the previously used structure and instead will simply be described with normal text. Also their implementation is considered less of a priority than the previously described applications.

Plot locations of APIs

This is based on the idea of "seeing your social network friends with their locations on a map". This allows to see over how many countries or even continents the network of friends or in this case APIs spans. The most notable service which provides geographical maps is Google Maps, but others are available as well [38]. Now if locations are provided for APIs, like the server locations for available web services or the development locations for applications providing the API, then a set of APIs could be used to place markers on a map showing where they are located and maybe even draw relations (e.g. similarity, inheritance, etc.) between those places. The set could be determined for instance by selecting all APIs which are developed in a project or based on the dependencies between APIs.

Giving awards to APIs or their developers

Certain portals like YouTube tend to give awards for achievements, which are usually based on some form of voting. Examples would be "Most seen video", "Most liked post" or "Best content provider based on community voting". This can maybe be also applied to the API network with awards like: most active API development, the most used API, most innovative API or "indie" API of the month. This could further motivate developers in partaking in the API network and provide free advertisement. Another similar thing would be a "Seal of Approval/Quality" which could be handed out to certain APIs. For this the specific evaluation criteria would have to be defined and provided, like "consistent employment of best practices".

API specific events

In many social networks it is possible to organize real life events and invite people which can accept, reject or state that they are undecided. The events in the API network could be organized on two levels: they could be for the people behind the APIs or they can be seen as events happening between APIs in the form of an orchestration or a collaboration. The second one would however require a formal way to describe and coordinate such events. The first one would focus on getting people together that for example have used or want to use a specific API or application that provides the API.

[38] Interestingly enough there are not only maps of the real world available, but also of fictional worlds like 'Middle Earth' from the 'Lord of the Rings' or even for worlds from computer games like 'World of Warcraft'.

3.2 API Network Concept

When looking through the described applications one can see that a lot is dependent on the structure of the APIs, usually requiring a simple graph with nodes (APIs) and relations of different types (depending on the case). Only few cases require more or rather different information to be realised. Furthermore the granularity at which the APIs are considered is very important and can change the results. For instance when considering only functions it is difficult to determine dependencies on API level, because they will often not contain other function calls in their signature or API description. However on the object-oriented level where an API is a bunch of data with a set of functions, which in turn can get passed other APIs we can state dependencies between these APIs.

The goal of this section is not to describe in detail how to cover every previously described use case, but instead concentrates on the description of a concept which supports the use cases. In this chapter first a structure for the API network will be derived both from the surveyed social networks as well as using the previously described cases. Afterwards several ideas for the visualization of the API network will be outlined. In the end the algorithms and mechanisms necessary for certain use cases will be described in further detail.

3.2.1 API Network Structure

Just like for social networks the structure of an API network is a major part, since it affects what data can be provided, how the API network can be employed and what algorithms and mechanisms can be used on it. Table 3.1 shows the concepts found in social networks and the derived concepts for an API in the API network. The table also contains social network concepts for which not fitting concept for APIs could be thought of, since not everyone makes sense in the domain of APIs. Also the *Exceptions* concepts is missing a social network counterpart, since nothing fitting in the domain of social networks was found.

The following list provides descriptions for better understanding of the different concepts:

Name for a simple labelling of an API to give a human some idea about what it does.

Signature is the unique denomination of an API and therefore useful as an identifier, since different APIs can have the same names in different applications or tools. The signature can depend on the type of API, whether it is a simple function or an object description. It should contain enough information to relate it to the application, the module or package, the class, the function and parameters, if any of those are present. Examples would be a functions signature or the mixture of a module- and class-name.

Social Network Concept	**API Network Concept**
Name	Name
Identifier	Signature
Email and Contact information	Call
Current location and Hometown	Location
Time zone	—
Gender	—
Birthday / Age	Release date
Language(s)	Definition language
Biography / About me / CV	Evolution
Profile picture	—
Relationship status	Development status
Family	Inheritance
Friends / Contacts	Similarity
Grouping of persons	Clustering of APIs
Followers	Dependencies
Work / Employment and Projects	Uses
Education (university, school, experience)	Responsible actor
Religion / Political views	—
People who inspire / Favourite quotes	Applied patterns
Interests (music, books, research, etc.)	Description
Webpage / Weblog	Webpage
Photo album	—
Activities	—
Sharing content	Data exchange
Presentation / Style	—
Awards / Grants	—
Documents and Publications	Documentation
Conferences	Events
Donations	Compensation
Job offers	—
Account types (Person, Organization, etc.)	Granularity
—	Exceptions

Table 3.1: API network concepts derived from social networks

Call states how to contact or make a simple call to the API. This depends heavily on the implementation. If an API is accessible as a web service then the WSDL could be used. It basically contains the technical information required to use the API.

Location tells where an API is located in the world, where it is from. For some APIs it may be difficult to assign to a specific location, therefore it can be omitted at times.

Release date specifies when the API has been made public in the described form.

Definition language states what language has been used to define the API. Examples would be IDL, WSDL, Objective-C or Java.

Evolution describes the history of the API and is close to versioning.

Development status shows the current status of the API. It should provide a measure to give an idea about the reliability of the API and whether it or an alternative should be preferred.

Inheritance is meant in the same sense as in object-oriented programming languages. Inheritance is useful when an API describes the structure and functionality of objects instead of simple functions to calls.

Similarity denotes some form of likeness between two APIs. This can be because they provide similar functionality, are used in the same program or just have the same name. Depending on the use case different aspects for similarity can be deemed useful.

Clustering of APIs allows to put several APIs in one group with some meaning to it. It could be seen as a simpler form of similarity, where all APIs of a cluster are considered alike in some aspect.

Dependencies show which APIs rely on other APIs or resources. Like inheritance it makes only sense for certain types of APIs, because function prototypes generally don't depend on other functions.

Uses provides a list of application areas where the API is used in, like projects. This can provide a reference to help developers to use this API themselves or can be used for advertising.

Responsible actor allows to contact either a person or an organisation to provide feedback, convey wishes or ask questions about the API.

Applied patterns lists one, several or no patterns applicable or used for this API.

Description provides a human with additional information on what the API is for, on what to pay attention to etc.

Webpage or weblog or forum (i.e. external source) for an API with even more information, if one is present.

Data exchange further describes how data is exchanged with the API, how it is to be structured, what type of encoding should be used etc.

Documentation should point or contain the official documentation of the API. It is important because APIs cannot be used without any knowledge about them and are difficult to use if the documentation is missing.

Events which have some connection to the API, either because it was presented at a conference, a workshop was held or to further develop the API at a meeting. Future events can be used to advertise an API.

Compensation can allow the responsible actor of an API to specify some of payment for the use of their API.

Exceptions are important, because when calling an API's implementation it is possible that things go wrong, in which case an exception or an error is returned. Information about what can go wrong as well as why and how to identify it can help developers using an API to properly handle those situations.

Granularity or the type of API is of interest since it allows to distinguish the description of APIs which employ different paradigms, like procedural APIs where only functions are defined or object-oriented APIs where object descriptions contain their properties and functions. As described in 3.1 several cases make only sense for APIs at a certain granularity.

Most of the conversions from social network concepts to API network concepts are straight forward, like *Name*, *Birthday* and *Family*. Still some will be explained in more detail to shed some light on the more complicated conversions. The correlation between *Friendship* and *Similarity* is based on the thought that friends in social networks usually have some thing in common, for example similar interests or working for the same company. One major similarity between the *Followers* relation from Twitter and *Dependency* is that both are directed relations. Furthermore an actor in the social network follows someone they receive messages from other actors resulting in communication. An API who depends on another API or some object communicates with those as well to call functions or retrieve data. The *Employment* concept states where people use their skills for something bigger and *Uses* indicates something comparable for APIs, by stating where they are employed and where they add additional value.

It can be seen as a more general inverse concept of the dependency. The *Responsible actor* is derived from *Education*, because the education defines a lot who people are and what they can, just like the responsible actor defines this for their APIs. The concept for different types of *Granularity* is deduced from the different types of accounts that are often available for social networks, like businesses, organisation and artists. APIs can also fall into different categories based on their scope like modules, object and class descriptions or simple functions and APIs may be handled differently depending to which type they belong.

Using these concepts a structure is derived in form of properties and relations and are depicted in figures 3.7 and 3.8 and are further described in tables 3.2 and 3.3. Entity-Relationship models are chosen for the description, because they provide a clear and intuitive view on the designed data. This structure should provide more than enough information for the cases described in the previous chapter.

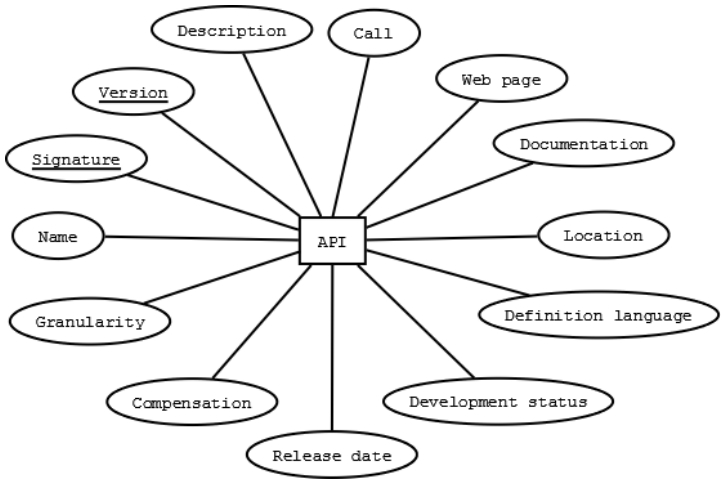

Figure 3.7: Properties of an API network

Again most of the transitions between the concepts and the described structure should be straight forward. However, the structure does not contain an explicit property or relation for the *Data exchange*, since the information on how to exchange data with the API should be covered by the *Call* property. Additionally *Events* and *Uses* are both handled through the *applied in [Events]* relation, considering *Projects* to be *Events*. Furthermore two different approaches are feasible to depict the *Evolution*. The first would store the different versions and use those to determine the required changes between two versions while the second one would store a reference version, usually the newest version, as well as the changes performed between all the versions allowing to deduce the versions by following the path. In this concept the first approach is used and is not described by one single property or relation, but instead divided

Property	**Datatype**	**Description & Constraints**
Name	String	(Mandatory)
Signature	String	(Mandatory, Identifier)
Version	String	(Mandatory, Identifier) The version is defined as a character sequence to allow different styles of version numbers.
Description	String	(Optional)
Call	URL	(Optional) The URL should point to the location where all the necessary information for a call to the API can be found.
Web page	URL	(Optional)
Documentation	URL	(Mandatory)
Location	String	(Optional) Either containing latitude and longitude or the name of the country, the state and the city.
Definition language	String	(Optional)
Development status	Enumeration	(Mandatory) Possible values are: *a*) starting, *b*) in development, *c*) testing, *d*) fixing, *e*) stable, *f*) deprecated and *g*) unknown.
Release date	Date	(Mandatory) Either a past date since when the API is available or a future date to indicate when it will be made available.
Compensation	String	(Optional) Should clarify in a comprehensible way how compensation for using the API works. Either a short text like "1 cent per 100 calls" or a reference to a license or service level agreement.
Granularity	Number	(Mandatory) The number should indicate the level of granularity of the API. Instead of a clear cut design like most social networks use a more tolerant approach is chosen with a value denoting the API type. The proposed interpretations are: a value of 0 would indicate that the API describes properties or functions on a class level, a value of -1000 would indicate that the API describes a single function and a value of 1000 would indicate that the API is structured like a module or package. Values in between would indicate a mixture of the styles.

Table 3.2: Properties of an API in the API network

Relation	**Endpoint**[a]	**Cardinality**
Description & Constraints		
belongs to	Cluster	n:m
(Optional, Directed)		
supervised by	Actor	n:m
(Mandatory, Directed)		
applies	Pattern	n:m
(Optional, Directed)		
throws	Exception	n:m
(Optional, Directed)		
similar to	API	n:m
(Optional, Undirected) Each *similar to* relation should also store a value indicating how similar the two APIs are and the aspect for which this similarity is provided. Commonly the values for the similarity are between 0.0 (regarded different) and 1.0 (regarded the same). In most cases the aspect of interest for similarity between APIs would be *functionality*, depiciting alternatives for an API to solve the same or a similar problem.		
inherits from	API	n:m
(Optional, Directed) Can also store a level to indicate transitive inheritance and how long the shortest path for the inheritance is.		
predecessor	API	1:m
(Special, Directed) It is mandatory unless it is the first version of the API. An API can have only one predecessor, but it is possible to provide two or more successors for one API version and therefore branch the version tree.		
consists of	API	n:m
(Optional, Directed) Allows decomposing APIs if necessary. Target APIs must have a lower *Granularity* than the source API.		
depends on	API	n:m
(Optional, Directed) Can also store a level to indicate transitive dependencies and how long the shortest path for this dependency is.		
applied in	Event	n:m
(Optional, Directed)		

[a] Only the target endpoint is provided since the source of all relations is an API.

Table 3.3: Relations of an API in the API network

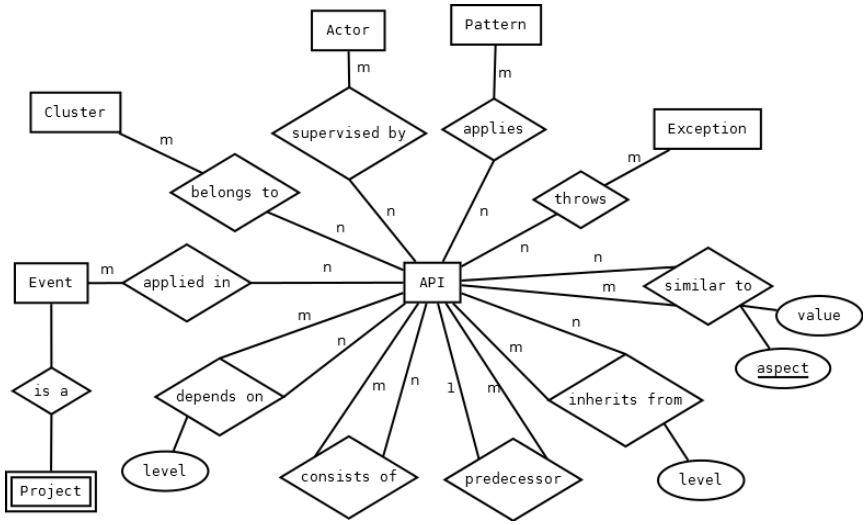

Figure 3.8: Relations of an API network

into several: *a*) *Version* providing an identifier for the different versions, *b*) *Predecessor* stating the previous version and *c*) *Development status* which can declare an API obsolete through the deprecated status.

3.2.2 API Network Visualisation

An elaborate graphical notation for the APIs is considered not important here. It is not necessary because all the described applications in this thesis focus on APIs as vertexes (or nodes) and show different relations between them. Modelling elements used in other modelling languages are not present, like activities, actors, roles, sequences or entities. Therefore using an icon which indisputably identifies an API and distinguishes it from other elements is not necessary since in most cases all the visualized elements will be APIs. A different visualization in form of a report, a list or a table can additionally provide the desired information in a more readable way. These usually do not rely on an icon for the described concepts either.

Therefore a simple notation for an API is chosen consisting of *a*) one shape, *b*) a colour filling the shape and *c*) a size for the shape. This simple notations provides some benefits compared to an elaborate icon. First it is easier to implement. Second different shapes can be used to visualize information taking values from a set of known values. In some cases the shape could also be based on a discrete numerical value, for example with the amount of edges a star shape has. Third the colour of the shape can be used to visualize information about the API either restricted to an enumerated set or a range of numeric values. Fourth the size of the API can be

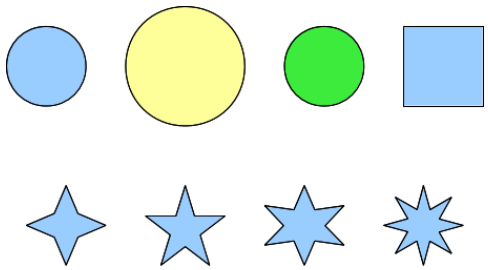

Figure 3.9: Examples for different notations which can be used to visualize different aspects.

used to visualize attributes which take values form a numeric range. This is possible, because the notation is not restricted to a certain shape, colour or size as it would usually be with an icon. Examples for enumerated values visualized through the colour or the shape would be the used definition language, the development status, the supervisor or the location (reduced to either a city, a state or a country) or whether a property has a missing value. Examples for numeric ranges which could be visualized through the colour or the size of a shape are the granularity, the number of predecessors, the number of dependencies or the amount of differences between today and its first release. Some examples as inspiration for the reader can be seen in figure 3.9.

Considering that an API network can contain a large amount of nodes it is not reasonable to rely on manually creating visual models. Also viewing the whole network at once would probably be counter-productive. An example for why can be seen in figure 3.10 where not enough space is available for the amount of elements to show. Therefore filtered views showing only the parts of the network which are of interest makes more sense. The part of the API network could be limited either through value thresholds (requiring that certain property values are smaller, larger, equal or contain a specific value) or the existence of certain relations. Regular expressions could also be used to filter some of the textual properties of APIs. This reduces the amount of APIs to show, providing a better overview for a certain part of the API network. The remaining APIs can then be automatically positioned using one of the many positioning algorithms found in literature and on the internet. The topic of positioning will not be further elaborated here, since it could cover a master thesis of its own.

3.2.3 API Network Processing

To structure and store information about APIs itself can be useful, but to further increase the utility of the API network some possibilities of processing the network are described here. This section starts with some easy processing of the finding transitive relations and an overview of

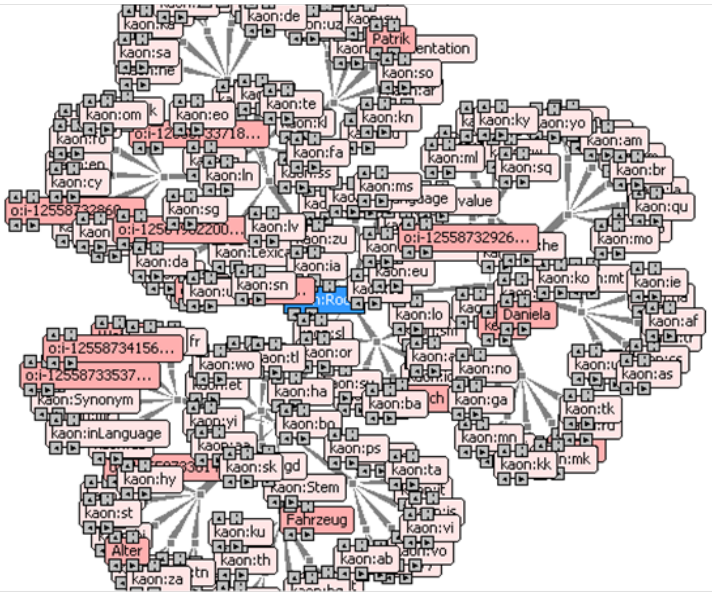

Figure 3.10: Trying to visualize many elements at once.

general measures, turns its focus to the clustering the APIs and finishes with comparing and determining the similarity between two APIs.

Inference of transitive relations

A simple case to utilize the APIs structure which relies on the inference of transitive relations is to analyse the evolution of the API. For this different versions of the same API have to be represented as separate entities and are linked with the directed *predecessor* relation. Using a graph search algorithm a chain of all predecessors, successors as well as branches can be determined for an API and they can be analysed how their attributes and relations have evolved. To find all predecessors one has to simply follow the *predecessor* relation up to the root, since an API can have only one predecessor in this network. However to also find all branches an algorithm similar to a depth-first or breadth-first search can be used. The goal is not to find one specific API, but instead remembering all reachable APIs until no new APIs can be discovered. In a similar manner all inherited or dependent APIs can be determined, providing all the “ancestors” or other dependent APIs for which one has to look out for.

Here a simple algorithm based on breadth-first search using the desired graph and the starting point as an input is described. It discovers all distinct vertices that can be reached from the starting point, which can be one or several vertices. For the sake of simplicity it is assumed that

the desired graph only contains the vertices and relations which are of interest. For example if all the predecessors of an API should be found then only the predecessor edges are present in the graph and to find all transitive dependencies only the dependency edges are used in the graph. Also the input containing the starting vertices is considered a set (referred to as STR). In addition three further sets are needed: one containing all the found vertices and also representing the final result (referred to as RES), another containing the current vertices used for the search (referred to as CUR) and the last which is to contain newly found vertices (referred to as FND). The simple algorithm would be as follows:

1. First the vertices from STR are copied to CUR and RES.
2. As long as CUR is not empty:
    a) Each vertex which is connected to one of the vertices from CUR is added to FND, but only if it is not already present in RES.
    b) Then all the vertices from CUR are added to RES and CUR should be cleared.
    c) After this all the vertices from FND are added to CUR and FND should be cleared.

Once this algorithm stops all the reachable vertices should be in RES together with the starting vertices. The STR set is maintained in order to be able to distinguish between the starting vertices and the reachable vertices. Here it is assumed that sets don't have duplicate entries and that trying to add an already available element is simply ignored. For further efficiency the RES set can be organized as a search-tree or use hash-values to check more efficiently if a vertex is already present. Furthermore RES could not only contain the vertices as a set but also a number assigned to each one through a map. This number would be the counter of how often the loop at step 2 is performed (i.e. 0 before step c is reached, 1 after it is reached the first time, 2 for the second time etc.) and indicate the transitive distance between the starting vertices and the found ones. Another variation could omit adding STR to RES in the first step to also find loops in the graph.

Those transitive relations can be used for example to compare older API versions with newer ones, once all the versions through an APIs evolution have been found or when transitive dependencies are determined to use the in-degree of an API to identify how many other APIs rely on it.

Overview Measures

The network describing the APIs can be used to determine general measures about the objects, relations and properties, similar to how source code can be used to provide an overview over

the complexity of a framework or library by listing the number of classes. The measures can further be applied to articulate statements, like for example about the importance of an API and therefore also its overall priority. These measures can generally be divided into two different types, ones that apply to the graph or a part of it and others which apply to a certain API. For both the work of Antoniou and Tsompa [2008] has been used, since it provides also the formulas[39] which can be applied to both weighted and unweighted graphs (usually by assuming a weight of 1) and can therefore be applied to a broader range of cases. Additionally some algorithms to determine shortest paths can be picked from Cormen et al. [2009], most notably the Bellman-Ford algorithm (see Lawler [1976]), the Johnson algorithm (see Johnson [1977]) and the Floyd-Warshall algorithm (see Floyd [1962]). Furthermore some own simple measures are provided as well for which an implementation should not be too difficult to create.

The following list describes examples for measures applicable for the whole or part of the graph:

- Number of APIs, clusters, etc.
  - How many APIs are currently in the network?
  - How many different description languages are used?
  - What is the average number of APIs per cluster?
- Number of relations of a specific type (also in conjunction with specific endpoints)
  - How many dependencies are there?
  - How many APIs are supervised by a certain actor?
- Number of APIs with a specific attribute value
  - How many APIs use a specific description language?
  - What are the newest APIs?
  - How many stable APIs are available?
  - How many distinct APIs (i.e. not part of the same version tree) are currently in the network?
- Shortest paths and their length for a certain relation type between APIs in the network (Cormen et al. [2009], Lawler [1976], Johnson [1977], Floyd [1962])
- The characteristic path length of the network (Antoniou and Tsompa [2008])

[39] for the specific formulas please refer to the cited work.

- The degree distribution of a graph

This list describes examples for measures for a certain API:

- The degree for a certain relation type (Antoniou and Tsompa [2008])
  - How many dependencies does this API have?
  - How many similar APIs are known?
- The cluster coefficient (Antoniou and Tsompa [2008])
- The shortest paths and their length to all other APIs for a certain relation type (Cormen et al. [2009])

Clustering

Clustering in the context of graphs and especially in this thesis should be understood as the detection of several vertices having some form of connection between them and are therefore assigned to one or several clusters (also called communities in the context of social networks or groups in general). In social networks such clusters are based for example on common interests or the same place of work and the user often joins freely by his own choice, as it is the active part of the network. Those clusters allow to quickly segment huge communities into smaller more manageable groups sharing some kind of analogy (like market segments). Therefore this is considered one of the more important procedures and more attention is devoted to this topic.

However since APIs don't alter the API network themselves either a human or a mechanism or algorithm has to assign them to a cluster. Since manually determining and adding each API to one or several clusters is a tedious task some support should be provided. Several approaches to finding clusters in networks or graphs have been investigated and shall be presented.

One approach which can be used to find clusters is the clique finding algorithm presented in Bron and Kerbosch [1973] (also called Bron-Kerbosch algorithm). It detects and enumerates all possible cliques[40] in a simple graph[41]. However, it should be applied on an undirected graph to properly work. The vertices in such a simple graph represent the parts of interest (in this case the APIs) and the edges between them should indicate some form of relation which is of interest to clustering the vertices according to it. Those edges can either be already available (like the dependency) or created based on other characteristics of the vertices (like the similarity based on the APIs function). Since the Bron-Kerbosch algorithm works on a simple graph it can be

[40] A clique a set of vertices where all distinct vertices are connected to one another (i.e. a subgraph which is complete)

[41] A graph focusing only on edges and vertices, not their attributes or other characteristics.

applied to a broad range of domains, like social networks or API networks. Furthermore it provides all possible solutions from which the best can be chosen.

Two other algorithms working on simple graphs and creating clusters are presented in Clauset et al. [2004] and Newman [2006]. Both are based on *Modularity*, a measure of a graph indicating how well the chosen clusters are based on the edges inside a cluster in relation to the edges between clusters. The formula for *Modularity* based on Clauset et al. [2004] is shown in formula 3.1:

$$Q = \frac{1}{2 * m} * \sum_{vw} \left(\left[A_{vw} - \frac{k_v * k_w}{2 * m} \right] * \delta\left(c_v, c_w\right) \right) \tag{3.1}$$

In this formula Q is the modularity, m is the amount of edges, A is the adjacency matrix (considered here to only contain the values 0 or 1), k is the degree of a vertex, c represents the cluster to which a vertex belongs and the function $\delta\left(c_v, c_w\right)$ equals 1 if both parameters are the same cluster and 0 otherwise. The formula is based on counting the amount of edges inside a community which would simply result in formula 3.2.

$$\frac{1}{2 * m} * \sum_{vw} \left(A_{vw} * \delta\left(c_v, c_w\right)\right) \tag{3.2}$$

This however is difficult to optimize since the optimal solution is to have one cluster with all vertices in it and calculating a value of 1. Therefore some form of penalty has to be introduced for edges missing between vertices of a cluster. For the modularity formula from Clauset et al. [2004] this penalty is in the form of $\frac{k_v * k_w}{2 * m}$, which is "the probability of an edge existing between vertices v and w if connections are made at random but respecting vertex degrees"[42] resulting in considering connections between vertices with low degrees stronger than between vertices with high degrees.

While both described algorithms optimize the modularity, they do so using different approaches. In Newman [2006] whole graph is considered as one cluster at the start and "cut" into two clusters which would yield the highest increase in modularity is determined. For this the $\delta\left(c_v, c_w\right)$ portion of the formula is replaced by $\frac{1}{2} * (s_v * s_w + 1)$, where s is a vector containing 1 and -1 values denoting to what cluster the vertex v/w belongs[43]. The values in s are determined by calculating the eigenvector for the highest eigenvalue of $A_{vw} - \frac{k_v * k_w}{2 * m}$. If the value in the

[42] according to Clauset et al. [2004]

[43] This provides the same result as $\delta\left(c_v, c_w\right)$, since the part in the parentheses is either 2 (same cluster) or 0 (different cluster).

eigenvector is positive for an index then s should be +1 at that index and -1 if it is negative. This can again be repeated on the created clusters to further divide them into smaller clusters.

In Clauset et al. [2004] instead each individual vertex is considered the member of its own cluster and they are then joined together to larger clusters by considering the change in modularity. For this the modularity change ΔQ is first calculated for all cluster pairs of v and w using the formula 3.3 and stored in a matrix.

$$\Delta Q_{vw} = \left(\frac{1}{2*m} - \frac{k_v * k_w}{(2*m)^2}\right) * A_{vw} \tag{3.3}$$

Additionally a vector a is calculated using $a_v = \frac{k_v}{2*m}$, which is necessary to later update ΔQ. After the initial calculation the highest positive modularity increase is chosen and the two clusters (here called i and j) are joined. Since clusters i and j are now together this means that the i-th (or j-th) row and column should be removed from ΔQ. Furthermore after each joining the values for all clusters in ΔQ have to be updated in relation to the new cluster, meaning the j-th (or i-th) row and column. How to update a certain modularity increase depends on the edges between the clusters. Let's consider j' to be the newly joined cluster and h to be the cluster for which the update should be performed. If h is connected to both i and j then the new modularity increase[44] $\Delta Q'_{j'h} = \Delta Q_{ih} + \Delta Q_{jh}$. If h is connected to only i but not j then $\Delta Q'_{j'h} = \Delta Q_{ih} - 2 * a_j * a_h$. If h is connected to only j but not i then $\Delta Q'_{j'h} = \Delta Q_{jh} - 2 * a_i * a_h$. Else h is not connected to neither i or j and therefore it is not necessary to update its value. Also the vector a has to be updated as well using $a'_{j'} = a_i + a_j$. After the updates are finished the next joining can be performed until now more positive modularity increases can be found. The here provided description is not the most efficient one and for more details on how to optimise the algorithm please see Clauset et al. [2004].

A different approach is described in Branting [2010], where the description length for the structure of a graph and its clusters is used. The concept is that "a partition of a graph that permits a more concise description of the graph is more faithful to the actual community structure than a partition leading to a less concise description"[45]. Therefore it evaluates the amount of data required to describe the structure using bits, and elects the shortest description as the best one. In the article the bit-length of the description is based on:

- Bits for the number of vertices
- Bits for the number of clusters

[44] The same applies to $\Delta Q'_{hj'}$ for this and all other update cases.
[45] from Branting [2010]

- Bits for cluster membership of vertices
- Bits for the relations between clusters
- Bits for the relations between vertices

Since the criteria for the description are based only on vertices, relations and clusters it can be employed on simple graphs. In Branting [2010] also three approaches for the calculation of the lengths are provided, two of which are taken from other sources (Roseva and Bergstrom; Chakrabarti) and one presented in the article. However, due to their long nature they will not be covered here in detail for the sake of space. Unfortunately, unlike Newman [2006] and Clauset et al. [2004], Branting [2010] describes only the measure which should be optimized but not how this can be accomplished or implemented in an easy or efficient manner.

In Xu et al. [2010] another approach is given which can be used to classify vertices and therefore assign them to clusters. It uses an Infinite Hidden Relational Model (IHRM), which is based on a Hidden Relational Model (HRM), to predict cluster membership for vertices in a graph. The difference between those two is that the IHRML allows the number of clusters to approach infinity. Figure 3.11 shows an example for a graph and its HRM.

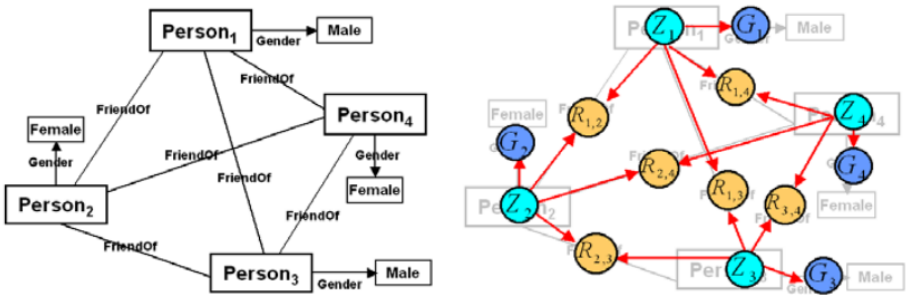

Figure 3.11: A graph (left) and its Hidden Relational Model (right) (from Xu et al. [2010])

Potential relations between two vertices are considered and the R elements represent a random variable which denotes the state of one relation. In this example relations show friendship between two persons and R can either be yes or no. The G elements are similar to R, however they denote the state for a certain attribute value or whole profile. In the example only the gender is considered as part of the profile. The Z elements denote a hidden variable, which "can be thought of as unknown attributes of persons"[46]. A dependency between the three variable types (R, G and Z) is assumed and depicted by the red edges. In the case of the HRM both the profile and the relation state are dependent on the hidden variables. This "implies that if

[46] from Xu et al. [2010]

hidden variables were known, both person attributes and relationships can be well predicted"[47], but more over the state of the hidden variables also depicts the membership to a cluster. The membership to a cluster however is not calculated precisely but rather depicted as a probability that a vertex belongs to a certain cluster. For more information about the calculations I redirect the reader to the original article of Xu et al. [2010].

For the concept of the API network the algorithm as described in Clauset et al. [2004] is used, since the general principle on which it works is simple and still plausible. Furthermore it provides the algorithm for the actual implementation. However one downside noticed when using the algorithm is that it prefers to add vertices with a low degree first instead of creating clusters where all the vertices are connected to one another. The graph in figure 3.12 illustrates this issue, where the first merging of clusters performed is between number 1 and 2, even though intuitively the first joins should be 2, 3 and 5.

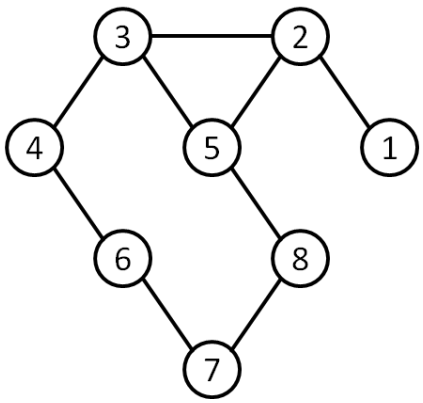

Figure 3.12: Simple example graph, where Clauset et al. [2004] modularity optimization results in joining vertices 1 and 2 first instead of 2, 3 and 4.

To counteract this the algorithm from Bron and Kerbosch [1973] can be used to first create clusters for the cliques and afterwards adding vertices to those clusters based on the modularity gain. An example for this can be seen in figure 3.13, showing a graph with all its cliques identified. Cliques of size 2 or smaller are omitted.

Since the algorithm from Clauset et al. [2004] only allows an object to be part of one cluster, an independent set of clusters[48] has to be found. Now according to Tarjan and Trojanowski [1976] an independent set of a graph G is a clique of the complement of G. So the clique finding algorithm can be reused for this purpose as well by creating a new graph where each vertex denotes a cluster and the edges between vertices indicate that two clusters contain the same

[47] from Xu et al. [2010]
[48] Meaning a set of clusters without overlapping, i.e. containing the same objects.

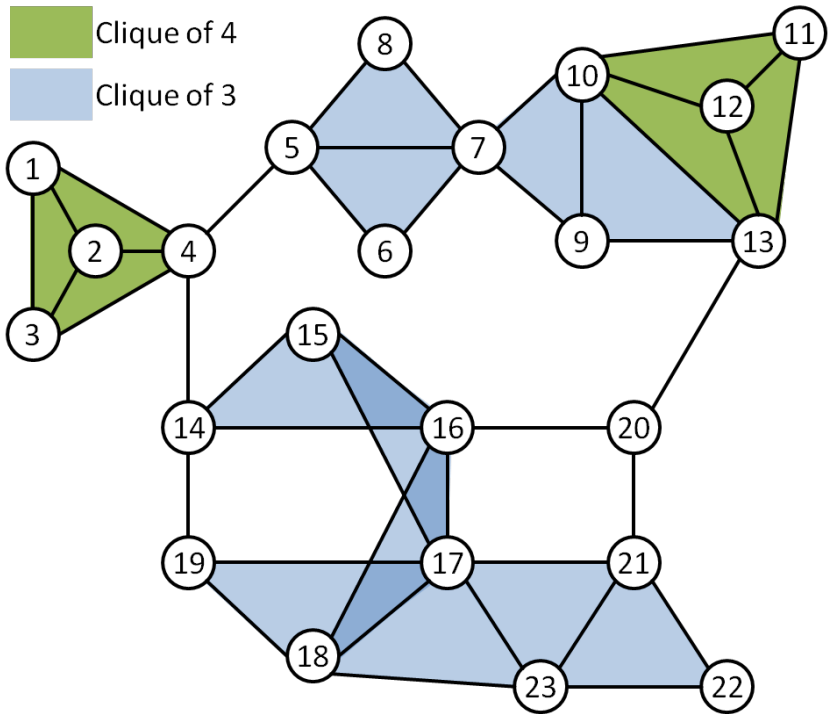

Figure 3.13: Complex example graph, containing several cliques which are adjacent

object (e.g. API). Figure 3.14 shows such a graph containing the clique clusters from figure 3.13. Now inverting the binary adjacency matrix for this results in the complement of the graph. Running the Bron-Kerbosch algorithm on the complement finds all sets of distinct cliques. From those sets the solution containing the most APIs and the fewest clusters is then chosen. This solution can afterwards be used as the input for the algorithm from Clauset et al. [2004].

Furthermore to prevent small and almost meaningless clusters the vertices with a degree of 2 or less can be removed from the input graph used by the clustering algorithms. This destroys bridges between larger clusters so that they do not influence the algorithms and consequentially isolate the individual clusters.

Comparison

The structure of the APIs also allows for the characteristics to be compared to find differences, like changes between different versions of APIs. The result of the comparison of two characteristics of course depends on their type. Generally the comparison should be directed, as in the case of comparing the old version against the newer one. This would indicate what changes

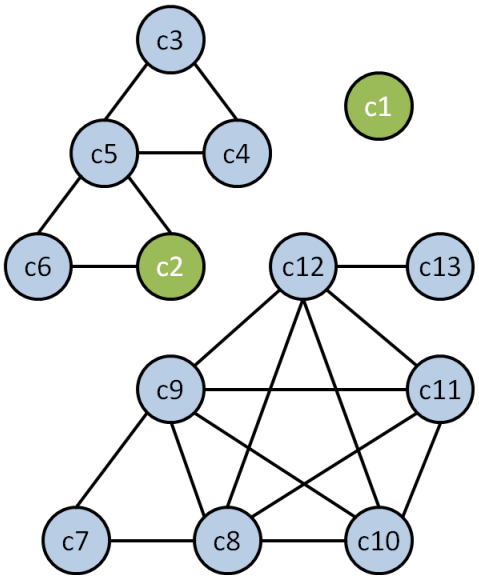

Figure 3.14: Complex example graph, containing several cliques which are adjacent

are necessary to get from the one version to the other, which would not be available with an undirected comparison.

Only the same attributes should be compared (e.g. name with name, signature with signature etc.). In the case of a string value the differences can be provided similar to the comparison of two text documents[49] with the difference to treat one word as one line and to highlighting the differences in the string. Those changes can be *remove*, *change* or *add*. Enumerations and URLs can be treated in a similar fashion. For numbers and dates the simple difference by subtracting one value from the other should be enough, where values of 0 can be omitted since they indicate no change.

This can cover the general characteristics like the name or signature. The relations can be covered similar to a string, however with the *change* also considering the relation endpoints (i.e. something in the endpoint has changed). Furthermore besides the general characteristics some of the previously presented overview measures can be compared as well.

Similarity

Besides comparing two APIs for changes, they can also be processed to determine a similarity. Similarity simply denotes that some features of two things are alike. However, unlike the

[49] Like the diff command in Linux.

comparison that is directed, the similarity should be undirected (i.e. apply from the point of view of both APIs). The chosen features to determine similarity can however be quite different. Depending on which are chosen also the statement that two things are alike changes. For example the similarity can be determined by comparing the friends of people in a social network, the interests of the persons, the school and place of work or the names. Depending on which of those characteristics is chosen different things can be inferred. Comparing the friends of two people could lead to identifying friend preference and proposing new friends. Comparing the names could help identify the affinity to a certain family. Calculating a similarity based on the school and work place could help classify people into certain domains. In the context of this thesis however the similarity of APIs is mostly based on the functions they provide. This should allow to find alternatives to an API which does not properly satisfy the needs of the user. This function can be compared to the friend recommendation of a social network.

Generally the similarity can be explicitly described in a graph as an edge between two vertices and often contains a number indicating how similar those two are considered. To a certain degree this similarity can be calculated automatically. Certain clustering algorithms are for example based on deducing the similarity first and assigning the vertices to the clusters based on it, like Xu et al. [2010] where the probability of belonging to a certain cluster is based on the (hidden) variables of a vertex.

This thesis however will concentrate on a simple approach where the similarity is calculated based on the name denoting the functions an API provides[50]. If an aggregated structure of APIs is available (i.e. where one API can contain others) it might be useful to also use the contained parts of an API when calculating the similarity.

This similarity calculation is based on some simple assumptions:

1. The name or the contained APIs names denote the function.
2. The sequence of words in the names is not relevant.
3. The individual words of the names can be identified.
4. There are no spelling errors in the names.

Those assumptions should pose no problem for most used for APIs, where the name hints at their use, like with the functions 'getCustomers' or 'multiplyMatrices'. Also the names often follow a specific convention like camel-case to distinguish the different words.

[50] However the general concept presented here can be adapted to also base the similarity on other characteristics of APIs, like common predecessors or common dependencies

To determine the similarity two APIs are observed at a time and the words of both APIs names are identified and compared between them using the Jaccard index[51]. The formula 3.4 describes this. The comparison of individual words is performed using a simple string equivalence where the case should be ignored. Based on how many words are contained in both API names the similarity is calculated. This is similar to the calculation of probability in statistics where the amount of valid results is divided by the amount of all possible results. Here a word in the name of an API is considered valid if it can also be found in the other API, while the amount of possible words is the sum of distinct words of both APIs.

$$SimilarityValue = \frac{|w_1 \cap w_2|}{|w_1 \cup w_2|} \tag{3.4}$$

In formula 3.4 w_1 and w_2 are sets containing the words in the names of the APIs (1 and 2) to be compared. To determine the intersection two words are considered equal if they have the same sequence of characters, otherwise they are distinct. The sum of distinct words is represented by $|w_1 \cup w_2|$. Additional enhancements to the individual words can be used for better similarity calculation.

One such enhancement is the use of a word stemmer reduce the words to their root word. This allows to bypass small differences between words and rather compare the meaning they represent instead of just the character sequence. This means that the words 'Attribute' and 'Attributes' would be considered the same. Also some API developers use abbreviations, others use different abbreviations and others try to omit them. An example for this is 'Conf' which usually stands for 'Configuration' and might also be abbreviated as 'Config' or just 'Cfg'. Also some abbreviations cover more than one word and are written entirely in upper case (like XML or API). This makes it difficult to find them when expecting camel-case names. To mitigate these problems a dictionary can be provided and the names of the APIs processed using it to identify those abbreviations and make the names uniform by spelling them out. During this processing however the sequence of the words (or rather letters) is of importance.

For this master thesis a simple dictionary has been created based on the APIs of the ADOxx platform containing probably all abbreviations used there and the words they actually represent. Examples for such abbreviations are API (Application Programming Interface), ATTR (Attribute), XUL (XML User Interface language) and XML (Extensible Markup Language). Of course the entries can be reapplied to one another to write out the XML from XUL. This is represented using RDF for which several tools and libraries are available[52] to handle them.

[51] Sometimes also called Jaccard similarity coefficient. additional information about this measure can be found in Naseem et al. [2011]

[52] See `http://www.w3.org/RDF/`

4 Realisation of an API Network

This chapter deals with the implementation of the concept presented in section 3.2. First a short description about the Open Model Initiative[53] and the Open Model Repository is provided. The Open Model Repository can be used to store and provide the API network to the public. Afterwards the actual implementation is described in both text and class diagrams. The implementation concentrates on the parts necessary to allow testing the API network on the APIs from the use case described in chapter 5. Therefore it does not cover all of the described application areas from section 3.1 like the evolution of APIs. However it provides support for the cases which are not fully implemented and it should be not difficult to further extended when needed.

4.1 Open Model Initiative

The major vision of the *Open Model Initiative* (OMI) is "Models for everyone", with the conviction that "models will become one of the most important means for communication between people in general and an essential factor of production for knowledge intensive business in particular" according to Karagiannis et al. [2008]. Working towards this vision it aims to build and support a community of both experts and novices which deals with the difference aspects of models, like creation, maintenance and distribution. To achieve this a portal as well as a meta-modelling platform and services for it are provided by the Open Model Initiative. The portal allows the community to assemble and exchange knowledge through the Internet. In addition, workshops and other measures allow the community to meet, discuss and work together. The provided meta-modelling platform is an iteration of the ADOxx toolkit. Some of the supported services include a service for drawing element notations and exporting them, a publishing service for models and a repository for meta-models called the *Open Model Repository*. More about the Open Model Initiative can be found at `http://www.openmodels.at` (accessed 25.09.2013) and in Karagiannis et al. [2008].

[53] The feasibility study Karagiannis et al. [2008] can be found at `http://cms.dke.univie.ac.at/uploads/media/Open_Models_Feasibility_Study_SEPT_2008.pdf`, accessed 19.10.2013

The Open Models Initiative has been chosen for several reasons. For one the here chosen meta-model approach for the concept fits into the context of the OMI. Second the services of the OMI rely on APIs and could benefit from an API network. Third it provides a means for persistent storage for the here created concept through the Open Models Repository, which will be covered in the next section.

4.2 Open Model Repository

The *Open Model Repository* (OMR) has been developed for the OMI in the course of the master thesis by Schremser [2011]. Since then it has been further developed and extended in the course of the OMI. Its main goal is to support the management of modelling methods. It provides functionality to manage the content, query the available information and user management just to name a few. The structure of the content (or data) of modelling methods is based on the modelling framework described in section 2.1.3 and bears close resemblance to a graph. Objects like *Modelling Element*, *Modelling Class*, *Project* and *Modelling Language* are connected through relations. One example for such a relation is between a *Modelling Element* and a *Modelling Language* indicating what elements are used in what languages. There is inheritance between the different object types, it is however only based on the reuse of attributes and relations. Further documentation and source code of the OMR can be found in the SVN of the OMI[54].

There are several reasons for choosing the OMR to implement the API network concept: *a*) it is the *Open* Model Repository, *b*) it is based on scientific work, *c*) it fits the selected meta-modelling approach and *d*) it can be accessed through Java.

There are some constraints and quirks in the OMR which have to be considered. One is that every object must have a unique ID in the form of a Universally Unique identifier (UUID). Also it is currently not possible to attach data to relations between objects. Furthermore relations must have exactly two endpoints, one source and one target.

These things among others have led to make some compromises when storing the API Network in the OMR:

- Transitive relations will not be stored, because the information about the level of transitivity cannot be attached to the relations. Instead they can be derived on an as-needed basis.

[54] Accessible at `svn://svn.openmodels.at/REPOS/OMCore/OMRepository`.

- Weighted relations will not be stored as well and have to be recreated every time they are needed.
- Exceptions and Patterns should be stored as simple values instead of creating specific objects for them. This is enough for the experiments here since the structures of the Exceptions and Patterns are not further analysed anyway at this point.
- Events will not be added to the OMR since this would require a change in the implemented inheritance hierarchy. Furthermore this would not provide any additional use to the OMR at this point. Instead the APIs are directly connected to projects.

The OMR also provides a search function. It uses a reference object indicating what is searched for and how[55]. For this attribute values are provided and the compare function for each one. It is not necessary to provide values for all attribute. All other objects are compared to this reference object and the amount of exact matches are returned for each object with at least one match.

4.3 Implementation

The API network is implemented using Java due to several reasons: *a*) the Open Model Repository provides access using Java, *b*) a Java implementation is available for the APIs of ADOxx (the use case), *c*) there are many useful libraries and packages available free of charge and *d*) I have the most experience with programming in Java.

One of the used Java libraries is the Java Universal Network/Graph Framework[56] (JUNG). It provides several classes to store many different graphs and allows easy access to their contents. Examples for the different graphs are hyper-graphs, undirected graphs, directed graphs and sparse graphs just to name a few. One key feature of JUNG is the use of Java generics allowing any type of vertices and edges in a graph. Furthermore it provides easy to use and customizable visualization classes based on Java SWING[57] and several different layout algorithms for the graphs. Therefore JUNG is used instead of creating an own data structure for depicting graphs, especially since other developers who use JUNG as well can reuse the here implemented algorithms.

Since the implementation consists of many different parts only the important ones will be described here in greater detail. Class diagrams will be used to provide an overview of the

[55] This is implemented in the form of the *FindObject* class in the OMR.
[56] For more see `http://jung.sourceforge.net/`, accessed 19.10.2013
[57] For more see `http://docs.oracle.com/javase/tutorial/uiswing/`, accessed 19.10.2013

important implemented classes[58]. Furthermore certain classes will be described using normal text as well as tables (see table 4.1). Table 4.2 provides a short overview through the available Java packages of the implementation.

Name of Class	
Description	*A description of the class providing information about its purpose and use.*
Properties	
Property name	*Description of the property. If the name is underlined it means it is a static/class property. Can be missing if a class contains no additional properties.*
...	...
Methods	
Method name	*Description of the method. If the name is underlined it means it is a static/class method. Can be missing if a class contains no additional methods.*
...	...

Table 4.1: Example table for a description

Several conventions are used in the class diagrams. First, access methods for properties are not further described in order to save space. Second, since generic types are employed the names of the classes as well as the properties and return types use the same syntax for generics as Java[59]. Third, some inheritance and implementation relations also provide information on how the generic parts are further restricted in the subclass. For the generic placeholder names V is used for the type of vertices and E for the type of edges, while T does not have one specific meaning and it therefore depends on where it is used.

The class diagram in figure 4.1[60] shows a part of the Java data structure from the Open Model Repository. All classes except *Cluster*, *API* and *Status* are already available in the OMR, while the other classes have been added to enable the depiction and storage of the API network. The classes of the OMR depict both the properties and relations as attributes. The attributes used to represent properties can contain only single values while those representing relations are a *Map* consisting of all the objects on the opposite end of the relation. The *Map* links a *NamedUUID* (UUID = Universally Unique Identifier) to the actual object. A *NamedUUID* contains the ID and the name of an object. A map is used instead of a list because different load strategies are available, one of which only loads the objects name and ID. Furthermore, for

[58] For better clarity all created *interfaces* will also be referred to as a *class* to prevent confusion with *Application Programming Interfaces*.
[59] Using <...> at the end of the class name, where ... can be a placeholder or an existing class.
[60] In this figure the attributes of *User*, *Location* and *Project* are omitted to save on space.

Package	Description
.tester	Contains classes used for testing and show how things are meant to be used.
.gui	Contains classes with the functionality and for assembling the GUI.
.analysis.clustering	Contains classes for determining clusters in a graph.
.analysis.similarity	Contains classes for determining similarities
.analysis.statistical	Contains classes for statistical analysis and overview measures.
.analysis.transformer	Contains classes implementing the Apache commons transformer interface.
.data	Contains classes to store some data in some form.
.data.graph	Contains implementations for graphs and the elements they contain.
.data.graph.helpers	Contains classes to help with different applications of graphs (e.g. filter vertices/edges, clone graphs etc.).
.data.matrix	Contains implementations of matrices. They are mainly used in the clustering implementations.
.htmlconversion	Contains classes for extracting the interface description from the use cases HTML help.
.idl	Contains classes for handling and transforming the interface description for the use case.
.text	Contains classes for processing text/strings.
.util	Contains general utility classes
openmodel	Contains classes created or adapted for the integration with the OMR

Table 4.2: Overview of implementation packages

each relation type two maps are available, one in each of the endpoints. This allows to navigate a relation in both directions. The finer details like navigation direction and load strategy are handled through OMR specific Java annotations.

The *MMRepositoryObject* class is the root class for anything in the Open Model Repository. It ensures that everything has a unique ID using a universally unique identifier as well as a name. *AnnotateableObject* allows its instances to be annotated with a URI through an *Annotation* object. The *User* class depicts a registered user in the Open Model Initiative (OMI), while *Project* describes a project in the OMI. An instance of *Location* represents a real world location on earth. The classes *User*, *Project* and *Location* are not further described in the class diagram since their details are of no interest. The class *VersionedObject* provides two additional attributes *description* and *comment* as well as properties for the versioning of objects, like *creationdate*, *enddate* and *obsolete*.

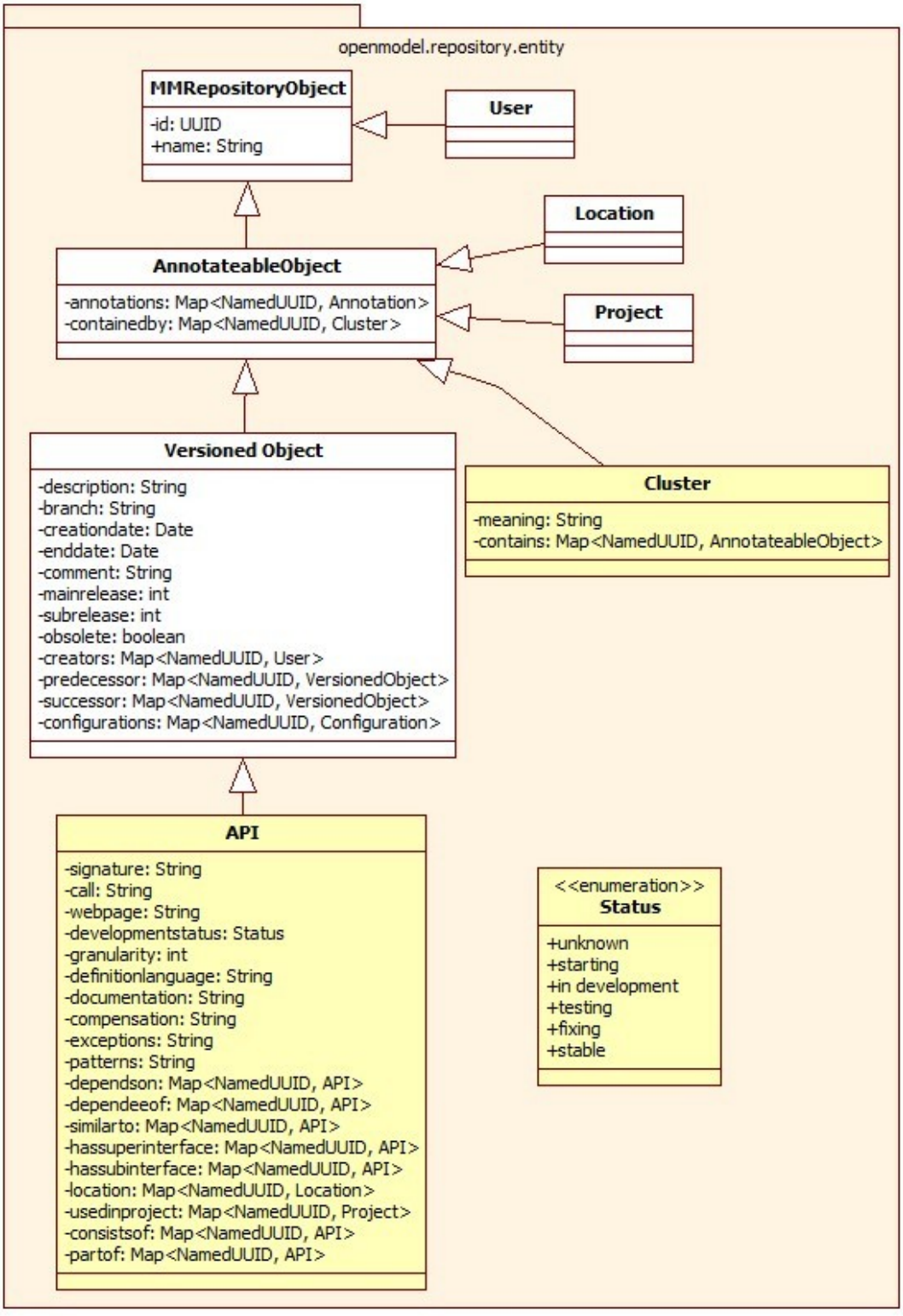

Figure 4.1: Realisation of the API structure in the OMR[60]

The *API* class represents the structure of an API as described in section 3.2.1 with some differences. Several properties are already provided by the super-classes, like *creationdate* for a release date or using *creators* for the *supervised by* relation. Furthermore *Exceptions* and *Patterns* are simply provided as a character sequence instead of explicitly using elements to identify them. On the other hand a relation to a *Location* is used instead of a string to provide the location of an API. Also the *applied in* relation is called *usedinproject* and relates to a project instead of an event. The class *Cluster* allows to store several instances of *AnnotateableObject*, to which an API belongs, together with a meaning. *Status* is an enumeration depicting the several possible states for the development of an API. Deprecated is missing from the implementation of *Status* since *VersionedObject* already provides a means to indicate this using the property *obsolete*.

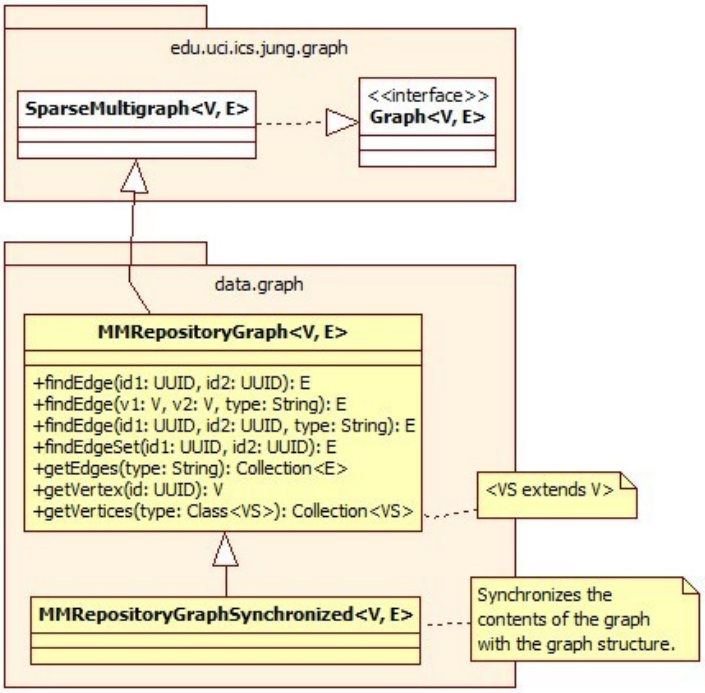

Figure 4.2: Graph classes for use with OM-Repository

Figure 4.2 shows two classes from JUNG as well as two additional implementations created for use with the OMR. The most important part of JUNG for this implementation is the *Graph* class, which allows to store and access the information about a graph. Unlike the OMR which stores parts of the graph locally in each vertex the JUNG graph structure has one global representation of a graph, without requiring the vertices it contains to have any data about the edges between them. This makes implementing algorithms which operate on graphs, like the ones described in chapter 3.2.3, much easier and allows them to work also outside the OMR.

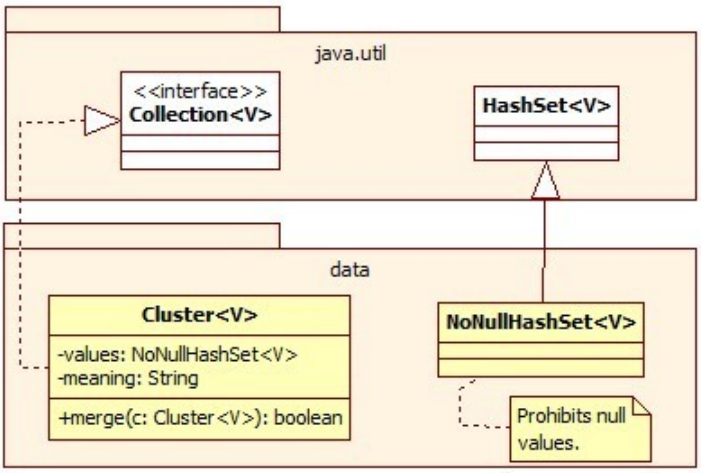

Figure 4.3: Additional data classes

JUNG also provides the special implementation *SparseMultigraph*, which is suitable to store sparse graphs and allows parallel edges between vertices. A multi-graph is necessary for the OMR representation, since it is not prohibited to have two edges with different types between the same vertices. Based on the *SparseMultigraph* two additional graph implementations are provided specifically for the OMR. They are further described in tables 4.3 and 4.4. Both classes also allow to depict only certain parts of the graph in the OMR making them useful for experimentation.

Two new data classes for storing multiple values are shown in Figure 4.3. The *NoNullHashSet* simply extends the already provided *HashSet* class and prohibits null values. Furthermore an additional implementation for a *Cluster* is provided which makes use of the created *NoNull-HashSet*. Unlike the cluster implementation from the OMR this class is more flexible, because it does not restrict the values it can hold to *AnnotateableObject*. Additionally it provides some convenience methods for handling clusters, like merging, transforming and comparing clusters.

Figure 4.4 depicts the classes created to represent a very generic graph which does not require a fixed schema or meta-model allowing for easier experimentation. Generally three important concepts are available: *1*) vertices or nodes, *2*) edges or relations and *3*) attributes. The most basic vertex and edge should provide an identification, which can be used to distinguish two different vertices or edges from one another.

Furthermore a more specific type of vertex which can also store attributes with values is

MMRepositoryGraph	
Description	It is a special graph implementation to handle data from the OMR. It takes care of several constraints posed by the implementation of the OMR as well as providing some comfort functions to easier access the data. *MMRepositoryGraph* requires all stored vertices to extend *MMRepositoryObject* as well as edges to extend *MMRepositoryEdge*, which requires each edge to have a type. The class also ensures that every vertex which is to be stored contains an ID, if one is not present then a random ID is generated automatically. When adding edges the class checks if the provided edge type is allowed between the two vertices. To accomplish this the reflection functions of Java are used, which ensures that the allowed edge types are up to date according to the OMRs Java data structure. Furthermore when adding a new edge it checks if the relation is already depicted in the graph (i.e. same endpoints and same type) and if it is then it will not be added a second time.
Methods	
findEdge	Several methods for finding edges have been added. Those allow a more comfortable access to edges, by allowing to only use the IDs of vertices and by specifying a special type of edge to look for.
findEdgeSet	This method allows to find all edges of a specific type.
getVertex	Through this method a vertex with a specific ID can be retrieved from the graph.
getVertices	This method returns all vertices of a specified type. It is provided since the graph can use a very generic type for its vertices (like *MMRepositoryObject*) allowing to store all its subclasses (like *User*) in the graph as well and by using this method only vertices of a specific type (like *User*) can be retrieved.

Table 4.3: Description of class *MMRepositoryGraph*

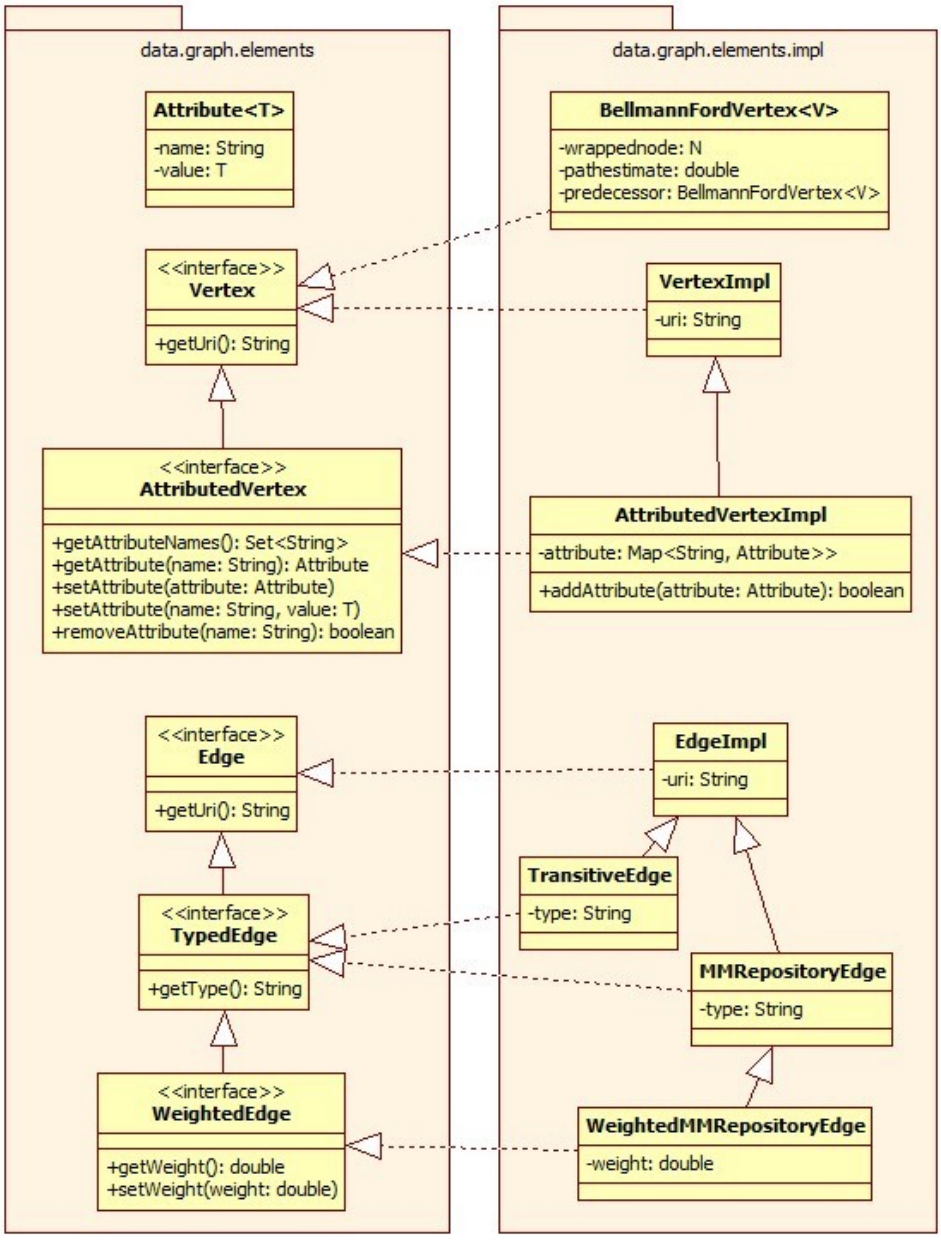

Figure 4.4: Elements implemented for the graphs

MMRepositoryGraphSynchronized	
Description	This class ensures that the global graph layout depicted by it is also propagated to the local representation of the OMR objects in addition to the functionality of *MMRepositoryGraph*. This allows to store a graph more directly in the OMR, without having to make sure that each edge in the graph is also represented in the way the OMR requires it, since this graph implementation already takes care of this.

Table 4.4: Description of class *MMRepositoryGraphSynchronized*

available in the form of *AttributedVertex*. An *Attribute* is very simple and similar to a JSON[61] member. It contains the name of the attribute as well as the value, which can be any type of object. For edges two more specific classes are available, *TypedEdge* and *WeightedEdge*. The *TypedEdge* allows to provide a type in form of a string, so that vertices can be connected through edges with different meanings (denoted by their type). The *WeightedEdge* further extends this by allowing to store a numeric weight with each edge. A numeric weight on the edge is especially useful when determining transitive relations between vertices, because the weight can store the distance between the two vertices, like the length of the shortest path between two vertices.

These classes provide a simple instrument for creating a graph ad hoc and experiment with it. Additionally the *BellmanFordVertex*, a special type of vertex, is created and used by the implemented Bellman Ford algorithm. More information about the vertex can be found in Table 4.5 while the implementation using this class is described on page 69.

Since the here presented implementation of the algorithms relies on JUNG graphs, but the graph is stored in the OMR, the class *GraphHelper* has been created to help managing the graphs. Furthermore some classes to quickly change a graph have been created as well. They are all depicted in figure 4.5, while the *GraphHelper* class is described in tables 4.6 and 4.7.

The class *RemoveCertainVertices* allows to scan through a graph and remove specific vertices. For this it uses a list of criteria. The *VertexCriterion* class provides a method to identify if a vertex fulfils it. One example implementation is *EdgeMinimumWeightCriterion*, which checks a vertices degree against the value specified by the *minweight* parameter. Using this criterion together with the *RemoveCertainVertices* class allows to eliminate vertices which have for example two or less edges. This is useful when clustering the graph, to prevent vertices which form simple "bridges" between clusters or vertices which are only connected to one other

[61] JavaScript Object Notation, for more about JSON see `http://www.json.org/`, accessed 25.09.2013.

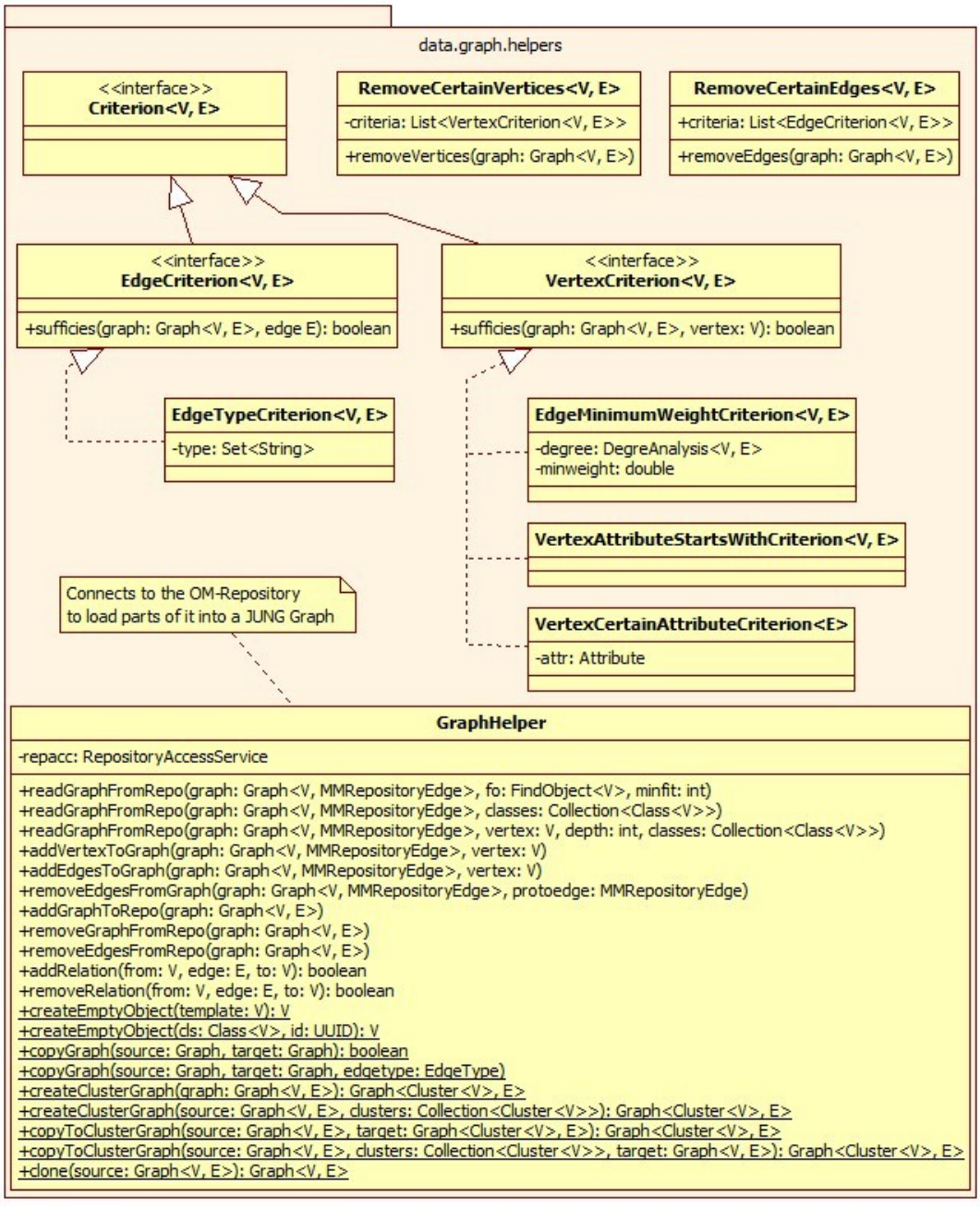

Figure 4.5: Classes aiding the management of graphs

BellmanFordVertex	
Description	A special type of vertex used by the implemented Bellman Ford algorithm. It stores the path information to a specific target vertex. The source vertex is fixed by the algorithm.
Properties	
wrappedvertex	The target vertex for which it provides the path information.
pathestimate	The estimated shortest path between a specific starting vertex an the vertex wrapped here.
predecessor	Stores the vertex prior in the path to the one wrapped by this vertex.
Methods	
getUri	The implementation of the method returns the identifier of the wrapped node. Therefore it is not necessary to provide each BellmanFordVertex with a unique identifier.

Table 4.5: Description of class *BellmanFordVertex*

vertex to influence the cluster result. Also if such vertices are removed it reduces the number of clusters containing only two nodes and running an algorithm on fewer vertices and edges reduces the running time. *RemoveCertainEdges* and *EdgeCriterion* work in a similar manner.

Matrices are also used besides graphs in this implementation. An overview of the available matrix implementations can be seen in figure 4.6. Their main application is in the realized clustering algorithm (more on page 72) as well as providing the result of the Floyd-Warshall algorithm and the Johnson algorithm (more on page 71). The main class here is *Matrix* which provides all important methods to access data in a matrix. Two special versions of the matrix are *SquareMatrix* which should ensure that the number of rows equals the number of columns and *BinaryMatrix* which only allows values of true/1 or false/0.

An *AdjacencyMatrix* extends the *SquareMatrix* by additionally storing the Elements. The first implementations of certain algorithms used the *AdjacencyMatrix* as an input for the graph, which allowed an easier implementation but was more difficult to maintain and use for testing. All currently implemented algorithms expect a JUNG graph as an input. The *ShortestPathsMatrix* is used to return the result of the Floyd-Warshall algorithm containing the shortest paths between all vertices and therefore also extends *SquareMatrix*.

Figure 4.7 shows an overview of implementations allowing to analyse a weighted or unweighted graph using statistic measures based on the ones described in Antoniou and Tsompa [2008]. They have been divided into two types: *a*) *VertexAnalysis* for measures for single vertices and *b*) *GraphAnalysis* for measures over the whole graph or all its vertices.

GraphHelper	
Description	This class helps with managing graphs together with the OMR.
Properties	
repacc	This property provides the access to the create, read, update and delete functions of the OMR.
Methods	
readGraphFromRepo	This method ready parts from the OMR and adds them to the graph object which is specified as a parameter. The graph may already contain vertices. There are three implementations of this method. Each one allows a different style to control what vertices are added. For each vertex all possible edges are added as well. One implementation finds vertices using the search function of the OMR. The search uses a *FindObject* and the method also makes use of an integer (*minfit*) which dictates how many of the specified parts of the object need to be similar so it is added to the graph. Another one uses a collection of types which dictate the types of vertices to read, while the last one additionally provides a starting object and only adds vertices which are interconnected with one another. The *depth* parameter can be used to control how deep the connections between the objects should be followed. Using a negative number for *depth* is equal to "infinite".
addVertexToGraph	This adds a vertex to the graph if it is not already contained and adds all the possible edges from the OMR as well.
addEdgesToGraph	Adds all the edges of a vertex depicted in the OMR to the graph.
removeEdgesFromGraph	Removes all the edges from the graph which have the same type as *protoedge*.
addGraphToRepo	Adds all non existing vertices and edges of the graph to the OMR. It does not change existing attribute values and edges.
removeGraphFromRepo	Removes all the vertices of the graph from the OMR. This automatically invalidates and removes edges which connect those vertices.
removeEdgesFromRepo	Removes all the edges of the graph from the OMR.
addRelation	Adds an edge (or relation) to the OMR.
removeRelation	Removes an edge (or relation) from the OMR.

Table 4.6: Description of class *GraphHelper* (part 1)

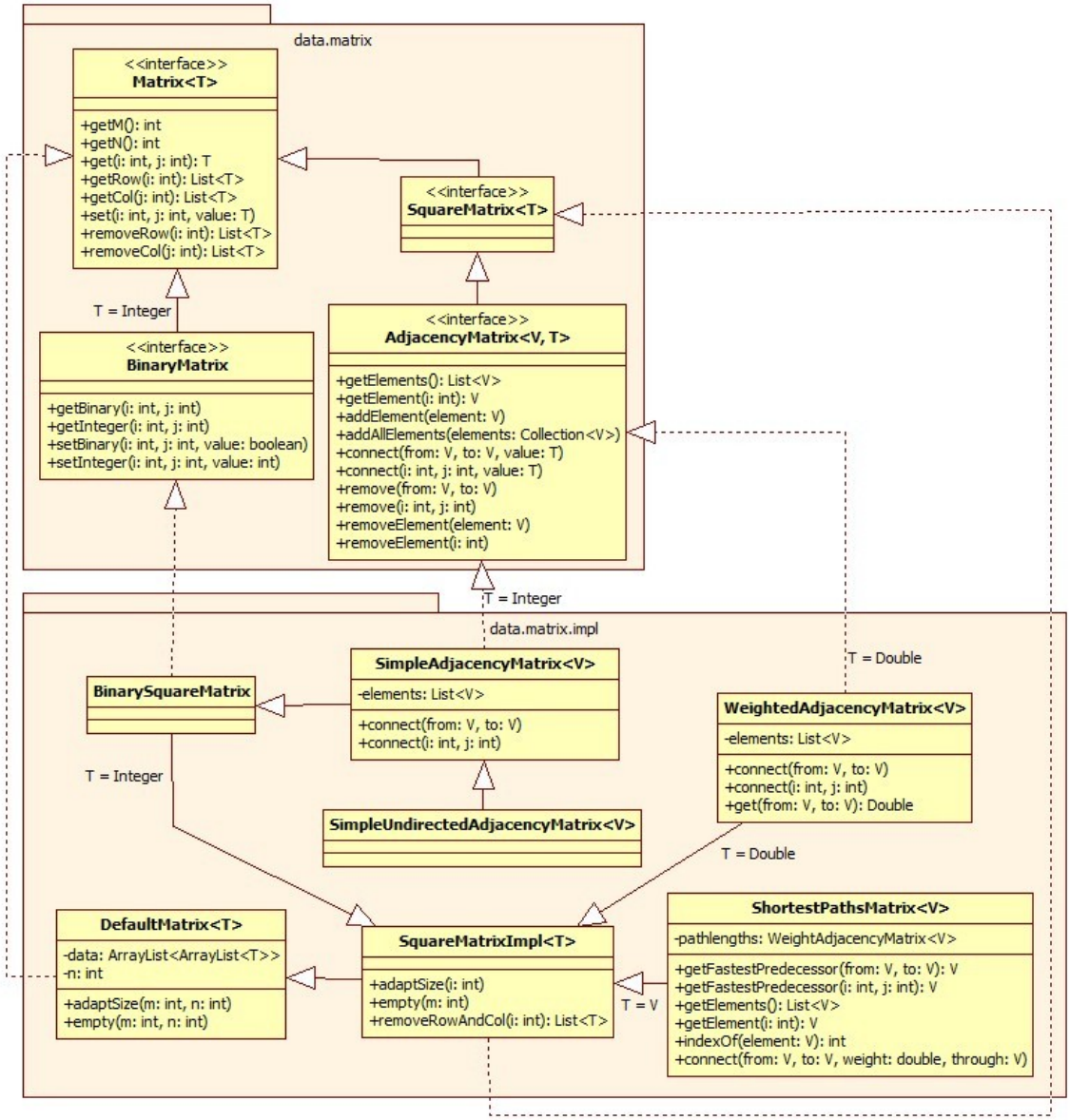

Figure 4.6: Matrix representations

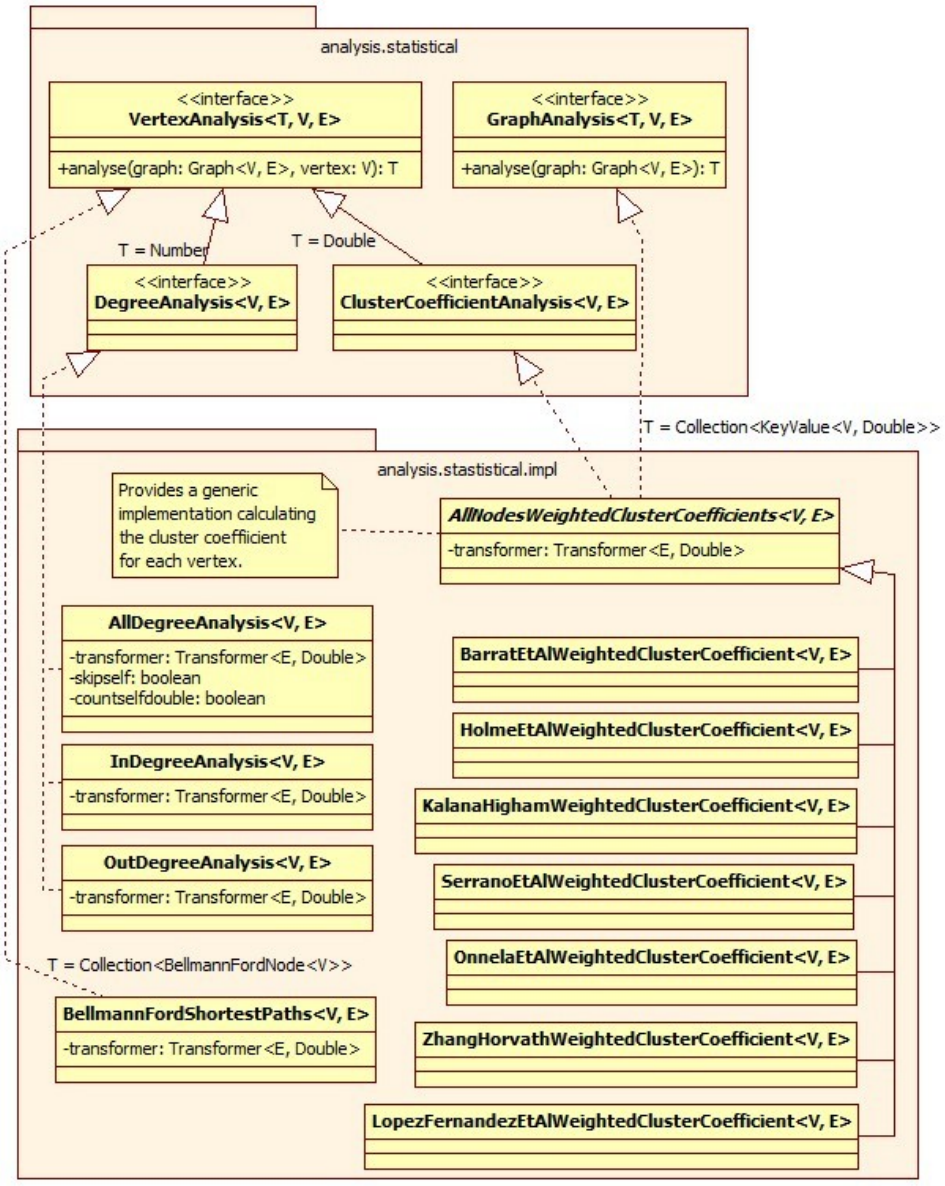

Figure 4.7: Diverse statistical analysis implementations

GraphHelper (continued)	
Methods (continued)	
createEmptyObject	Creates an empty object with an ID but without a name.
copyGraph	Copies all the vertices and edges from the *source* graph to the *target* graph.
createClusterGraph	Creates a new graph where each vertex is inside of a cluster. The second implementation allows to specify an initial clustering for the elements which is reused. The same edges are also in the created graph.
copyToClusterGraph	Copies the vertices and edges into a graph where each vertex is in a cluster. The return value is the same object as the *target* parameter.
clone	Creates a clone of a graph and returns it.

Table 4.7: Description of class *GraphHelper* (part 2)

For the *VertexAnalysis* there are the implementations to determine the degree of a vertex, different formulas for the cluster coefficient for a vertex as well as finding the shortest path using the Bellman Ford algorithm. The available implementations for getting the degree of a vertex allow to retrieve the degree through *AllDegreeAnalysis*, the in-degree (*InDegreeAnalysis*) and the out-degree (*OutDegreeAnalysis*) . The degree analysis further allows to skip edges where the vertex is at both endpoints (i.e. loops) as well as to decide if undirected loops should be counted twice. The individual clustering coefficient implementations use the formulas as described in Antoniou and Tsompa [2008], therefore please refer to this publication for a more detailed description about these formulas. They also implement *GraphAnalysis*, allowing to analyse ever vertex in the graph and return the result for each one.

The implemented Bellman Ford algorithm is based on the description from Cormen et al. [2009] (additional information can be found in Lawler [1976]). The implementation returns a *BellmanFordVertex* (details in table 4.5) for each vertex of the graph. To determine the shortest path to a specific vertex, select the *BellmanFordVertex* wrapping it, and follow the predecessors until the start vertex is reached.

All implementations make use of an Apache commons *Transformer* to make them customizable. The transformer decides which data to use from the edge as its weight. For instance in a shortest path analysis, where vertices would be geographic places and edges would contain both the distance between places and a semantic similarity the transformer would decide which of those two values to use or how to create a new value. An implementation of a *Transformer* could also always return a value of 1, which allows to analyse a graph as if its edges were unweighted.

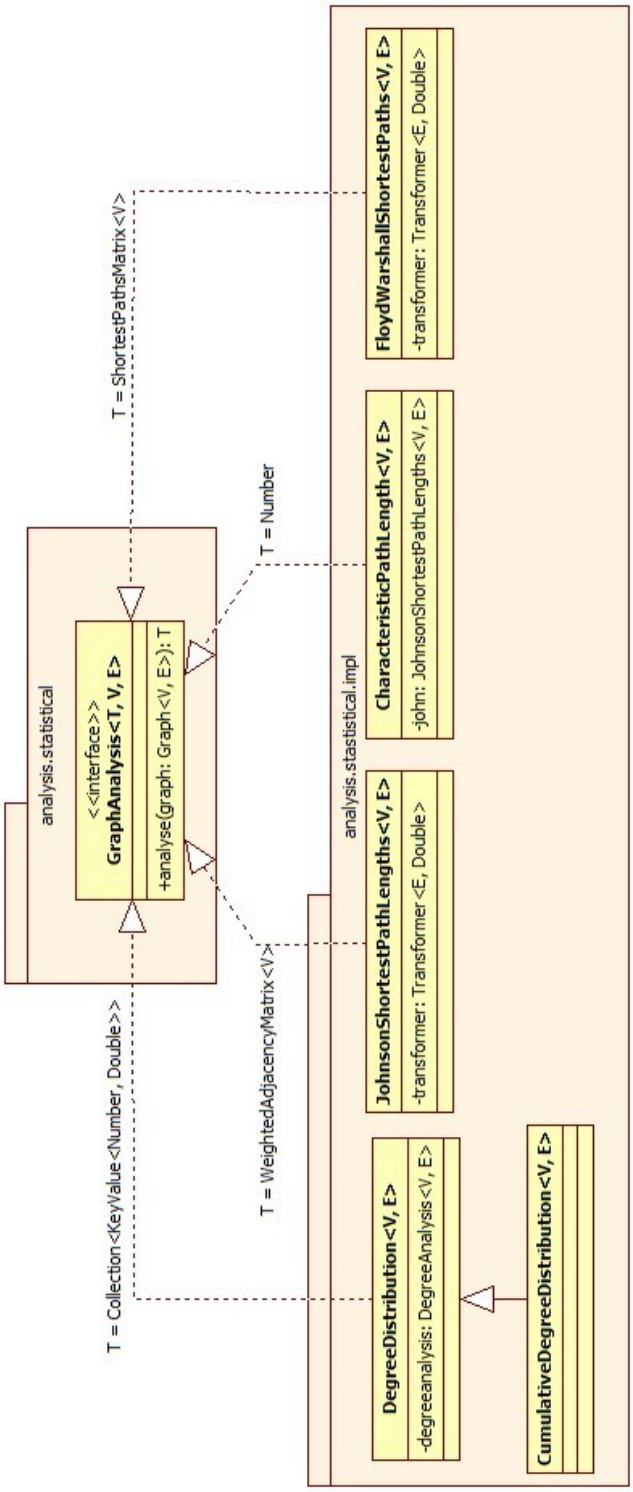

Figure 4.8: Algorithm implementations working on a whole graph

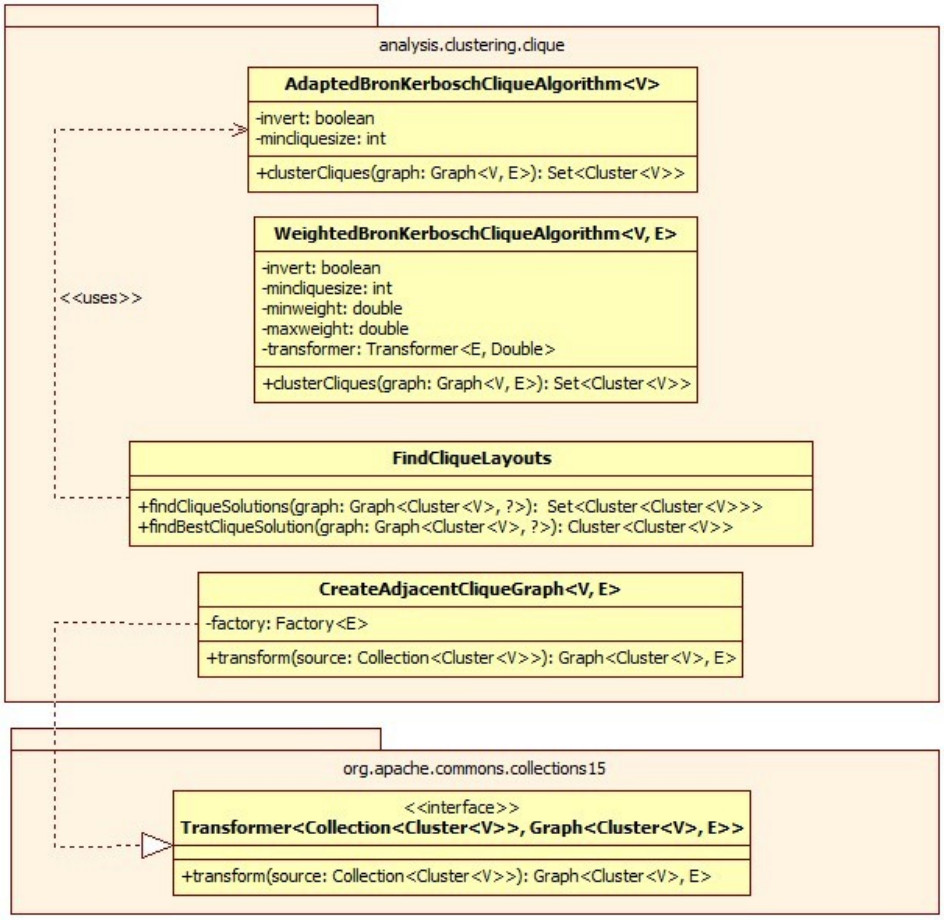

Figure 4.9: Algorithms for clustering based on cliques

In Figure 4.8 additional implementations for analysing a whole graph are depicted. Both *JohnsonShortesPathLengths* and *FloydWarshallShortestPaths* can be used to determine the shortest paths between vertices in a given graph and are based on their description in Cormen et al. [2009]. Additional information about the Johnson shortest-paths can be found in Johnson [1977] and the Floyd-Warshall algorithm in Floyd [1962]. The *DegreeDistribution* and *CumulativeDegreeDistribution* can be used to determine the degrees of all vertices in a graph. For this one of the previous degree distribution implementation can be used (see page 69). These values can then be used to draw a histogram or a pareto chart providing an overview of the degrees. The *CharacteristicPathLength* calculates the average of the shortest path lengths of all the vertices using the formula described in Antoniou and Tsompa [2008].

Figures 4.9 and 4.10 show the implementations for determining clusters in the graphs based on the relations between the vertices. While Figure 4.9 shows the implementations for finding

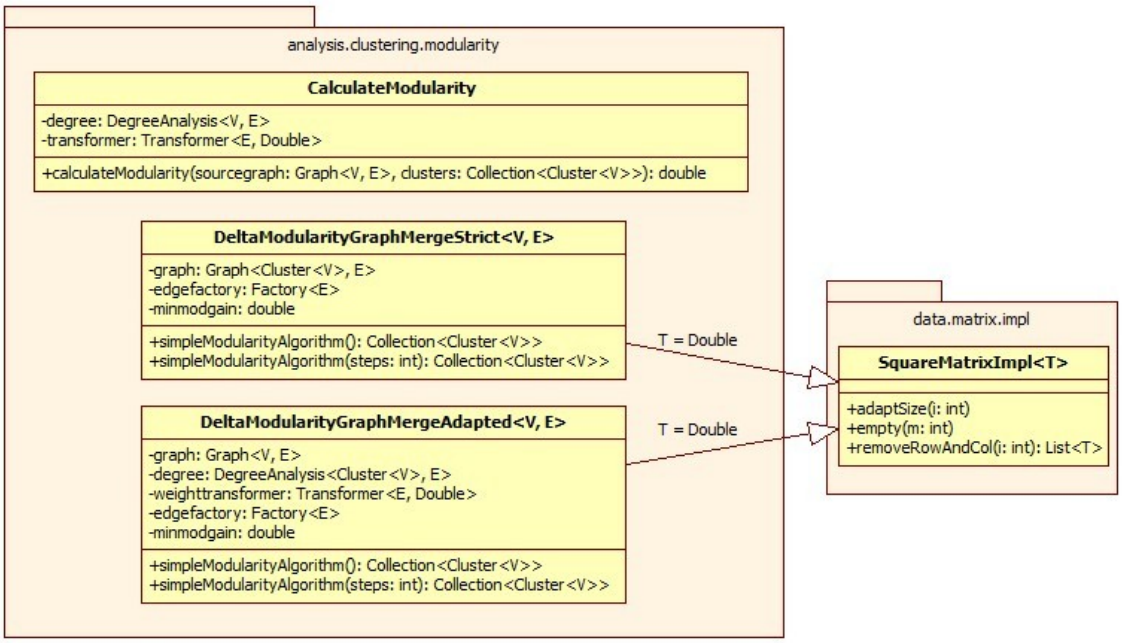

Figure 4.10: Algorithms for clustering based on modularity

the cliques in a graph and creating clusters from those, Figure 4.10 contains the classes using modularity to detect possible clusters.

The *AdaptedBronKerboschCliqueAlgorithm* and *WeightedBronKerboschCliqueAlgorithm* are based on the algorithm described in Bron and Kerbosch [1973]. They contain however some adaptations. For one they allow to create clusters from "inverted" cliques, meaning that all vertices contained are not connected to one another. Furthermore the a minimum clique size can be specified. This prevents creating clusters of a smaller size than desired. The *WeightedBronKerboschCliqueAlgorithm* further allows to consider only edges with a weight within a certain range and ignoring edges outside this range.

To find "the best" solution for several clusters the class *FindCliqueLayout* is used. "The best" is considered a constellation consisting of the most elements[62] and using the biggest clusters. Furthermore every element is only allowed to be contained in one cluster. To find the best constellation the class reuses the clique algorithm finding "inverted" cliques. The used input is a graph where each vertex is a clique and an edge between two vertices indicates that they contain some of the same elements. The internal result contains all constellations of clusters with unique elements and the a constellation containing the most elements and biggest clusters

[62] Here element is used instead of vertex to prevent confusion. An element is a vertex from the original graph which is contained in a cluster/clique. The vertices in the *FindCliqueLayout* are actually the clusters/cliques.

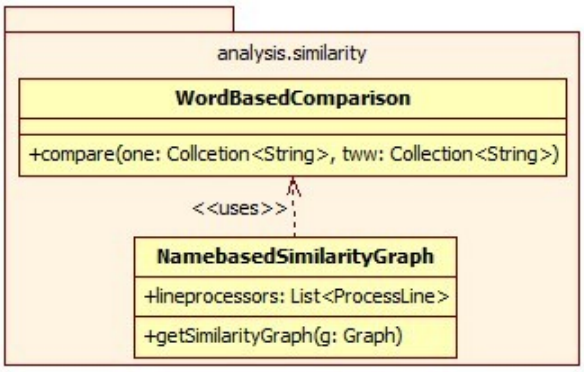

Figure 4.11: Implementations for calculating similarity

is then chosen as the final result.

The *CreateAdjacencyCliqueGraph* class is used to construct the necessary input for the *FindCliqueLayout* class from a collection of clusters. It reuses the clusters as vertices but creates new edges using the provided factory. The edge is created between two vertices if both clusters contain the same element.

Both *DeltaModularityGraphMergeStrict* and *DeltaModularityGraphMergeAdapted* are based on the algorithm described in Clauset et al. [2004]. While the strict version follows more closely, the adapted version allows to also determine clusters in weighted graphs, as well as starting from some already available clusters. For example one could first use the implemented clique algorithm to determine some initial clusters and then further enhance the result using the *DeltaModularityGraphMergeAdapted*. Both classes themselves represent the matrix containing the delta modularity values, which is the reason for extending *SquareMatrixImpl*. Furthermore both provide two methods to determine clusters. One merges clusters as long as the modularity increases while the other method also has a maximum number of merges it can perform, allowing to terminate sooner. The class *CalculateModularity* can be used to directly calculate the modularity of any graph containing clusters based on the formula described in Clauset et al. [2004].

Figure 4.11 shows the implementations for calculating the similarity. The *WordBasedComparison* is a class that compares to sets of words as described in section 3.2.3. Those sets should however first be processed using the available text enhancers (see page 4.3 and figure 4.16). The *NamebasedSimilarityGraph* extends a graph by adding similarities as weighed edges based on the names of the vertices. It uses line processors (see page 75) to enhance the names of the vertices in the graph.

In order to use some of the here presented implementations both *Transformers* and *Factories*

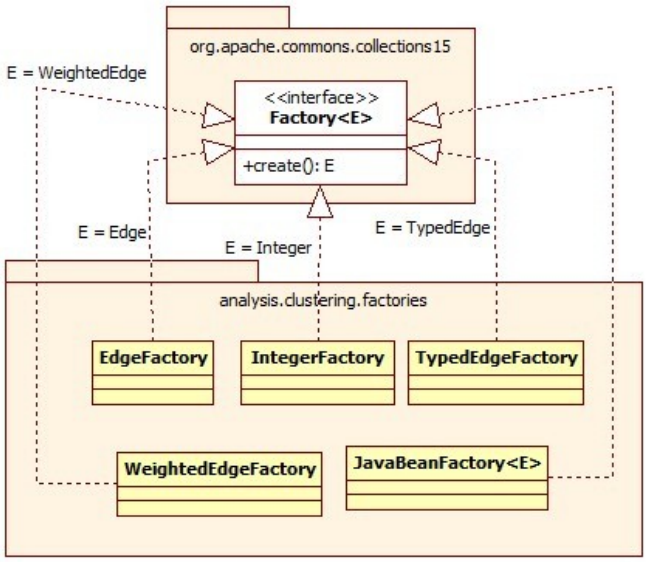

Figure 4.12: Factories for creation of relation objects

are necessary. Several special implementations for factories are shown in figure 4.12 and figure 4.13 depicts the implementations for transformers. All classes implementing *Factory* should be self explanatory, creating a new unique instance of the object. The *WeightedEdgeTransformer* simply uses the weight of an *WeightedEdge*, while the *FixedTransformer* always returns the same value. The *UnweightedTransformer* returns for each *Edge* a weight of 1. This together with the statistical analysis implementations from figure 4.7 provides the results for a graph as if it was unweighted.

The classes in figures 4.14 and 4.15 are used to extract the information about the APIs of the ADOxx platform, while figure 4.16 contains classes used to enhance the extracted information. These are necessary in order to retrieve the information about the APIs for the chosen use case automatically. If another use case uses a different description for their APIs then a different bridge has to be created in order to use them. Additionally the implementations from 4.16 are used for the calculation of similarity between APIs.

Initially the APIs of the ADOxx platform are described in an HTML help file (.chm) containing the XPIDL[63] code for each class. These XPIDL descriptions have been extracted as separate HTML files. However they still contain not used information like the HTML tags for links or text formatting. These have be removed using *ConvertHtmlToXml* and then *RemoveXmlElements*. Besides the HTML elements each code line is also preceded by the line number and can contain

[63] A special interface description language based on IDL to describe Mozilla's XPCOM interface classes. For more see `https://developer.mozilla.org/en/docs/XPIDL`, accessed 19.10.2013

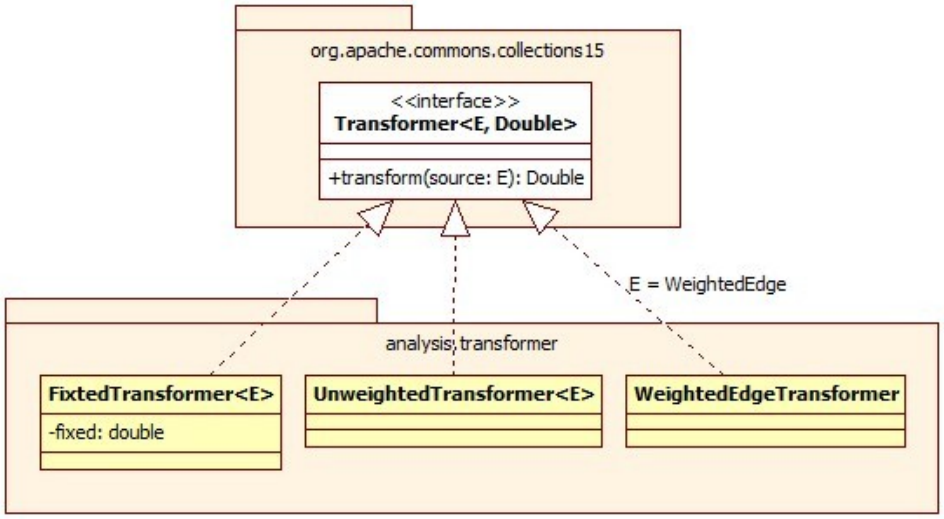

Figure 4.13: Edge transformers for weighted and unweighted graphs

HTML entities which are replaced in the *ReprocessText* class. The result is the pure XPIDL code.

To process the code a grammar has been created using the ANTLR library[64], which creates a lexer and a parser out of an ANTLR grammar description file. Then the class *TransformIdlToObject* is used to process the XPIDL files and create a graph where each XPIDL interface is a vertex and edges between those vertices indicate dependency or sub/super-interface relation. The dependency between the XPIDL interfaces is determined using the parameter types, the return value types and thrown exception of the provided functions as well as the types of the contained attributes. The class *GenerateAdjacencyMatrix* is provided if instead of a graph a directed or undirected adjacency matrix has to be used.

To enhance the data in the generated graph the two process classes *ProcessLine* and *ProcessMultipleLines* are provided. Implementations of *ProcessLine* only work on one single line at a time while the implementations of *ProcessMultipleLines* can work on the whole content and should watch out for newline characters. Their implementations *RemoveLineNumbers* and *UnescapeHtmlEntities* are used by the class *ReprocessText*. Additional implementations are provided to enhance the content used names in the form of *SeperateCamelcaseText*, *UseVocabularyWords* and *StemmWordsSnowball*. The first simply separates the individual words if a camel case style is used to name classes, functions etc. *UseVocabularyWords* is a bit more complicated. It uses a provided vocabulary to write out abbreviations. This is useful since sometimes an abbreviation like “Dir” is used and other times it is spelled out as “Directory”.

[64] For more see `http://www.antlr3.org/`, accessed 19.10.2013

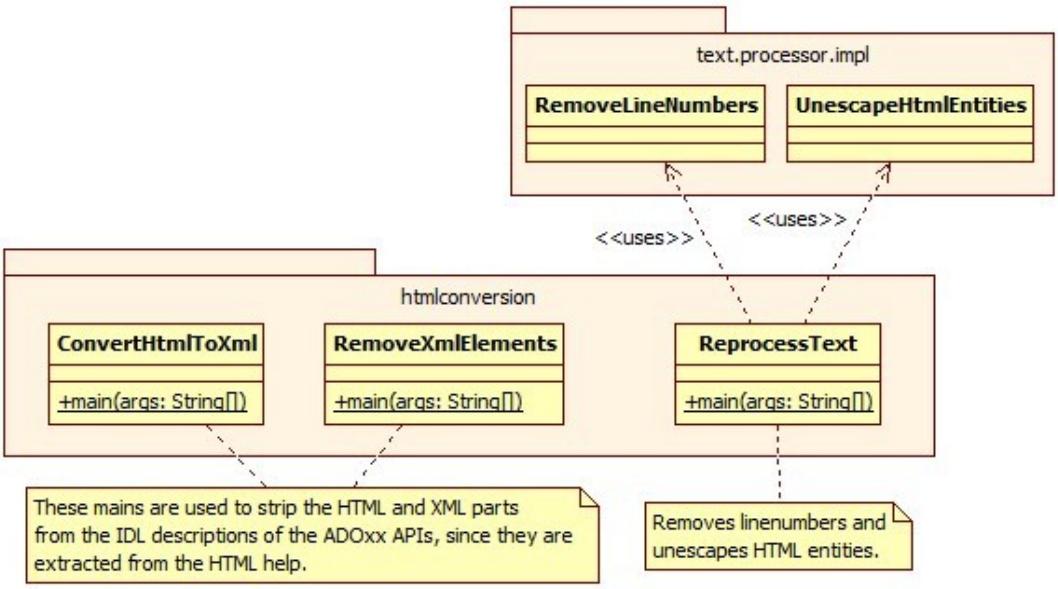

Figure 4.14: Converters from HTML to text

The *StemmWordsSnowball* uses the Snowball word stemmer algorithm[65] to reduce words to their stem. This is useful for the comparison of words.

Additionally some utility classes with commonly used operation are provided and can be seen in figure 4.17. The *FileHelper* makes reading and writing strings into files easier. *JavaBeanHelper* and *MMRepositoryReflectionHelper* use reflection to make processing of the OMR classes describing the data structure easier by returning the corresponding methods to access the data. These are used by the *GraphHelper* implementation. Alternatively *MMRepositoryEdgeHelper* can be used which concentrates on the relations in the OMR. It helps testing and creating relations which are allowed in the OMR and is therefore used by *MMRepositoryGraph* to control edge creation. This is necessary since a JUNG graph usually permits an edge between all vertices independent of their type which is not allowed in the OMR.

4.4 Graphical User Interface

In order to perform the evaluation a more or less simple graphical user interface has been created supporting some functions. Figure 4.18 shows a screenshot of the GUI. There 3 graphs are loaded, the initial graph from the interface descriptions, a graph containing only the interfaces and a graph containing only the specific applications interfaces. Those can be seen on the left

[65] More information can be found at `http://snowball.tartarus.org/`, accessed on 28.09.2013

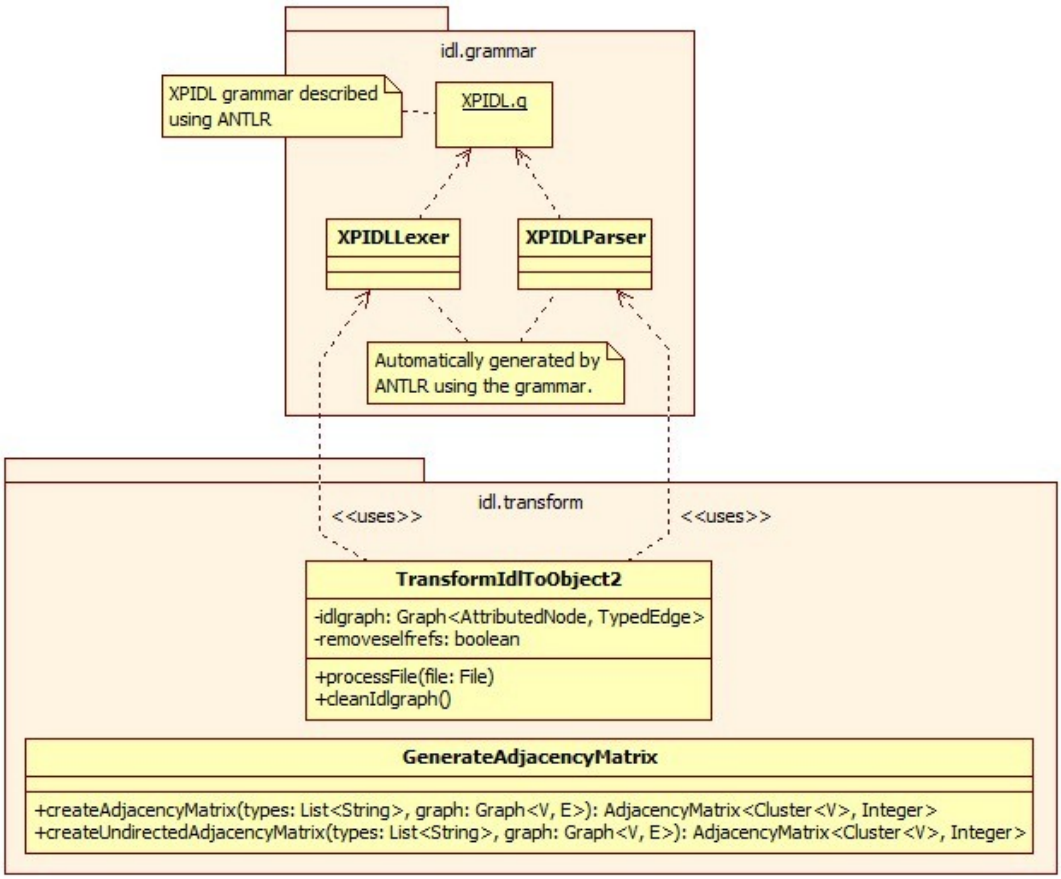

Figure 4.15: Transformers for IDL and matrices

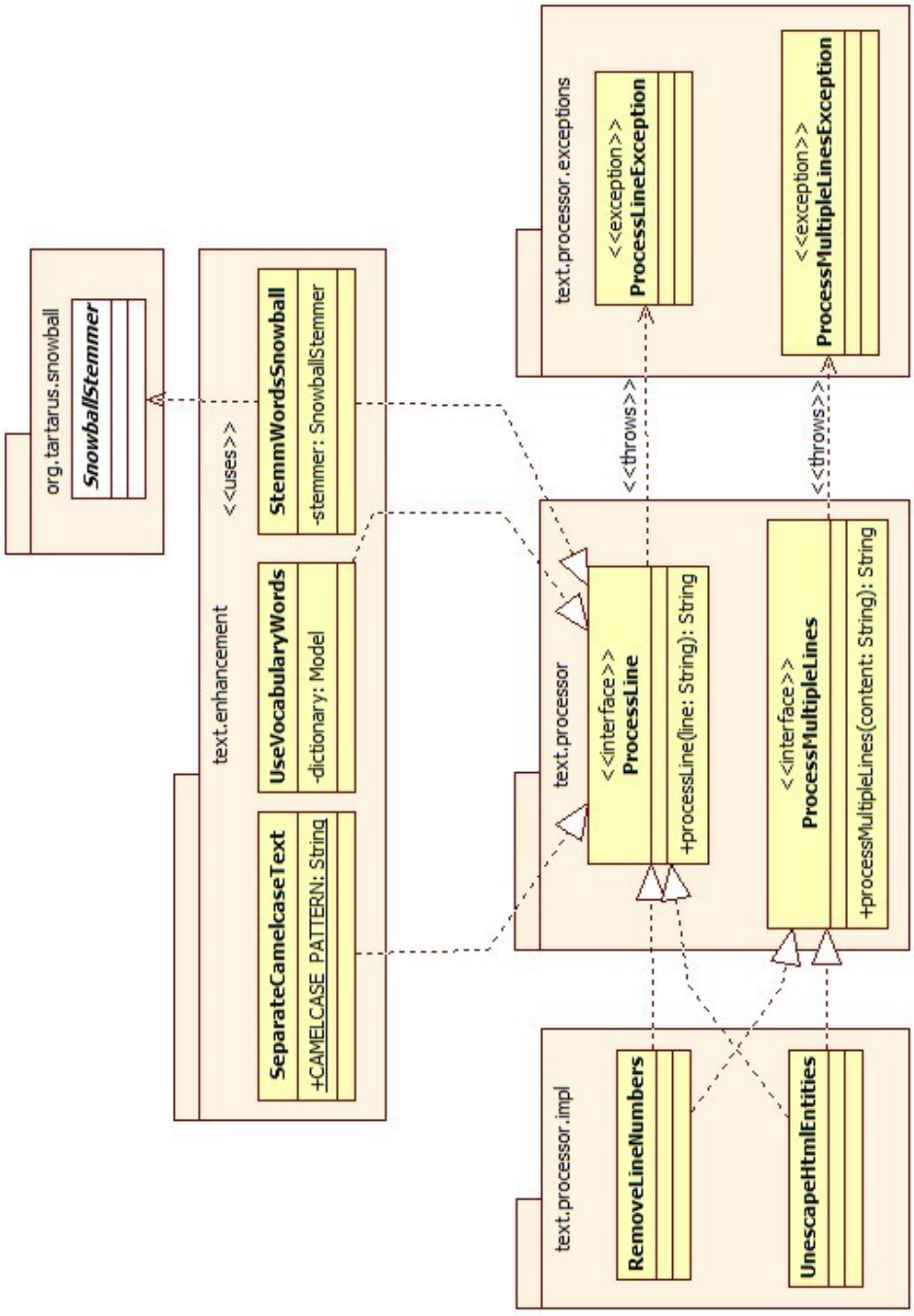

Figure 4.16: Text processing classes

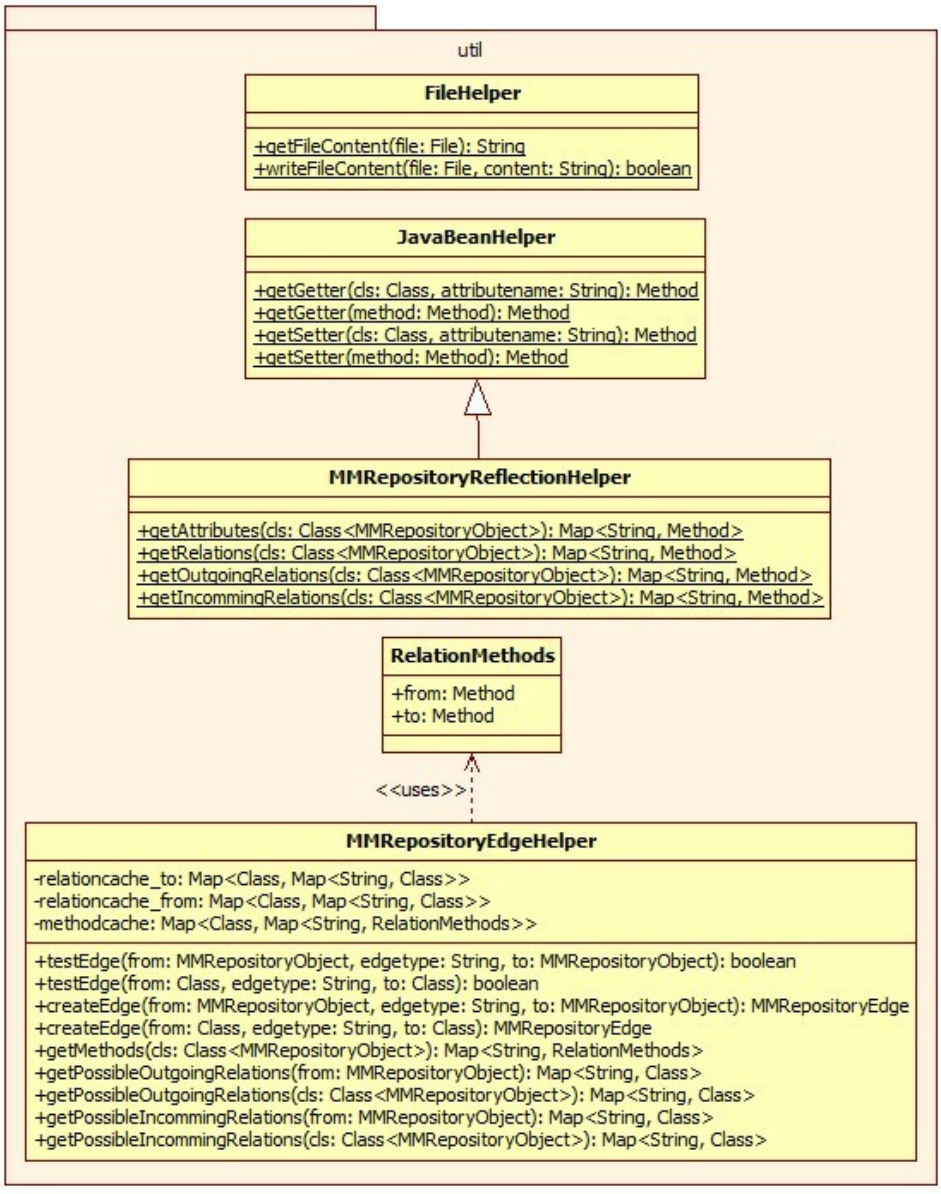

Figure 4.17: Utility classes with commonly used operations

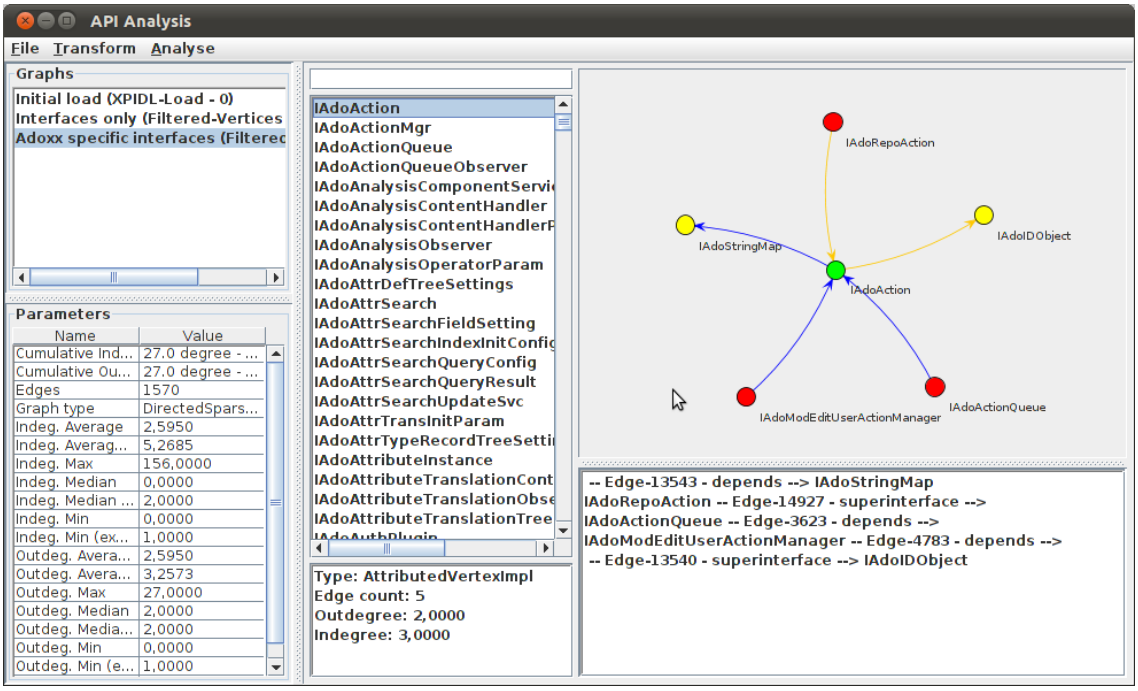

Figure 4.18: Screenshot of the Graphical User Interface

side. Below the list of loaded graphs are some properties of those graphs, like the amount of vertices and edges or the average out-degree. To the right is the graph explorer part. It contains a list of all vertices in the graph with a simple search field above it and some details about the selected vertex below the list. To its right is a visualization of the graph using the algorithms available from the JUNG library with some changes to notation. Under the visualization a list of all the edges incident to the selected vertex is shown.

The visualization shows the selected vertex as well as all the vertices it is connected to. Additionally the colours indicate the selected vertex in green, vertices connected through incoming relations in red and vertices connected through outgoing relations in yellow. The edges are also colour-coded with blue edges being dependencies, yellow edges being inheritances and black for all other types of edges. Transitive edges are also shown using a dashed instead of a solid line. It also supports panning and zooming the shown part with the mouse and mouse-wheel and jumping to a different vertex by selecting it and double clicking. The later can also be performed by double clicking on an edge entry in the list below.

Most of the functions can be accessed through the menu bar at the top and through hotkeys as well. Some of those functions include creating a graph based on the XPIDL descriptions, removing certain vertices or edges from the graph, creating clusters, calculating similarity and performing degree analysis among others. Most of those require that a graph is selected in the

graph list to work. However, some of the functions are handled through the mouse alone. As already said panning and zooming is one of those. Additionally to load the data from a graph to the graph explorer portion it has to be selected and double clicked in the graph list. Another function that is only accessible through the mouse is the performed by double clicking on a cell in the “Parameters" table. This opens a popup window showing the contents of the row in case they exceed the cell size. Double clicking the right mouse button on the vertex list also creates a popup that contains all the vertices in a text format.

5 Use case: ADOxx APIs

In order to test and evaluate the presented concept and created implementation the APIs available in the ADOxx platform that is employed in the OMI will be used. There are several reasons for choosing this application. One of those reasons is the cooperation between the industry and the university during several projects. This cooperation made it possible for me to obtain some knowledge and documentation about the available APIs in their platform. Also ADOxx in general deals with meta-modelling and the creation of modelling methods[66], therefore providing APIs for the chosen modelling method approach for the concept. Additionally descriptions of the APIs provided by the platform were available to me in a help file using an XPIDL syntax. The specifics about how these are accessed have been presented at the end of section 4.3 starting at page 74. Since XPIDL is used for the API descriptions they also follow an object oriented approach where interfaces can belong to a module and contain functions.

Some basic information about the APIs in general can be found in table 5.1. It should be noted that the APIs can generally be categorized into two groups, one containing basic interfaces that are reused by the ADOxx platform and one containing interfaces specific for the ADOxx platform. The evaluation will at times include both, however some will only focus on the ADOxx specific APIs to reduce the amount of data and simplify the evaluation. Also the evaluation will only consider the described interfaces and not their available functions. As seen in the table the API descriptions do not explicitly separate the interfaces into modules.

Now again the evaluation of this use case will not cover everything presented in the envisioned applications in section 3.1. For one because some of the envisioned applications would require data over time which was not available, like for the evolution of the APIs. Additionally the parts of cases where additional humans are necessary will be avoided to prevent a dependency on them. Also certain parts would simply not be meaningful, like the measure for “How many different description languages are used?”, since in this case only one description language is used (XPID). For each application only an excerpt of the result (mostly overview measures) will be shown since the total amount of data would require too much space. The evaluation was carried out on a Linux PC with a 1.3GHz dual-core processor and 4 gigabyte of RAM. The following applications will be presented:

[66] Syntax, Semantic, Procedure and Mechanisms/Algorithms in short.

Total number of ...	
Modules	0
Interfaces	2081
Functions	6433
Total number of ADOxx specific ...	
Modules	0
Interfaces	605
Functions	3697
Average number of ...	
Interfaces per Module	—
Functions per Interface	3.09
Dependencies per Interface	1.77
Average number of ADOxx specific...	
Interfaces per Module	—
Functions per Interface	6.11
Dependencies per Interface	2.24

Table 5.1: Basic information about use case APIs

- Infer transitive dependencies and inheritances
- Calculate similarity
- Dependency clustering (only ADOxx APIs)
- Similarity clustering (only ADOxx APIs)

5.1 Infer transitive dependencies and inheritances

First the implementation for determining transitive relations, in this case dependencies and inheritance is tested. This will be performed for both all interfaces as well as only for the ADOxx specific interfaces. Concerning the runtime of the implementation: it takes about 42 seconds on the largest available graph (8514 vertices and 13995 edges) and less than 2 seconds on a smaller meaningful graph (2081 vertices and 3685 edges) to determine all transitive edges.

Tables 5.2 and 5.3 show measures for all interfaces and the ADOxx specific interfaces respectively. They cover the total amount of relations, the smallest, largest and average amount of outgoing relations from a vertex (the out-degree) as well the median, all for dependencies and inheritances of the direct and transitive variety. The values for the transitive dependencies also contain the direct ones. Additionally the smallest, largest and median values are provided with omitting vertices that have an out-degree of 0 to filter isolated interfaces from interfering.

Direct Dependencies	3685
Min direct Dependencies	0 (1 not counting 0)
Avg direct Dependencies	1.77 (2.51 not counting 0)
Med direct Dependencies	1 (2 not counting 0)
Max direct Dependencies	34
Transitive Dependencies	32687
Min trans. Dependencies	0 (1 not counting 0)
Avg trans. Dependencies	15.71 (22.22 not counting 0)
Med trans. Dependencies	3 (7 not counting 0)
Max trans. Dependencies	272
Direct Inheritances	2038
Min direct Inheritances	0 (1 not counting 0)
Avg direct Inheritances	0.98 (1 not counting 0)
Med direct Inheritances	1 (1 not counting 0)
Max direct Inheritances	1
Transitive Inheritances	3269
Min trans. Inheritances	0 (1 not counting 0)
Avg trans. Inheritances	1.57 (1.60 not counting 0)
Med direct Inheritances	1 (1 not counting 0)
Max trans. Inheritances	6

Table 5.2: Information about direct and transitive dependencies and inheritances of all APIs

As can be seen in both tables, there is a steep rise in dependencies when comparing the direct and the transitive ones. Most notably the generic interface *nsISupports* is used by 921 other interfaces when considering transitive dependencies, while direct dependencies only make 230 of those. From the ADOxx specific interface *IAdoID* is used by 336 other interfaces transitively, while directly only by 156. Those are good indicators that those interfaces have to be taken good care of, since many other interfaces rely on them.

The inheritances however hardly double in any of the cases. Still the highest amount of transitive inheritances shows that the deepest hierarchy for ADOxx specific interfaces is 5 levels, and about 2 on average if a hierarchy is present. However, more than half of the ADOxx specific interfaces do not have an ADOxx specific super-interface and are either derived from a generic interface or none at all.

5.2 Calculate similarity

For the calculation of the similarity only the interfaces without their functions have been used to reduce the amount of vertices in the graph as well as the amount of processing needed

Direct Dependencies	1356
Min direct Dependencies	0 (1 not counting 0)
Avg direct Dependencies	2.24 (3.14 not counting 0)
Med direct Dependencies	1 (2 not counting 0)
Max direct Dependencies	27
Transitive Dependencies	17107
Min trans. Dependencies	0 (1 not counting 0)
Avg trans. Dependencies	28.28 (39.60 not counting 0)
Med trans. Dependencies	3 (13 not counting 0)
Max trans. Dependencies	172
Direct Inheritances	214
Min direct Inheritances	0 (1 not counting 0)
Avg direct Inheritances	0.35 (1 not counting 0)
Med direct Inheritances	0 (1 not counting 0)
Max direct Inheritances	1
Transitive Inheritances	402
Min trans. Inheritances	0 (1 not counting 0)
Avg trans. Inheritances	0.66 (1.88 not counting 0)
Med trans. Inheritances	0 (2 not counting 0)
Max trans. Inheritances	5

Table 5.3: Information about direct and transitive dependencies and inheritances of the ADOxx specific APIs

for each vertex. It has been performed for both all available interfaces and only the ADOxx specific ones based on their names as described in the concept (see section 3.2.3). For this the implemented text processors and enhancer have been used in the sequence: 1) separating based on CamelCase, 2) using the prepared vocabulary for spelling out abbreviations and 3) using the Snowball word stemmer on each word . In addition when only considering the ADOxx specific interface the string “IAdo”, with which each of those interfaces starts, is removed to prevent it from influencing the result. Also different similarity thresholds have been used when deciding to create similarity relations so the available measures give a better overview of the situation. Table 5.4[67] shows the result for a threshold of 0.01, table 5.5 for a threshold of 0.5 and table 5.6 for a threshold of 0.75. The out-degree sums the similarity values from the edges connected to a vertex.

With the current implementation it takes about 17 seconds to calculate the similarity on a graph with 2081 vertices (all interfaces)[68] when the threshold is set to 0.01 and 2.5 seconds with

[67] Here also the “IAdo” has been removed from the interfaces, since otherwise to many similarities were created and a heap space exception occurred.
[68] loading the graph in the user interface (without visualizing) took longer.

605 vertices (only ADOxx specific interfaces). Interestingly those values increase noticeably when setting a higher threshold with 10 (all) or 1.3 (only ADOxx) seconds with a 0.5 threshold and 5.4 (all) or 0.7 (only ADOxx) seconds with a 0.75 threshold. The threshold however only influences the decision if a similarity edge should be added and therefore if the "add edge" operation is performed. The total amount of possible similarity edges is 2164240 for a graph with 2081 vertices and 182710 for a graph with 605 vertices.

All interfaces (2081 vert.)	1010431 similarity edges
Min out-degree of similarities	0 (0.17 not counting 0)
Avg out-degree of similarities	153.35 (153.8 not counting 0)
Med out-degree of similarities	187.63 (187.75 not counting 0)
Max out-degree of similarities	367.67
ADOxx specific interfaces (605 vert.)	19092 similarity edges
Min out-degree of similarities	0 (0.25 not counting 0)
Avg out-degree of similarities	14.41 (14.75 not counting 0)
Med out-degree of similarities	12.56 (12.93 not counting 0)
Max out-degree of similarities	44.32

Table 5.4: Overview measures for the similarity graph with a similarity threshold of 0.01

All interfaces (2081 vert.)	26191 similarity edges
Min out-degree of similarities	0 (0.5 not counting 0)
Avg out-degree of similarities	15.20 (16.36 not counting 0)
Med out-degree of similarities	4.10 (4.59 not counting 0)
Max out-degree of similarities	144.6
ADOxx specific interfaces (605 vert.)	1273 similarity edges
Min out-degree of similarities	0 (0.5 not counting 0)
Avg out-degree of similarities	2.41 (2.96 not counting 0)
Med out-degree of similarities	1.25 (1.95 not counting 0)
Max out-degree of similarities	20.13

Table 5.5: Overview measures for the similarity graph with a similarity threshold of 0.50

From those measures it can be seen that around 1% of all the possible similarities have a value of 0.5 or higher, which means that at least half of the words are considered equal and that only around 10% of the ADOxx specific interface share at least one common word, ignoring the "IAdo" which they all contain.

When looking in greater detail at the result of the similarity calculation with a threshold of 0.5 one can find similarity between for example the interface *IAdoAction* and *IAdoSecurityAction*, *IAdoActionMgr*, *IAdoRepoAction* and *IAdoActionQueue*. The *IAdoAction* is actually used in *IAdoActionQueue* and *IAdoActionMgr* manages those Queues. *IAdoSecurityAction* and

All interfaces (2081 vert.)	3158 similarity edges
Min out-degree of similarities	0 (0.75 not counting 0)
Avg out-degree of similarities	2.44 (4.55 not counting 0)
Med out-degree of similarities	0.75 (0.89 not counting 0)
Max out-degree of similarities	60.70
ADOxx specific interfaces (605 vert.)	120 similarity edges
Min out-degree of similarities	0 (0.75 not counting 0)
Avg out-degree of similarities	0.31 (1.10 not counting 0)
Med out-degree of similarities	0 (0.8 not counting 0)
Max out-degree of similarities	8.80

Table 5.6: Overview measures for the similarity graph with a similarity threshold of 0.75

IAdoRepoAction are also a type of action, where the first is not a sub-interface of *IAdoAction* and the later is. So those similarity relations can be considered useful to find alternatives or the right interfaces for using *IAdoAction*.

Another example is with a group of LADP specific interfaces, where *IAdoLDAPGroup* is similar to *IAdoLDAPUser*, *IAdoLDAPDirectory*, *IAdoLDAPDirectorySettings*, *IAdoLDAPDirectorySettingsValue*, *IAdoLDAPHandler*, *IAdoLDAPTreeSettings* and *IAdoLDAPImportSettings*. All of those are connected because of spelling out the LDAP and they also rely to some degree on one another, like the LDAP directory that uses a string type user ID that is handled by an LDAP user or the import settings that provide group and user information for the import process.

However, there are still similarity edges missing that a human would add. For example the manager interfaces (representing services that manage certain things like actions etc.) are not automatically considered similar by the implementation. Also a human might consider a higher similarity between interfaces like *IAdoClipboardContent*, *IAdoClipboardCutHandler* and *IAdoClipboardService* than the one calculated by the presented concept, which is smaller than 0.4. A possible solution to those shortcomings would be to not consider all words equal in the similarity formula. This means that weights would be assigned to the words, which currently always are 1. The calculation would sum up all weights of the words in both interface names and divide it by the sum of the weights of all used words. The weights could either be provide by the user stating that “Manager” is an important word, or by also considering the sequence of words and putting an emphasis on words at the beginning.

5.3 Dependency clustering

The creating of clusters based on dependency has been performed only on the ADOxx specific interfaces, in order to reduce the amount of vertices and therefore the runtime. Also it did not use the transitive inferred relations. It has been tried only using the modularity algorithm (see table 5.7) and both the clique and the modularity algorithm in that sequence. It should be noted that this is not an evaluation of the algorithms, since this is covered by their respective scientific work (see Bron and Kerbosch [1973] and Clauset et al. [2004]) and instead it is understood that they create groups (clusters) based on the relations between them.

The runtime for performing only the clustering based on modularity took around 6 seconds for a graph with 605 vertices and 1356 relations. Executing the clustering based on cliques however took more 10 minutes for the same graph. Considering that it is meant as a quality improvement for the modularity based algorithms, the trade-off seems not worth considering the huge time loss of the current implementation. Hence it has been aborted before finishing and therefore there are no results available. Most of the time for the clique based clustering is lost not on finding the cliques, but determining a fitting solution once all cliques have been enumerated, so this would be the part that would require improvement to make it feasible.

Cluster count	129
Cluster count containing >1 element	24
Average cluster size	4.67
Average cluster size containing >1 element	20.83
% of interfaces in larger clusters	83%
Inter-cluster relations	200
Average cluster dependency out-degree	3.10 (21.05 without 0)

Table 5.7: Overview measures using only the modularity clustering of dependencies

Looking through the first results from table 5.7 there are many clusters that are not connected to other clusters and many of those contain only one API. However those containing more than one element are rather sizable and looking at the measures one can calculate that they make up around 83% of the graph. And those bigger clusters also have many dependencies between one another as can be seen through the average cluster out-degree (not counting clusters with 0 out-degree). This allows to assume that there is high coupling both inside and between the clusters.

5.4 Similarity clustering

Again the creation of clusters based on similarity is only performed for the ADOxx specific interfaces and without considering transitive relations. Before the clustering, first the similarity graph is created just as the one in section 5.2 with a threshold of 0.5 and 0.75. The overview measures for using the modularity based clustering can be seen in table 5.8 and 5.9. As shown in table 5.1 the interfaces have not been separated into modules. Now the similarity clustering for the platforms APIs could be used to support the creation of modules and assignment of the interfaces to those modules.

The runtime for performing only the clustering based on modularity took around 3 seconds for a graph with 605 vertices and 1273 relations. The clique algorithm has also been tested here, however this time it produced an out of memory exception after 5 minutes and failed to finish.

Cluster count	169
Cluster count containing >1 element	58
Average cluster size	3.58
Average cluster size containing >1 element	8.52
% of interfaces in larger clusters	82%
Inter-cluster relations	40
Average cluster similarity out-degree	0.47 (5.33 without 0)

Table 5.8: Overview measures using only the modularity clustering of similarities bigger than 0.5

Cluster count	490
Cluster count containing >1 element	58
Average cluster size	1.23
Average cluster size containing >1 element	2.98
% of interfaces in larger clusters	29%
Inter-cluster relations	0
Average cluster similarity out-degree	0

Table 5.9: Overview measures using only the modularity clustering of similarities bigger than 0.75

As can be seen in table 5.8, there are more bigger clusters that on average contain fewer elements than during the dependency clustering evaluation. Also different from the dependency case there are fewer inter-cluster similarity relations. Still almost the same amount of interfaces (82%) is part of a cluster with a size bigger than 1.

Looking through some of the created clusters based on similarities of 0.5 or larger one can find ones containing for example *IAdoCardinality*, *IAdoCardinalityContainer IAdoContainer*, *IAdoCoreCardinalityUtils*, *IAdoSimpleContainer* and *IAdoCoreCardinality*, which share some commonality, since they either deal with cardinalities, containers or both (which is the link for the similarity between the two others). Another cluster, containing 18 interfaces in total, has mostly Action, Component, Security and Manager interfaces.

The results from similarities of 0.75 or larger are on the other hand safer, with a cluster containing for example *IAdoComponentMigratableSettings*, *IAdoCompSettingsSerializationSettings*, *IAdoComponentMigratableSettingsHelper*, *IAdoCompMigratableSettingsSerializationSettings*, *IAdoComponentMigratableSettingsFragment* and *IAdoComponentMigratableSettingsManager*. Here all interfaces deal with component settings in one way or another.

Considering those results the created clusters are decent, but could still use some enhancement by a human. Still they provide a good starting point for the human so they do not need to begin from scratch. Depending on the taste, either a smaller similarity threshold can be used and the clusters would then have to be split or a larger threshold where clusters have to be merged. In both cases additional calculated similarities can further be used to support the human in that task.

6 Summary and Outlook

Many people today participate in social networks, where they can connect with friends, check what is new and participate in activities like games or other event. The analysis of those social networks is of some interest, but can this research also be applied to other domains? Using a meta-modelling approach this has been performed for the domain of APIs.

The basics about the three major topics used in this master thesis (meta-modelling, social networks and APIs) have been presented. It is interesting that there are different views of what an API is, but those differences apply on a finer detail. In general an API is seen as an interface to access functionality from an application. A survey of some more or less popular social networks like Facebook, Mendeley and FOAF has been performed as well. In this survey the concepts that are used in social networks and the general interest for processing the network data has been identified. Most of the found work concentrates on finding additional relations like family ties or friendships between the people and determining clusters/groups of people.

The information gained from the survey has then been used to create a concept for the API network. This concept covered all the parts required for a modelling method, namely the modelling language with its syntax, semantic and notation, the procedures that can be employed as well as the mechanisms and algorithms. The modelling language was mostly derived from the structure that can be found in social networks and the mechanisms and algorithms on the processing of those networks. The procedure for the API network was brought closer through envisioned applications, i.e. what it can be used for.

Afterwards, this concept has been realised with consideration of the Open Model Initiative. This initiative has a repository which stores information about projects dealing with modelling methods, and could benefit from an API network to cover the necessary mechanisms and algorithm part for a modelling method. The implementation covers many of the envisioned applications necessary to perform the next step and a graphical user interface for convenience.

The next step was the evaluation of the proposed concept, using the implementation on the APIs of the (meta-)modelling toolkit ADOxx. In general the evaluation led to decent results in most cases. Because of the larger size of the network mostly the overview measure have proven

useful during the evaluation itself. The determination of similarities and clustering yielded decent result, however there is always room for improvement.

One restriction posed by the algorithms used in the proposed concept was that an API can belong to only one cluster. However, depending on the aspect the APIs are clustered upon it makes sense to allow one API in several clusters when automatically creating and proposing them. One possible solution for this would be to perform the clustering based on modularity as described and then further use the modularity to see if certain nodes should also be added to other clusters. Another improvement of the implementation could also consider the weight of the relations when using the modularity for clustering. Another problem was with creating clusters based on cliques. The problem was not finding all the possible cliques, but selecting a fitting clique setup, which took too long for being an improvement for the modularity approach.

Another improvement could address the similarity calculation. The approach of using the vocabulary to identify and handle abbreviations could also consider narrower and broader senses of words. Also the concept could be extended to not only cover APIs, but their implementations as well. There the provided data could help with deploying and using the implementations, where dependencies could be on other implementations or files in general. Additionally the network would then provide some form of documentation for the implementations as well.

Those improvements show that there is always the opportunity for more science, concepts and work on the API network, and in a broader sense in this world in general.

Bibliography

[Antoniou and Tsompa 2008] Antoniou, Ioannis ; Tsompa, Eleni: Statistical Analysis of Weighted Networks. In: *Discrete Dynamics in Nature and Society* 2008 (2008) (cited on pages 43, 44, 65, 69 und 71).

[Branting 2010] Branting, L. K.: Information theoretic criteria for community detection. In: *Proceedings of the Second international conference on Advances in social network mining and analysis*, Springer-Verlag, 2010. – ISBN 978–3–642–14928–3, S. 114–130 (cited on pages 16, 46 und 47).

[Bron and Kerbosch 1973] Bron, Coen ; Kerbosch, Joep: Algorithm 457: finding all cliques of an undirected graph. In: *Commun. ACM* 16 (1973), Nr. 9, S. 575–577 (cited on pages 44, 48, 72 und 88).

[Clauset et al. 2004] Clauset, Aaron ; Newman, M.E.J. ; Moore, Cristopher: Finding community structure in very large networks. In: *Phys. Rev. E* 70 (2004) (cited on pages xi, 16, 45, 46, 47, 48, 49, 73 und 88).

[Computerworld 2000] Computerworld: *Application Programming Interface - Computerworld*. `http://www.computerworld.com/s/article/43487/Application_Programming_Interface`, 2000. – accessed on 08.06.2011 (cited on page 16).

[Cormen et al. 2009] Cormen, Thomas H. ; Leiserson, Charles E. ; Rivest, Ronald L. ; Stein, Clifford: *Introduction to Algorithms*. 3rd Edition. The MIT Press, 2009. – ISBN 978–0–262–03384–8 (cited on pages 43, 44, 69 und 71).

[Elmaghraby 1978] Elmaghraby, Salah E.: The Economic Lot Scheduling Problem (ELSP): Review and Extensions. In: *Management Science* 24 (1978), Nr. 6, S. 587–598 (cited on page 4).

[Erl 2008] Erl, Thomas: *SOA - Entwurfsprinzipien für serviceorientierte Architektur*. Addison-Wesley, 2008. – ISBN 978–3–8273–2651–5 (cited on page 17).

[Floyd 1962] Floyd, Robert W.: Algorithm 97: Shortest path. In: *Commun. ACM* 5 (1962), Nr. 6, S. 345 (cited on pages 43 und 71).

[Hansen and Neumann 2005] Hansen, Hans Robert ; Neumann, Gustaf: *Wirtschaftsinformatik 1 - Grundlagen und Anwendung*. 9. Auflage. Lucius & Lucius, 2005. – ISBN 3–8252–2669–7 (cited on page 17).

[How stuff works 2011] How stuff works: *"What is an API?"*. `http://communication.howstuffworks.com/how-to-leverage-an-api-for-conferencing1.htm`, 2011. – accessed on 08.06.2011 (cited on page 17).

[Höfferer 2008] Höfferer, Peter: *The METON-Architecture: Achieving Semantic Integration and Interoperability Using Metamodels and Ontologies*, University of Vienna, Diss., 2008 (cited on pages xi, 4, 5, 7 und 8).

[Johnson 1977] Johnson, Donald B.: Efficient Algorithms for Shortest Paths in Sparse Networks. In: *J. ACM* 24 (1977), Nr. 1, S. 1–13 (cited on pages 43 und 71).

[Karagiannis et al. 2008] Karagiannis, Dimitris ; Grossmann, Wilfried ; Höfferer, Peter: *Open Model Initiative - A Feasibility Study*. 2008 (cited on page 53).

[Karagiannis and Kühn 2002] Karagiannis, Dimitris ; Kühn, Harald: Metamodelling Platforms. In: Bauknecht, Kurt (Hrsg.) ; Tjoa, A M. (Hrsg.): *Proceedings of the Third International Conference EC-Web*, 2002 (cited on pages xi und 9).

[Lawler 1976] Lawler, Eugene L.: *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart & Winston, 1976 (cited on pages 43 und 69).

[Mackay et al. 2003] Mackay, Daniel ; Noble, James ; Biddle, Robert: A leightweight web-based case tool for UML class diagrams. In: *Proceedings of the Fourth Australasian user interface conference on User interfaces 2003* Bd. 18, Australian Computer Society, Inc., 2003, S. 95–98 (cited on page 4).

[Naseem et al. 2011] Naseem, Rashid ; Magbool, Onaiza ; Muhammad, Siraj: Improved Similarity Measures for Software Clustering. In: Mens, Tom (Hrsg.) ; Kanellopoulos, Yiannis (Hrsg.) ; Winter, Andreas (Hrsg.): *CSMR*, IEEE Computer Socieety, 2011, S. 45–54 (cited on page 52).

[Newman 2006] Newman, M.E.J.: Modularity and community structure in networks. In: *Proc. Natl. Acad. Sci. USA* 103 (2006) (cited on pages 16, 45 und 47).

[PC Magazine 2011] PC Magazine: *API Definition from PC Magazine Encyclopedia*. `http://www.pcmag.com/encyclopedia_term/0,2542,t=application+programming+interface&i=37856,00.asp`, 2011. – accessed on 08.06.2011 (cited on pages 16 und 17).

[Schremser 2011] Schremser, Daniela: *Ein Repository für Modellierungsmethoden: Konzeption und Implementierung*, University of Vienna, Diplomarbeit, 2011 (cited on pages 1, 21 und 54).

[Tarjan and Trojanowski 1976] Tarjan, Robert Endre ; Trojanowski, Anthony E.: Finding a Maximum Independent Set / Computer Science Department Stanford University. 1976. – Forschungsbericht (cited on page 48).

[Xu et al. 2010] Xu, Zhao ; Tresp, Volker ; Rettinger, Achim ; Kersting, Kristian: Social network mining with nonparametric relational models. In: *Proceedings of the Second international conference on Advances in social network mining and analysis*, Springer-Verlag, 2010. – ISBN 978–3–642–14928–3, S. 77–96 (cited on pages xi, 16, 47, 48 und 51).

[Zheleva et al. 2010] Zheleva, Elena ; Getoor, Lise ; Golbeck, Jennifer ; Kuter, Ugr: Using friendship ties and family circles for link prediction. In: *Proceedings of the Second international conference on Advances in social network mining and analysis*, Springer-Verlag, 2010. – ISBN 978–3–642–14928–3, S. 97–113 (cited on page 16).

Appendix A: Source code

1 ANTLR Grammar for XPIDL

```
grammar XPIDL;
// May allow things which are not allowed in XPIDL but are valid for IDL.

options {
  language = Java;
  output = AST;
  ASTLabelType = CommonTree;
}

tokens {
MODULE;
INTERFACE;
SUPERINTERFACES;
METHOD;
PARAMETER;
THROWSEXCEPTION;
ATTRIBUTE;
EXPRESSION;
TYPEDEF;
NATIVETYPE;
}

@header {
package at.patrik.idl.grammar;
}
@lexer::header {
package at.patrik.idl.grammar;
}
```

```
//
// DOES NOT COVER:
// struct => are in %{C++ parts which are ignored
// enum => are in %{C++ parts which are ignored
// sequence => Not used at all
// valuetype => Not used at all
// union => Not used for declaration
// Module scoping
//

/*********** Head Rule ***********/

filehead
: idlcontent* EOF!
;

idlcontent
: preprocessor
| idlinterface
| module
| nativetype
| typedef
;

/************* Rules *************/

preprocessor
/* A preprocessing directive (or any line) may be continued on the next line
   in a source file by placing a backslash character, immediately before the
   newline at the end of the line to be continued. The preprocessor effects
   the continuation by deleting the backslash and the newline before the
   input sequence is divided into tokens. A backslash character may not be
   the last character in a source file. */
: '#include'^ '"'! (.* ('.' .*)?) '"'!
| '#ifndef'^ .* ('#endif'! | EOF!)
| '#'^
;
```

```
module
: 'module' identifier '{' idlcontent* '}' ';'
     -> ^(MODULE identifier idlcontent*)
;

idlinterface
: 'interface' identifier (superinterface)?
(  '{'
( methoddeclaration
| attributedeclaration
| preprocessor
)*
'}'
)?
';'
     -> ^(INTERFACE identifier superinterface?
     preprocessor* methoddeclaration* attributedeclaration*)
;

superinterface
: ':' identifier (',' identifier)* -> ^(SUPERINTERFACES identifier+)
;

methoddeclaration
: 'native'? type identifier '(' (parameter (',' parameter)*)? ')'
     ('raises' '(' exceptions ')')? ';'
     -> ^(METHOD type identifier parameter* exceptions?) ;

type
: (BASIC_TYPES+ | identifier)
;

parameter
: in_out type identifier -> ^(PARAMETER type identifier in_out)
;
```

```
exceptions
: identifier (',' identifier)* -> ^(THROWSEXCEPTION identifier+)
;

attributedeclaration
: 'native'? ('readonly' | 'const')? ('attribute')? type identifier
     ('=' expression)? ';'
     -> ^(ATTRIBUTE type identifier expression?)
;

typedef
: 'typedef' type identifier ';' -> ^(TYPEDEF type identifier)
;

// extra rule so identifiers get directly denoted in the tree
identifier
: IDENT
| LETTER
;

// It is possible to use native types in XPIDL by declaring them using the
// native declaration syntax which works similar to a typedef in C++.
// Uses a .* since the html characters are not properly escaped and there
// can be some weired parts in this but they are not important
nativetype
: 'native' nativeident '(' .* ')' ';' -> ^(NATIVETYPE nativeident)
;

nativeident
: '::'? IDENT ('<' nativeident+ '>')?
;

// Simple other things used for the tree to work!
in_out
: ('in' | 'out' | 'inout')
;
```

```
// Expressions

term
: DIGIT+
| '0x' .+
| IDENT
| LETTER
| '(' expression ')'
;

unary
: ('+'^ | '-'^)* term
;

mult
: unary (('*'^ | '/'^ | 'mod'^) unary)*
;

add
: mult (('+'^ | '-'^) mult)*
;

shift
: add (('<'^ | '<<'^ | '>'^ | '>>'^) add)*
;

and_or
: shift (('|'^ | '&'^) shift)*
;

// expressions are not evaluated
expression
: and_or -> ^(EXPRESSION and_or)
;
```

```
/************* Lexes *************/

// Characters to ignore
WS: (' ' | '\t' | '\n' | '\r' | '\f')+ {$channel = HIDDEN;};
COMMENT: '//' .* ('\n' | '\r') {$channel = HIDDEN;};
MULTILINE_COMMENT: '/*' .* '*/' {$channel = HIDDEN;};
SPECIAL_C_CODE: ('%{C++' | '%{ C++') .* ('%}' | '%} C++')
    {$channel = HIDDEN;};
// The alternative is required since it is found in jsdIDebuggerService.idl
SPECIAL_SCRIPTABLE_CODE: '[' .* ']' {$channel = HIDDEN;};

// Special enumerations
// The basic types of XPIDL, however the syntax descritipn at mozilla.org
// does not contain 'unsigned char' (which is however used in an itnerface)
BASIC_TYPES: ('void' | 'boolean' | 'octet' | 'float' | 'long'? 'double' |
    'short' | 'unsigned short' | 'long' 'long'? | 'unsigned long' 'long'? |
    'char' | 'unsigned char' | 'string' | 'wchar' | 'wstring');

// String types
fragment LOWERCASE: 'a'..'z';
fragment UPPERCASE: 'A'..'Z';

LETTER: (LOWERCASE | UPPERCASE);
IDENTLETTER: (LETTER | '_');
IDENT: IDENTLETTER (IDENTLETTER | DIGIT)*;

// Number types
DIGIT: ('0'..'9');
```

Abstract

Nowadays, millions of users around the word participate in social networks, which also attract some attention from the scientific community looking into analysing and processing those networks. This thesis assumes that a social network, which has a focus on the persons participating in it, can be adapted to work with API instead in order to create an API network. The aim is to use the insights from the work on social networks in the domain of APIs to support developers and also to see what kind of results can be achieved from applying knowledge from one domain in another. One of the benefits envisioned, besides a means for documentation, is to provide alternatives for developers through the determination of similarity between APIs and by clustering them. Another possible benefit would be the identification of important APIs through the network. To allow this a concept for an API network has been created. It is based on a survey of social networks that has been performed as part of this work. This concept mainly focuses on the structure of the API network and how it can be processed. It is then implemented and afterwards evaluated using the APIs from the ADOxx tool. Following the evaluation the results are used to indicate further paths for the evolution of the API network concept.

Note: The implementation is delivered in the enclosed CD.

Kurzfassung

Millionen von Benutzern auf der Welt nehmen heutzutage an Sozialen Netzwerken teil, welche die Aufmerksamkeit der Wissenschaftsgemeinde auf sich zieht und sich unter anderem mit der Analyse und Verarbeitung dieser Netzwerke beschäftigt. Diese Masterarbeit nimmt an, dass ein Soziales Netzwerk, in dessen Schwerpunkt Personen stehen, durch einige Anpassungen den Fokus auf APIs setzen kann um dadurch ein API Netzwerk zu erstellen. Das Bestreben ist es die Einsichten von den Arbeiten an Sozialen Netzwerken in dem Bereich der APIs einzusetzen um Entwickler zu unterstützten und herauszufinden welche Resultate beim Einsatz von Wissen der einen Domäne in einer Anderen erzielt werden können. Neben der Möglichkeit zur Dokumentation von APIs ist ein möglicher Nutzen für einen Entwickler das Vorschlagen von Alternativen, die durch Bestimmung der Ähnlichkeit und das Gruppieren anhand von dieser erreicht werden kann. Ein weiterer Vorteil von dem Netzwerk wäre die Identifikation von wichtigen oder einflussreichen APIs. Dafür wurde ein Konzept für das API Netzwerk erstellt, welches auf den Ergebnissen einer Untersuchung von Sozialen Netzwerken, die im Rahmen dieser Arbeit durchgeführt wurde, basiert. Die Struktur sowie die Verarbeitungsmöglichkeiten bilden den Schwerpunkt in diesem Konzept, das anschließend umgesetzt und unter mit den Beschreibungen der APIs des ADOxx Werkzeuges evaluiert wurde. Nach der Evaluation wurden die Resultate verwendet um weitere Schritte für die Evolution des API Netzwerks vorzuschlagen.

Hinweis: Die Implementierung befindet sich auf der beigelegten CD.

Lebenslauf

Angaben zur Person

Nachname / Vorname	**Burzynski Patrik**
Geburtsdatum	26.09.1985
Staatsangehörigkeit	Österreich
Adresse	Burggasse 57/12, 1070 Wien
Telefon	0043 1 9475818
Email	bigp@chello.at

Berufserfahrung

Datum	Februar 2012 bis voraussichtlich August 2014
Beruf oder Funktion	Projektmitarbeiter beim Projekt: ComVantage
Name und Adresse des Arbeitgebers	Universität Wien - Fakultät für Informatik, Forschungsgruppe Knowledge Engineering; Währinger Straße 29, 1090 Wien
Tätigkeitsbereich oder Branche	Wirtschaftsinformatik
Datum	März 2009 bis August 2011
Beruf oder Funktion	Projektmitarbeiter beim Projekt: plugIT
Name und Adresse des Arbeitgebers	Universität Wien - Fakultät für Informatik, Forschungsgruppe Knowledge Engineering; Währinger Straße 29, 1090 Wien
Tätigkeitsbereich oder Branche	Wirtschaftsinformatik
Datum	Februar 2008 bis März 2009 (nicht zusammenhängend)
Beruf oder Funktion	Projektmitarbeiter beim Projekt: Semantic Culture Guide
Name und Adresse des Arbeitgebers	Universität Wien - Fakultät für Informatik, Forschungsgruppe Knowledge Engineering; Währinger Straße 29, 1090 Wien
Tätigkeitsbereich oder Branche	Wirtschaftsinformatik

Schul- und Berufsbildung

Datum	September 2005
Hauptfächer	Wirtschaftsinformatik
Name und Art der Einrichtung	Universität Wien - Fakultät für Informatik, Währinger Straße 29, 1090 Wien
Datum	September 2000 bis Juni 2005
Hauptfächer	Biomedizinische Technik
Name und Art der Einrichtung	Höhere Technische Lehranstalt - TGM, Wexstraße 19-23, 1200 Wien
Datum	September 1996 bis Juli 2000
Name und Art der Einrichtung	Bundesrealgymnasium 7, Kandelgasse 39, 1070 Wien
Datum	September 1992 bis Juni 1996
Name und Art der Einrichtung	Volksschule, Neustiftgasse 98-102, 1070 Wien

Persönliche Fähigkeiten und Kompetenzen

Muttersprache	Deutsch
Sonstige Sprachen	Englisch - Gute Kenntnisse im Verstehen, Sprechen und Schreiben Polnisch - Gute Kenntnisse im Verstehen und Sprechen
Technische Fähigkeiten und Kompetenzen IKT Fähigkeiten und Kompetenzen	• Elektronik-Kenntnisse (Schaltpläne, Printplattenfertigung etc.) • Umgang mit Office-Programmen (Textverarbeitung, Tabellenkalkulation, Präsentationen) • Erfahrung mit grundlegender Grafikverarbeitung in GIMP • Programmierkenntnisse in Java, JavaScript, C++ und PHP • Erfahrung mit SQL, diversen Servern (Apache, Tomcat, MySQL) und Semantischen Technologien (Semantic Web, RDF)